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

- **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)?

Heap, Hash Table (dict, map, hash map), LL ☹, Tree, set, array ☺, enum

- What algorithms do you know?

Dijkstra, Sorting, BS

- What programming languages do you know?

Java, Python, OCaml, Haskell, C, Rust,
Scala, PhP, R, JS

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

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

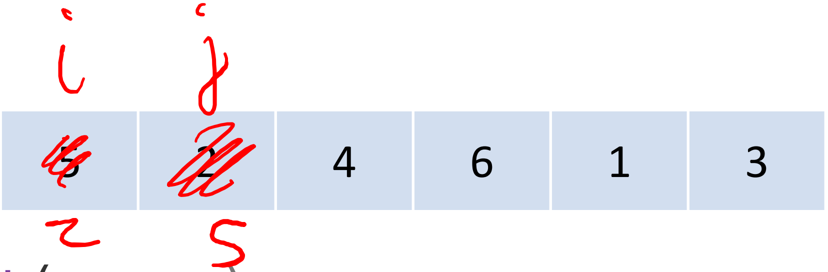
- **Input**: an array of n items, in arbitrary order
- **Output**: a reordering of the input into nondecreasing order
- **Assumptions**: none

We will

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

Specify the algorithm

Insertion Sort



```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j] = 2
        i = j - 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i - 1
        array[i + 1] = key
    RETURN array
```

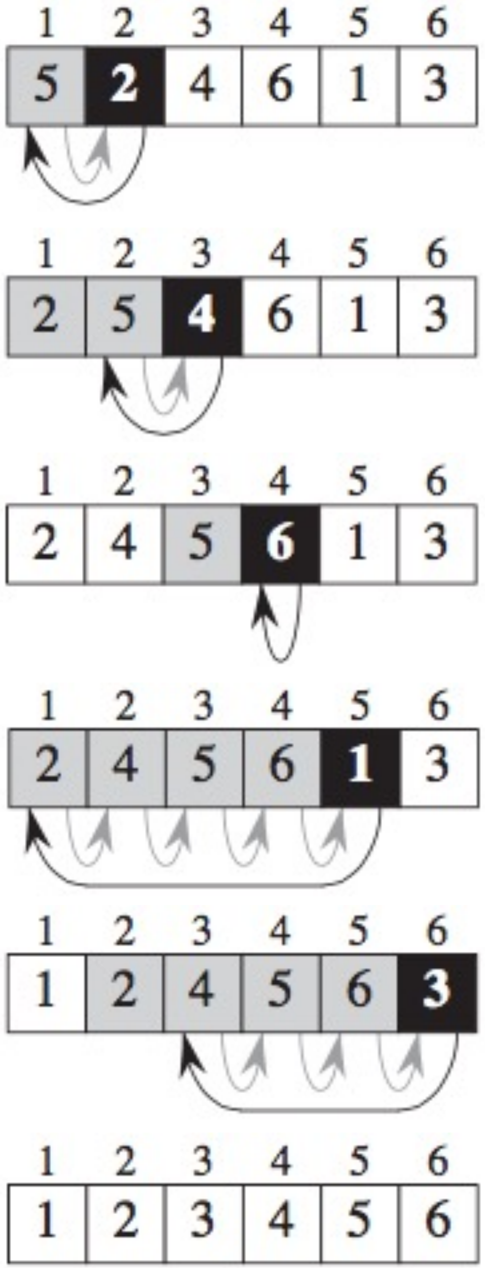
// Insert "key" into correct
// position to its left.

Insertion Sort

5	2	4	6	1	3

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j - 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i - 1
        array[i + 1] = key
    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 ≥ 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 array[0 ..= j-1] consists of the elements originally in array[0 ..= j-1], but in non-decreasing order.

What is a lemma?

an intermediate theorem in a proof

```
FUNCTION InsertionSort(array)
   FOR j IN [1 ..< array.length]
      key = array[j]
      i = j − 1
      WHILE i ≥ 0 && array[i] > key
         array[i + 1] = array[i]
         i = i − 1
      array[i + 1] = key
   RETURN array
```

Insertion Sort – Proof of correctness

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.

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    RETURN array
```

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 ith 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.

Insertion Sort – Proof of correctness

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

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j - 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i - 1
        array[i + 1] = key
    RETURN array
```

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.

For to While Loop

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

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

```
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.

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    RETURN array
```

Insertion Sort – Proof of correctness

2. **Maintenance**: <u>If the loop invariant is true before the ith iteration, then the loop invariant will be true before the i+1 iteration.</u>

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.

- Assume array[0 ..= 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.

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    RETURN array
```

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 array[0 ..= j-1] consists of the elements originally in array[0 ..= j-1], but in non-decreasing order.

- The loop terminates when j = n. Given the initialization and maintenance results, we have shown that: array[0 ..= j-1] → array[0 ..= n-1] in non-decreasing order.

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    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.

For every input array of length $n \geq 1$, the Insertion Sort algorithm performs at most $O(n^2)$ operations. ✓

For every input array of length $n \geq 1$, the Insertion Sort algorithm performs on average $O(n^2)$ operations. ✓

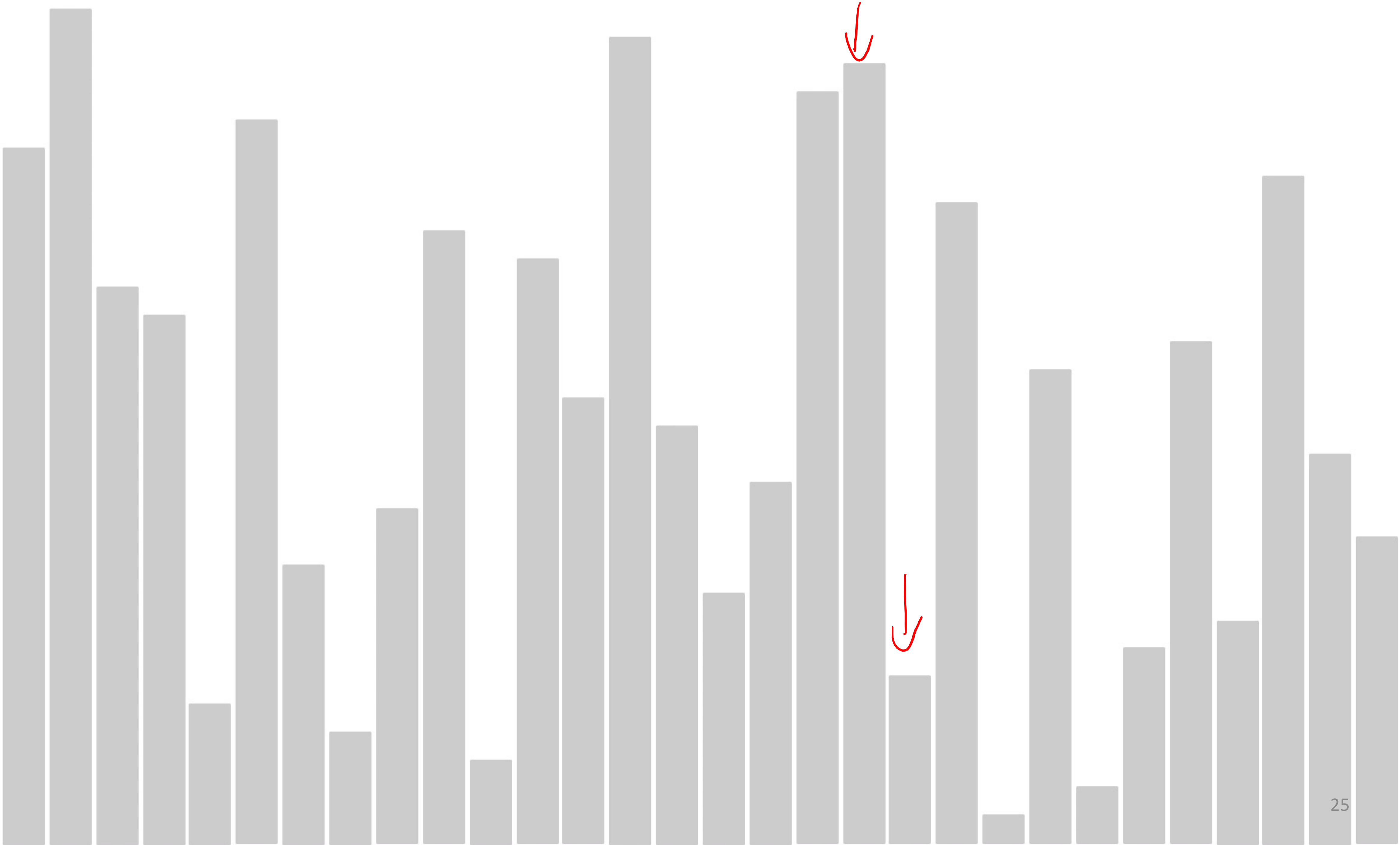
For every input array of length $n \geq 1$, the Insertion Sort algorithm performs at least $O(n^2)$ operations. ✓

Insertion Sort – Running time

On what does the running time depend?

- Number of items to sort
    - 3 numbers vs 1000

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    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)

```
FUNCTION InsertionSort(array)
    FOR j IN [1 ..< array.length]
        key = array[j]
        i = j − 1
        WHILE i ≥ 0 && array[i] > key
            array[i + 1] = array[i]
            i = i − 1
        array[i + 1] = key
    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. ?

```
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
```

	Cost
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
```

	Cost	**Executions**
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
```

	Cost	**Executions**
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
```

	Cost	**Executions**
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
```

	Cost	**Executions**
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
```

	Cost	Executions
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
```

	Cost	**Executions**
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)?

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

```
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
```

	Cost	**Executions**
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 = ?

Total Running Time $= 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1$

$= 10nx + 5n - 10x - 1$ x = 1

$= 10n + 5n - 10 - 1$

$= 15n \boxed{- 11}$

$T_s(n) = O(n)$

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
```

	Cost	**Executions**
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 = ?

Total Running Time $= 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1$

$= 10nx + 5n - 10x - 1$

x = n/2 on average

$= 5n^2 + 5n - 5n - 1$

$= 5\boxed{n^2} - 1$

$T_{RS}(n) = O(n^2)$

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