

# Insertion Sort

<https://cs.pomona.edu/classes/cs140/>

# Outline

## Topics and Learning Objectives

- Specify an algorithm
- Prove correctness
- Analyze total running time

## Exercise

- Friend Circles

# Extra Resources

Cormen

- **Chapter 2** of Introduction to Algorithms, Third Edition
- <https://www.toptal.com/developers/sorting-algorithms/>

# Survey (answer on Gradescope)

- What do you go by (for example, I go by Tony instead of Anthony)?
- What data structures do you know (any amount of familiarity)?
- What algorithms do you know?
- What programming languages do you know?

# Friend Circles Exercise

- Read the problem (about 1 minute)
  - Find the PDF on the course website
- Discuss with group for about 5 minutes
- Discuss as a class

# Warm-Up

6 8 16 -3 8  
-3 6 8 8



## Sorting Problem

- **Input:** an array of  $n$  items, in arbitrary order
- **Output:** a reordering of the input into nondecreasing order
- **Assumptions:** none

|       |        |         |         |       |       |          |        |
|-------|--------|---------|---------|-------|-------|----------|--------|
| Clark | Potter | Granger | Weasley | Snape | Clark | Lovegood | Malfoy |
|-------|--------|---------|---------|-------|-------|----------|--------|

|       |       |         |          |        |        |       |         |
|-------|-------|---------|----------|--------|--------|-------|---------|
| Clark | Clark | Granger | Lovegood | Malfoy | Potter | Snape | Weasley |
|-------|-------|---------|----------|--------|--------|-------|---------|

# Warm-Up

## Sorting Problem

- **Input:** an array of  $n$  items, in arbitrary order
- **Output:** a reordering of the input into nondecreasing order
- **Assumptions:** none

## We will

- Specify the algorithm (**learn my pseudocode**),
- Argue that it correctly sorts, and
- Analyze its running time.

Specify the algorithm



Base - 0

index

0 1 2 3 4 5  
i j

# Insertion Sort

|              |              |   |   |   |   |
|--------------|--------------|---|---|---|---|
| <del>1</del> | <del>2</del> | 4 | 6 | 1 | 3 |
|--------------|--------------|---|---|---|---|

5

1. **FUNCTION** InsertionSort(array)

2. **FOR** j **IN** [1 ..< array.length]

3. key = array[j]

4. i = j - 1

5. **WHILE** i ≥ 0 && array[i] > key

6. array[i + 1] = array[i]

7. i = i - 1

8. array[i + 1] = key

9. **RETURN** array

// right shift

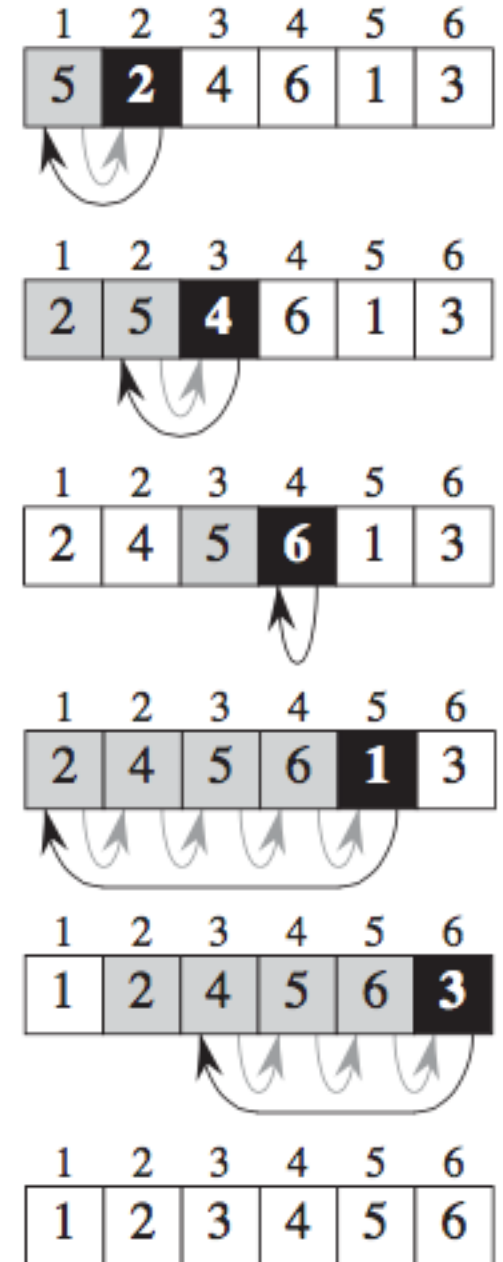
// Insert "key" into correct  
// position to its left.

# Insertion Sort

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 5 | 2 | 4 | 6 | 1 | 3 |
|---|---|---|---|---|---|

1. **FUNCTION** InsertionSort(array)
2.   **FOR** j **IN** [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     **WHILE** i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9. **RETURN** array

// Insert "key" into correct  
// position to its left.



# Argue that it correctly sorts

Proof of correctness

# Insertion Sort Correctness Theorem

**Theorem:** a proposition that can be proved by a chain of reasoning

*For every input array of length  $n \geq 1$ , the Insertion Sort algorithm reorders the array into nondecreasing order.*

# Insertion Sort – Proof of correctness

## Lemma (**loop invariant**)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.

What is a lemma?

*an intermediate theorem in a proof*

What is a theorem?

*a general proposition not self-evident but proved by a chain of reasoning; a truth established by means of accepted truths.*

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

# Insertion Sort – Proof of correctness

## Lemma (**loop invariant**)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.

## General conditions for **loop invariants**

1. **Initialization**: The loop invariant is satisfied at the beginning of the loop before the first iteration.
2. **Maintenance**: If the loop invariant is true before the  $i$ th iteration, then the loop invariant will be true before the  $i+1$  iteration.
3. **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.


```
1. FUNCTION InsertionSort(array)
2.   FOR  $j$  IN  $[1 \dots \text{array.length}]$ 
3.      $\text{key} = \text{array}[j]$ 
4.      $i = j - 1$ 
5.     WHILE  $i \geq 0 \ \&\& \ \text{array}[i] > \text{key}$ 
6.        $\text{array}[i + 1] = \text{array}[i]$ 
7.        $i = i - 1$ 
8.      $\text{array}[i + 1] = \text{key}$ 
9.   RETURN array
```

# Insertion Sort – Proof of correctness

1. Initialization: The loop invariant is satisfied at the beginning of the loop before the first iteration..

Lemma (loop invariant)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.



```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

# For to While Loop

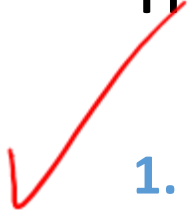
```
FOR j IN [1 ..< array.length]  
    ...
```

1. Initialization: The loop invariant is satisfied at the beginning of the loop before the first iteration..

```
j = 1  
WHILE j < array.length  
    ...  
    j = j + 1
```



# Insertion Sort – Proof of correctness

- 
1. **Initialization:** The loop invariant is satisfied at the beginning of the loop before the first iteration..

Lemma (loop invariant)

- At the start of the iteration with index  $j$ , the subarray `array[0 ..= j-1]` consists of the elements originally in `array[0 ..= j-1]`, but in non-decreasing order.

- When  $j = 1$ , the subarray is `array[0 ..= 1-1]`, which includes only the first element of the `array`. The single element subarray is sorted.

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

$j = 56$  array  $[0 \dots 55]$  are sorted

# Insertion Sort – Proof of correctness

array  $[0 \dots j-1] \rightarrow$  array  $[0 \dots j-1+1]$   
↑

2. **Maintenance:** If the loop invariant is true before the  $i$ th iteration, then the loop invariant will be true before the  $i+1$  iteration.

Lemma (loop invariant)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.

- Assume  $\text{array}[0 \dots j-1]$  is sorted. Informally, the loop operates by moving elements to the right until it finds the position of key. Next,  $j$  is incremented.

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

What is  $j$ ?  $\rightarrow [n], n-1$

# Insertion Sort – Proof of correctness

**3. Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Lemma (loop invariant)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.

- The loop terminates when  $j = n$ . Given the initialization and maintenance results, we have shown that:  $\text{array}[0 \dots j-1] \rightarrow \text{array}[0 \dots n-1]$  in non-decreasing order.

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```


Entire array

# For to While Loop

```
FOR j IN [1 ..< array.length]  
    ...
```

1. Initialization: The loop invariant is satisfied at the beginning of the loop before the first iteration..

```
j = 1  
WHILE j < array.length  
    ...  
    j = j + 1
```



# Insertion Sort – Proof of correctness

3. **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Lemma (loop invariant)

- At the start of the iteration with index  $j$ , the subarray  $\text{array}[0 \dots j-1]$  consists of the elements originally in  $\text{array}[0 \dots j-1]$ , but in non-decreasing order.

```
1. FUNCTION InsertionSort(array)
2.   FOR  $j$  IN  $[1 \dots \text{array.length}]$ 
3.      $\text{key} = \text{array}[j]$ 
4.      $i = j - 1$ 
5.     WHILE  $i \geq 0 \ \&\& \ \text{array}[i] > \text{key}$ 
6.        $\text{array}[i + 1] = \text{array}[i]$ 
7.        $i = i - 1$ 
8.      $\text{array}[i + 1] = \text{key}$ 
9.   RETURN array
```

# Analyze its running time

Proof of running time

# Insertion Sort – Running time

Analyze using the **RAM** (random access machine) model

- Instructions are executed one after another (no parallelism)
- Each instruction takes a constant amount of time
  - Arithmetic (+, -, \*, /, %, floor, ceiling)
  - Data movement (load, store, copy)
  - Control (branching, subroutine calls)
- **Ignores memory hierarchy!** [\(never forget: linked lists are awful\)](#)
- This is a very simplified way of looking at algorithms
- Compare algorithms while ignoring hardware

# Insertion Sort Running Time Theorem

**Theorem:** a proposition that can be proved by a chain of reasoning

*For every input array of length  $n \geq 1$ , the Insertion Sort algorithm performs at most  $5n^2$  operations.*




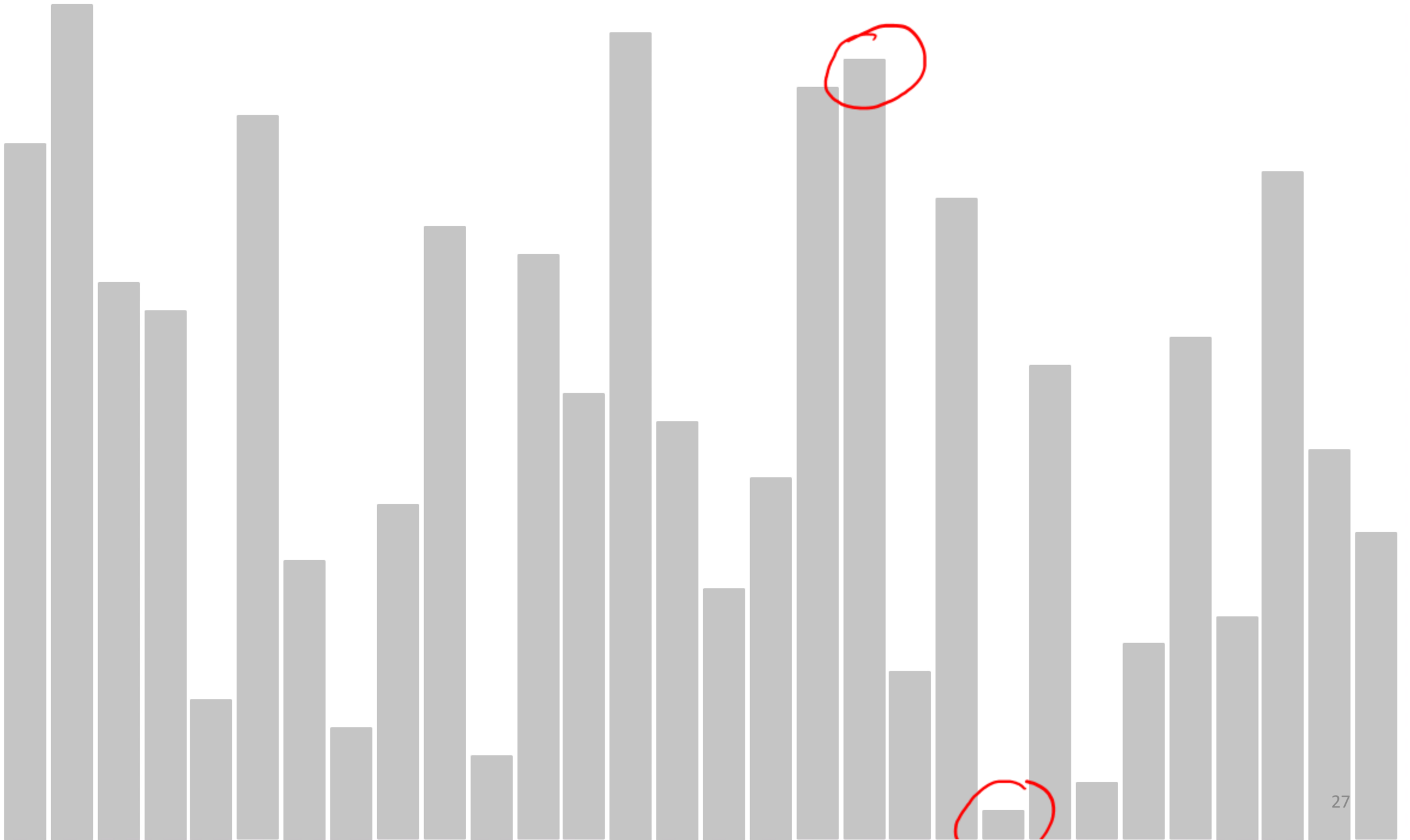
# Insertion Sort – Running time

On what does the running time depend?

- Number of items to sort
  - 3 numbers vs 1000

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```





# Insertion Sort – Running time

On what does the running time depend?

- Number of items to sort
  - 3 numbers vs 1000
- How much are they already sorted
  - The hint here is that the inner loop is a **while** loop (not a for loop)

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

```
1. FUNCTION InsertionSort(array)
2.   FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     array[i + 1] = key
9.   RETURN array
```

### Cost

```
1.   0
2.   ?
```

for ( int j = 1 ; j < array.len ; j++ )

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.  j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> |
|-----|-------------|
| 1.  | 0           |
| 2.  | 1           |
| 3.  | 2           |
| 4.  | 2           |
| 5.  | 2           |
| 6.  | 4           |
| 7.  | 4           |
| 8.  | 2           |
| 9.  | 3           |
| 10. | 2           |
| 11. | 1           |

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | length            |
| 4.  | 2           |                   |
| 5.  | 2           |                   |
| 6.  | 4           |                   |
| 7.  | 4           |                   |
| 8.  | 2           |                   |
| 9.  | 3           |                   |
| 10. | 2           |                   |
| 11. | 1           |                   |

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | ?                 |
| 7.  | 4           |                   |
| 8.  | 2           |                   |
| 9.  | 3           |                   |
| 10. | 2           |                   |
| 11. | 1           |                   |

Loop code always executes one fewer time than the condition check.

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | depends           |
| 7.  | 4           |                   |
| 8.  | 2           |                   |
| 9.  | 3           |                   |
| 10. | 2           |                   |
| 11. | 1           |                   |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is



```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | (n - 1)x          |
| 7.  | 4           | (n - 1)(x - 1)    |
| 8.  | 2           | (n - 1)(x - 1)    |
| 9.  | 3           | n - 1             |
| 10. | 2           | n - 1             |
| 11. | 1           | 1                 |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | + 1         | 1                 |
| 3.  | + 2         | n                 |
| 4.  | + 2         | n - 1             |
| 5.  | + 2         | n - 1             |
| 6.  | + 4         | (n - 1)x          |
| 7.  | 4           | (n - 1)(x - 1)    |
| 8.  | 2           | (n - 1)(x - 1)    |
| 9.  | 3           | n - 1             |
| 10. | 2           | n - 1             |
| 11. | 1           | 1                 |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is

What is the **total running time** (add up all operations)?

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | (n - 1)x          |
| 7.  | 4           | (n - 1)(x - 1)    |
| 8.  | 2           | (n - 1)(x - 1)    |
| 9.  | 3           | n - 1             |
| 10. | 2           | n - 1             |
| 11. | 1           | 1                 |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is

What is the **total running time** (add up all operations)?

$$\begin{aligned}
 \text{Total Running Time} &= 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1 \\
 &= 10nx + 5n - 10x - 1
 \end{aligned}$$

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | (n - 1)x          |
| 7.  | 4           | (n - 1)(x - 1)    |
| 8.  | 2           | (n - 1)(x - 1)    |
| 9.  | 3           | n - 1             |
| 10. | 2           | n - 1             |
| 11. | 1           | 1                 |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is

What is the **best-case** scenario?

**array** is already sorted

x = ?

$$\begin{aligned}
 \text{Total Running Time} &= 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1 \\
 &= 10nx + 5n - 10x - 1 \quad x = 1 \\
 &= 10n + 5n - 10 - 1 \\
 &= 15n - 11
 \end{aligned}$$

Is "- 11" a problem? Negative time?

```

1. FUNCTION InsertionSort(array)
2.   j = 1
3.   WHILE j < array.length
4.     key = array[j]
5.     i = j - 1
6.     WHILE i ≥ 0 && array[i] > key
7.       array[i + 1] = array[i]
8.       i = i - 1
9.     array[i + 1] = key
10.    j = j + 1
11.  RETURN array

```

|     | <u>Cost</u> | <u>Executions</u> |
|-----|-------------|-------------------|
| 1.  | 0           | 0                 |
| 2.  | 1           | 1                 |
| 3.  | 2           | n                 |
| 4.  | 2           | n - 1             |
| 5.  | 2           | n - 1             |
| 6.  | 4           | (n - 1)x          |
| 7.  | 4           | (n - 1)(x - 1)    |
| 8.  | 2           | (n - 1)(x - 1)    |
| 9.  | 3           | n - 1             |
| 10. | 2           | n - 1             |
| 11. | 1           | 1                 |

Loop code always executes one fewer time than the condition check.

Depends on how sorted **array** is

What is the **worst-case** scenario?

**array** is reverse sorted

x = ?

$$\begin{aligned}
 \text{Total Running Time} &= 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1 \\
 &= 10nx + 5n - 10x - 1 \quad \text{x = n/2 on average} \\
 &= 5n^2 + 5n - 5n - 1 \\
 &= 5n^2 - 1
 \end{aligned}$$

Expand  
Explicitly

# Best, Worst, and Average

We usually concentrate on worst-case

- Gives an upper bound on the running time for any input
- The worst case can occur fairly often
- The average case is often relatively as bad as the worst case

# Summary

- Introductions

- (Difficult) Exercise

- Specify an algorithm

- Prove correctness

- Analyze total running time