Insertion Sort

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

Outline

Topics and Learning Objectives

- Warm-Up and introduction
- Specify an algorithm
- Prove correctness
- Analyze total running time

Exercise

- Friend Circles

Extra Resources

- **Key resources for all lectures in this course:**
  - Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
  - Algorithms Illuminated by Tim Roughgarden

- For this lecture
  - **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

What can you do if your code is too slow?

- Use a better algorithm.
- Use a better data-structure.
- Use a lower-level system.
- Accept a less precise solution.

Why Study Algorithms?

They are…

- important for all branches of computer science (networking, cryptography, graphics, robotics, databases, machine learning, etc.)
- drive innovation (think search engines and marketing),
- a lens into other sciences (economics, control theory), and
- fun! (alternatively: good for interviews)

How is mathematical logic used to describe the behavior of functions or programs? What is a "theorem?" How do you "prove" theorems? What is an "automatic theorem prover?" What do all these mathematical terms—"induction," "generalization," "lemma"—have to do with programs being "correct?" How can "programmer's intuition" guide a foray into mathematics? How do you think about programs that call themselves without just going in circles?

--- J Strother Moore, The Little Prover

How is mathematical logic used to describe the behavior of functions or programs? What is a "theorem?" How do you "prove" theorems? What is an "automatic theorem prover?" What do all these mathematical terms—"induction," "generalization," "lemma"—have to do with programs being "correct?" How can "programmer's intuition" guide a foray into mathematics? How do you think about programs that call themselves without just going in circles?

--- J Strother Moore, The Little Prover

Sometimes, the best way to learn how to do something is just to sit down and try to do it.

There are two problems with that advice, and the problems are especially acute when mathematical logic is involved. First, you have to understand the "rules of the game." Those rules—if followed exactly—will ensure that what you "prove" is really true. Second, it is hard to keep in mind the precise statement of every rule—and if you make a mistake you might end up believing something is true when it is not.

Proofs are a lot like jigsaw puzzles. There are no rules about how jigsaw puzzles must be solved. The only rule concerns the final product: All the pieces must fit together, and the picture must look right. The same holds for proofs.

--- Daniel J. Velleman, How To Prove It

Proofs are a lot like jigsaw puzzles. There are no rules about how jigsaw puzzles must be solved. The only rule concerns the final product: All the pieces must fit together, and the picture must look right. The same holds for proofs.

--- Daniel J. Velleman, How To Prove It

Although there are no rules about how jigsaw puzzles must be solved, some techniques for solving them work better than others.

Structure and Language

- I'll try to use consistent language (theorems, problem statements, algorithms, etc.)
- I'll explicitly state the inputs, outputs, and assumptions when they are useful (when they make the proof easier to understand)
- I'll mostly use my own flavor or pseudocode (and it will change from time to time depending on what is useful, i.e., base-0 or base-1)

What do we typically analyze?

We can analyze and prove many things

- Running time
- Correctness
- Memory usage

Sometimes we analyze a

1. general problem (like sorting)
2. and sometimes a specific algorithm (like quicksort)

Some Useful Terms

- Proof: an argument which convinces other people that something is true
- Theorems: technical statements that are always true (or invalid)
- Lemmas: statements that assist in the proof of theorems
- Corollary: statements that follow immediately from an already-proved result
- Proposition: stand-alone technical statement not particularly important alone
- Axiom (or postulate): a statement taken on faith

Guiding Principles for Analysis

Focus on

- Worst-case analysis (not average or best case)
- Big-picture analysis (it’s OK to be a little loose with constant factors)
- Asymptotic analysis (bigger inputs are more interesting)

The High-Level Process

- State the theorem (including inputs, outputs, and assumptions)
- Introduce known axioms and useful definitions
- Follow a sequence of logical steps that sometimes includes new lemmas as a means of making the proof easier to follow and summarize
- Conclude and summarize (sometimes marked with QED or ■)

Warm-Up

Problem, **P(n)**: sorting an array of items using InsertionSort

- **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

Problem, **P(n)**: sorting an array of items using InsertionSort

- **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

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
```

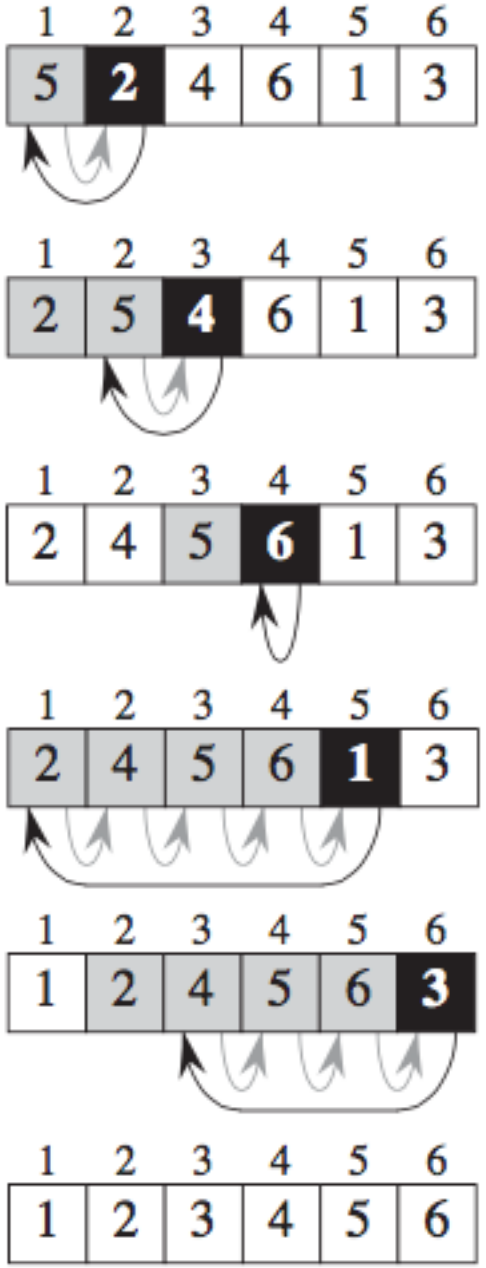
```
// Insert “key” into correct
// position to its left.
```

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 ≥ 1,

InsertionSort *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

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 array[0 ..= j-1] consists of the elements originally in array[0 ..= 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 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.

```
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

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.

```
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

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.

```
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

2. **Maintenance**: If the loop invariant is true before the ith 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 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.

```
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

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.

```
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

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

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

```
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 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.

```
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 Correctness Theorem

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

For every input array of length n ≥ 1,

InsertionSort *reorders the array into nondecreasing order.*

Given our result from the loop invariant lemma, we have shown that InsertionSort reorders the array into nondecreasing order.

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

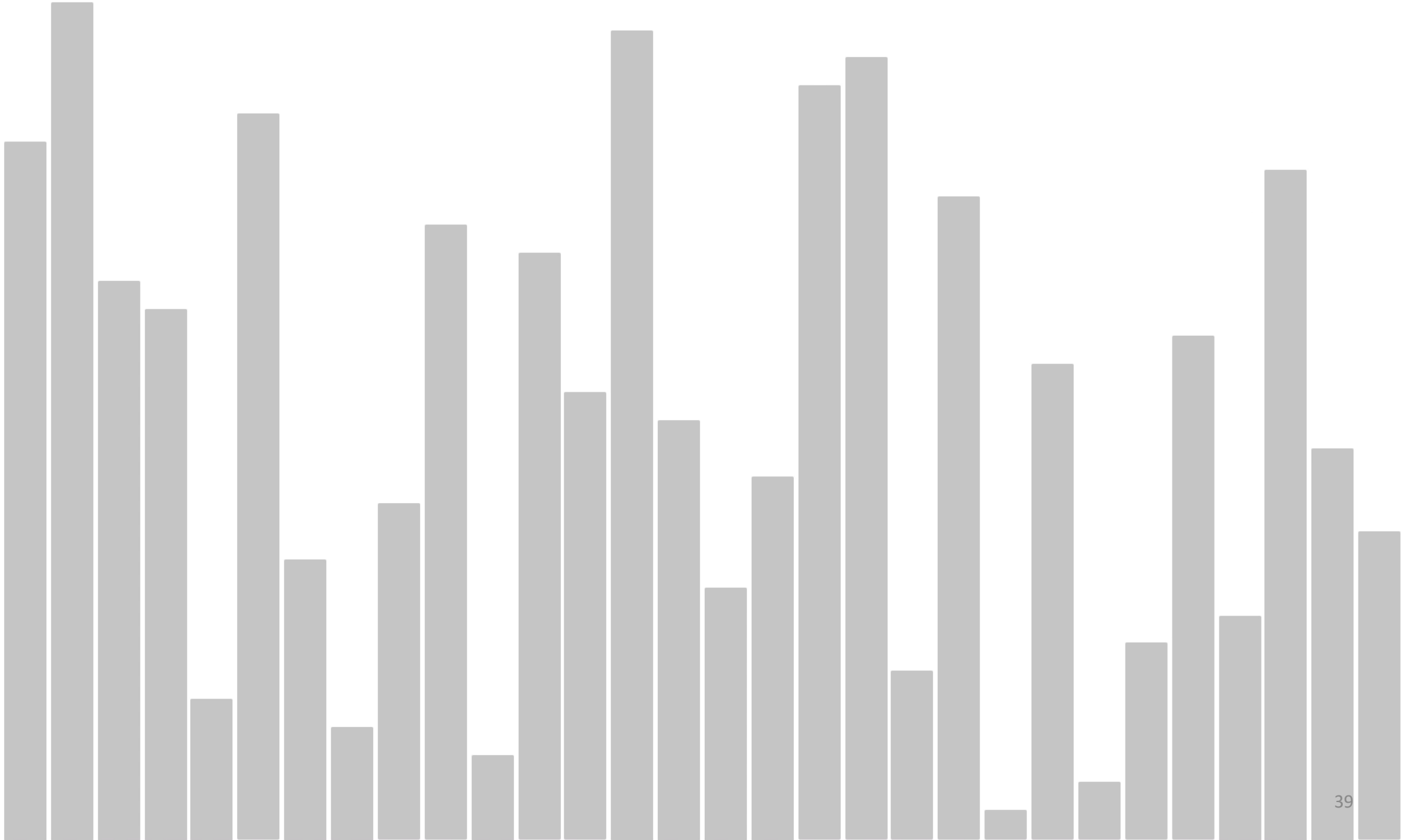
For every input array of length $n \geq 1$,
InsertionSort *performs at most* $5n^2$ *operations.*

Insertion Sort – Running time

What affects the running time of InsertionSort?

- 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

What affects the running time of InsertionSort?

- 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
```

Let’s count the number of operations.

```
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)?

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

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

$= 15n - 11$

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

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

$= 5n^2 - 1$

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