CSCI 241

Lecture 8

Runtime Analysis Revisited

Announcements

- No lab this week (work on A1)
- Quiz on Friday (as always)

Goals:

- Know how to determine the big-O runtime (aka asymptotic runtime class) of an algorithm given the number of operations it performs.
- Understand the basics of counting operations in recursive algorithms.
- Know the runtime complexity of the sorting algorithms we've covered.

Runtime Analysis: Overview

Why? We want a measure of performance where

- it is **independent** of what computer we run it on.

  Solution: count **operations** instead of clock time.

- Dependence on **problem size** is made explicit.

  Solution: express runtime as a function of **n**
  (or whatever variables define problem size)

- it is **simpler** than a raw count of operations and focuses on performance on **large problem sizes**.

  Solution: ignore constants, analyze **asymptotic** runtime.

Runtime Analysis: Overview

How?

1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.

   e.g., `sillyFindMax`: $\cancel{2} + \cancel{5N} + \cancel{6}N^2$

2. Drop constants and lower-order terms to find the **asymptotic runtime class.**

   $O(n^2)$

Runtime Analysis: Overview

How?

1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.

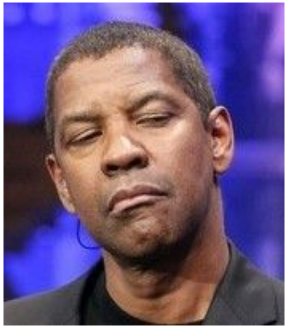
   e.g., `sillyFindMax`: $2 + 5N + 6N^2$

2. Drop constants and lower-order terms to find the **asymptotic runtime class.**

Runtime Analysis: Overview

How?

1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.



   e.g., `sillyFindMax`: $2 + 5N + 6N^2$

2. Drop constants and lower-order terms to find the **asymptotic runtime class.**

Really? *any* constant?

A practical argument:

Really? *any* constant?

A practical argument:

- My MacBook Pro from 2013: 3.17 **giga**FLOPS

floatingpoint operations per second

Really? *any* constant?

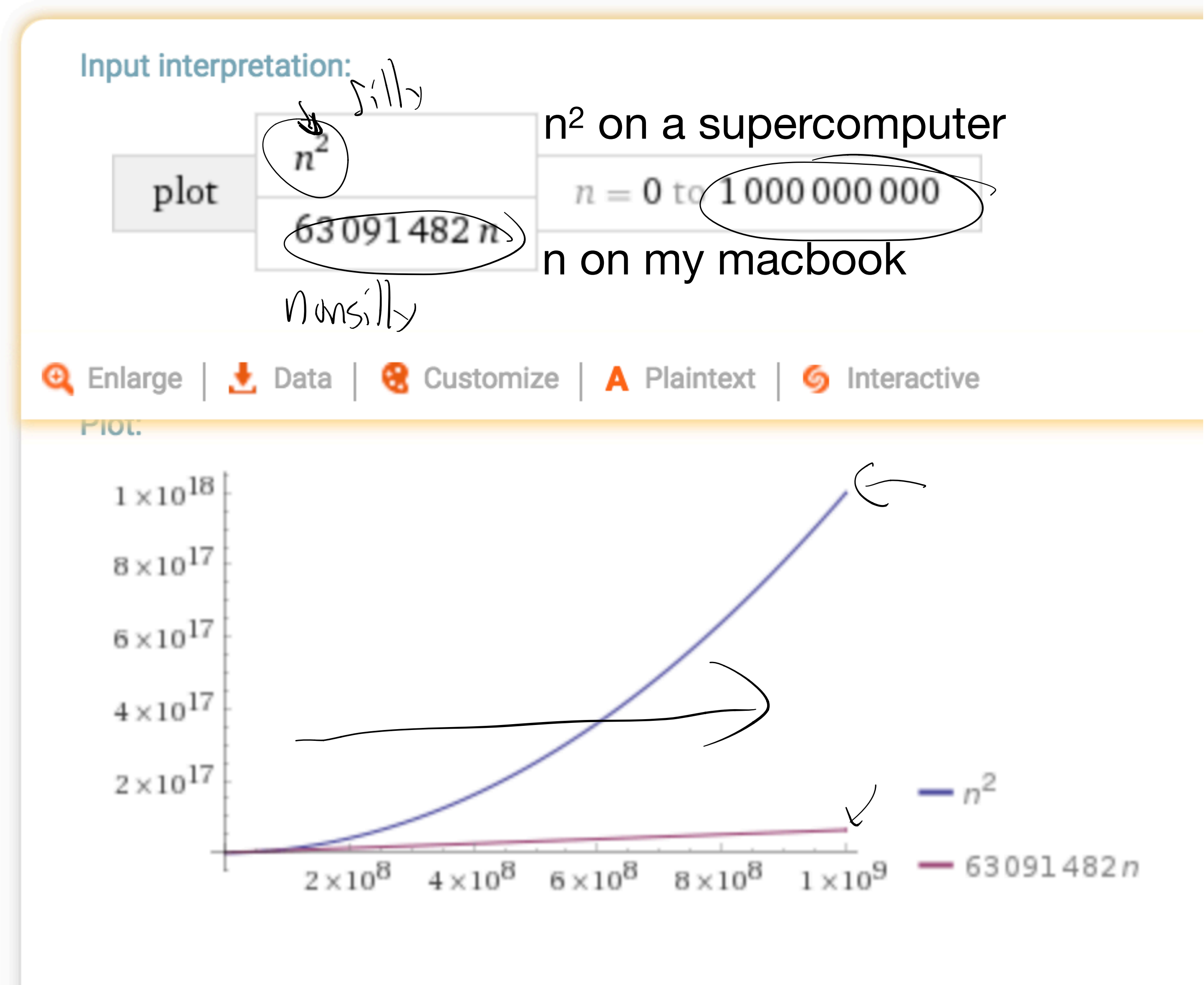
A practical argument:

• My MacBook Pro from 2013:	3.17 **giga**FLOPS
• Fastest supercomputer as of Nov. 2019:	200 **peta**FLOPS

Really? *any* constant?

A practical argument:

- My MacBook Pro from 2013: 3.17 **giga**FLOPS
- Fastest supercomputer as of Nov. 2019: 200 **peta**FLOPS
- Supercomputer is 63,091,482 times faster.



Input interpretation:

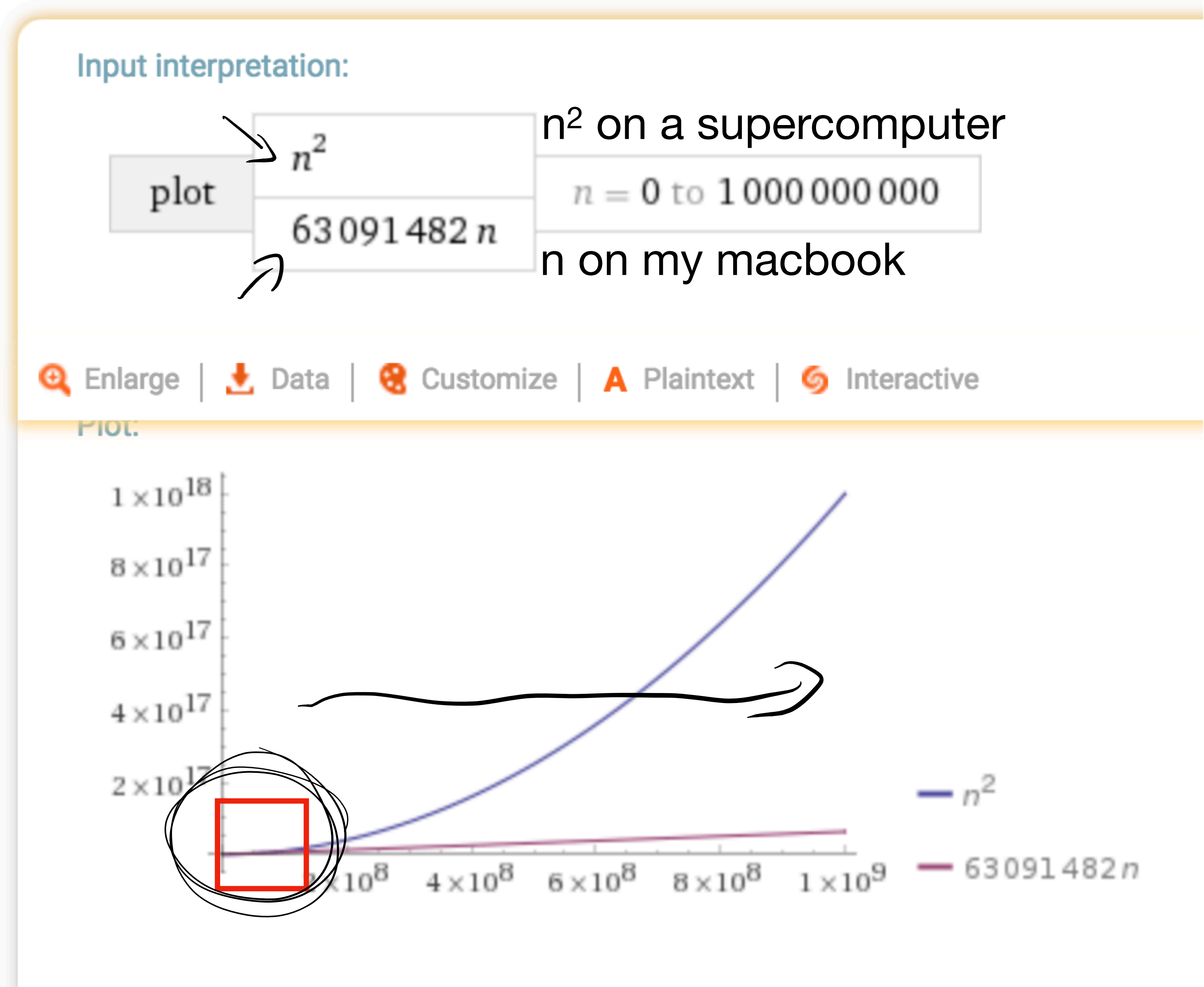
plot n^2 / $63091482\,n$ $n = 0$ to 1000000000

silly

nonsilly

n^2 on a supercomputer

n on my macbook

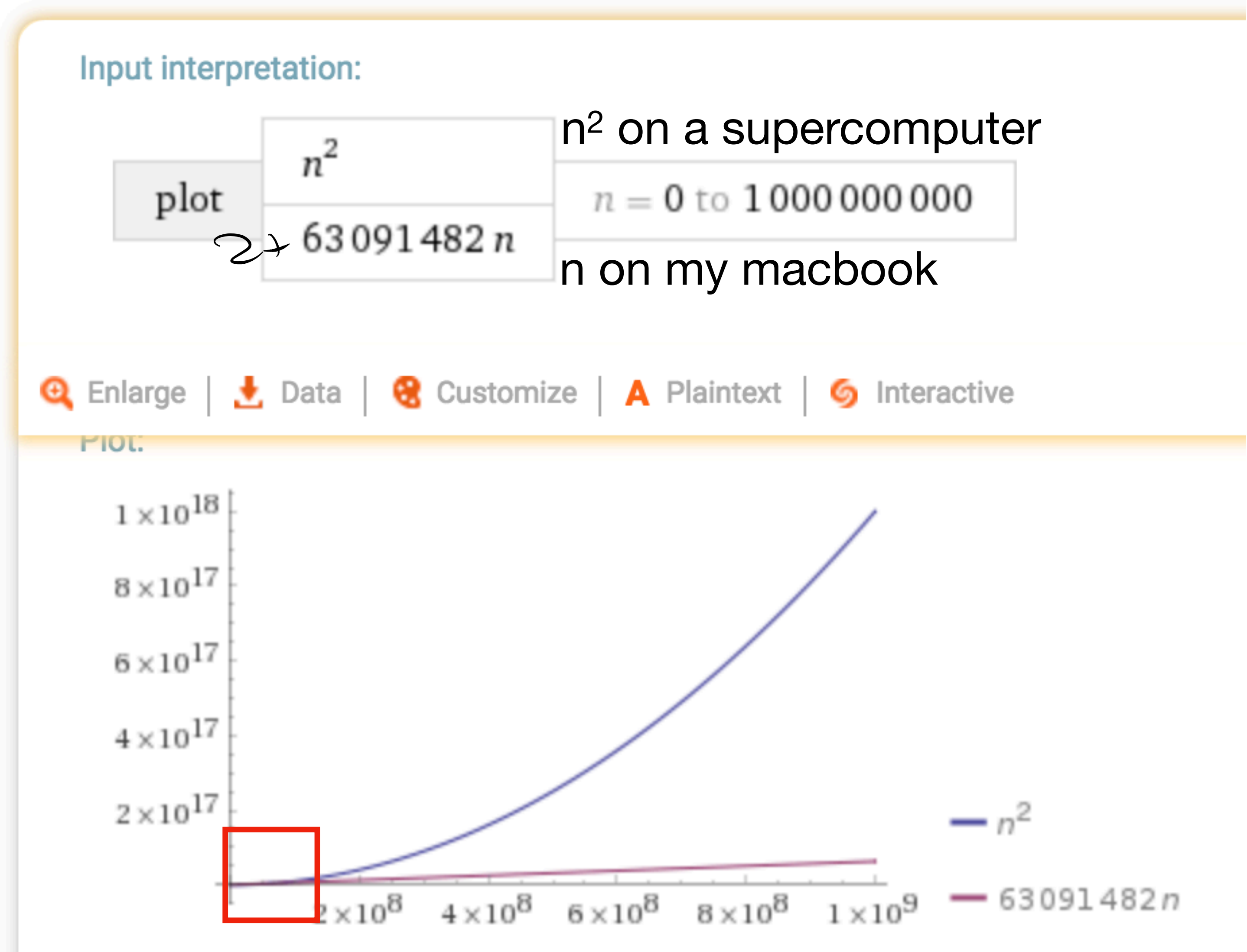


Input interpretation:

plot	n^2	$n = 0$ to 1000000000
	$63091482\,n$	

n^2 on a supercomputer

n on my macbook



Input interpretation:

n² on a supercomputer

plot	n^2	$n = 0$ to 1000000000
	2x $63091482\,n$	

n on my macbook

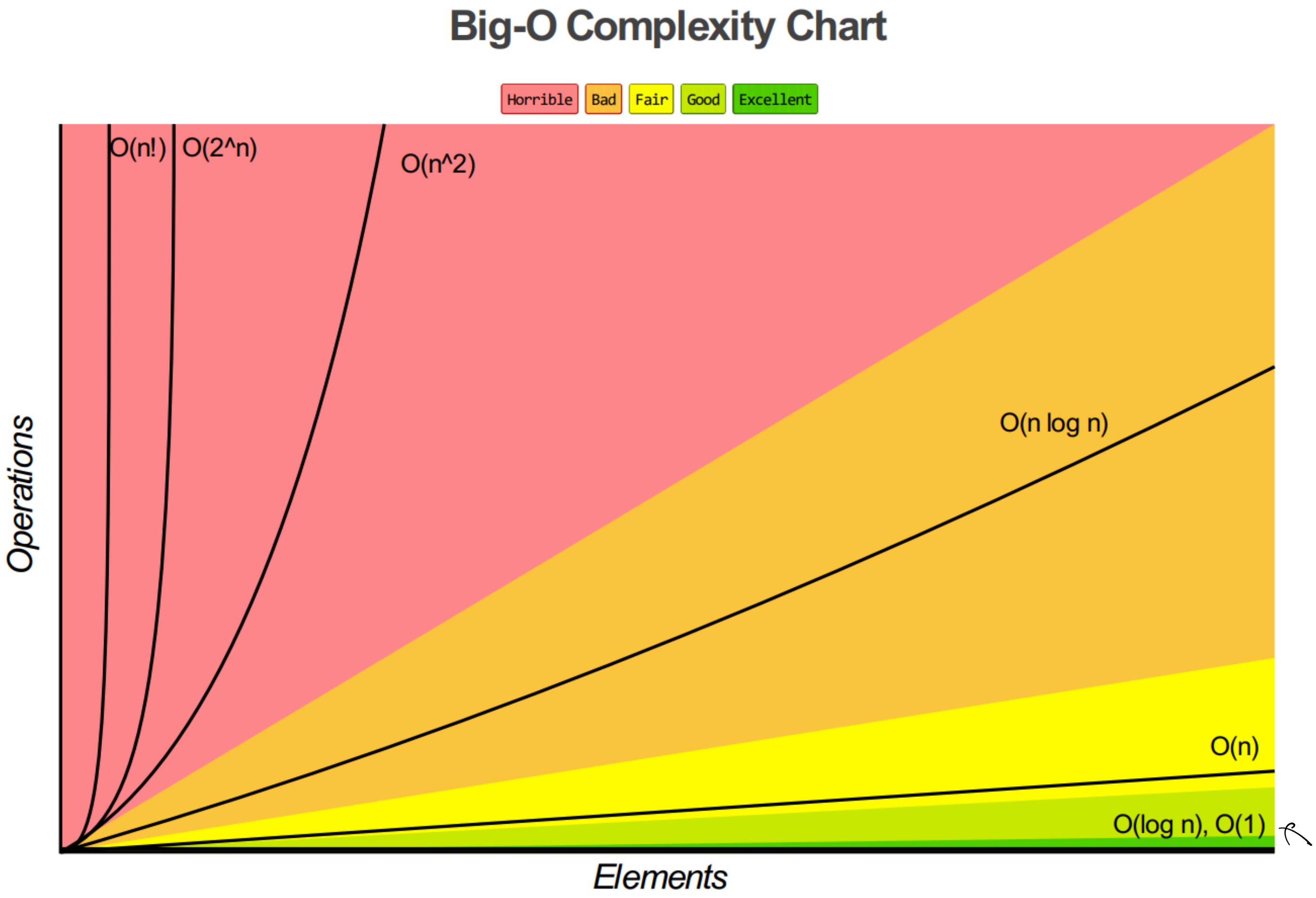
n^2 algorithm may be faster here!

Asymptotic Runtime Class

or, "big-O" runtime

- Tells us how the runtime **grows** as the input size grows.
- Doesn't tell us *anything* about runtime when the input is small!

Common Complexities



Counting Operations

What’s a constant-time operation? $O(1)$

- Anything that **doesn’t** depend on the input size:
    - Reading/writing from/to a variable or array location.
    - Evaluating an arithmetic or boolean expression.
    - Returning from a method.
- a constant # of any of the above

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
  - Reading/writing from/to a variable or array location.
  - Evaluating an arithmetic or boolean expression.
  - Returning from a method.

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
    - Reading/writing from/to a variable or array location.

      ```
      int i = 2; int b = 4; a[i] = b;
      ```
    - Evaluating an arithmetic or boolean expression.
    - Returning from a method.

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
  - Reading/writing from/to a variable or array location.

    ```
    int i = 2; int b = 4; a[i] = b;
    ```
  - Evaluating an arithmetic or boolean expression.

    ```
    int i = 0; int j = i+4; int k = i*j;
    ```
  - Returning from a method.

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
  - Reading/writing from/to a variable or array location.

    ```
    int i = 2; int b = 4; a[i] = b;
    ```
  - Evaluating an arithmetic or boolean expression.

    ```
    int i = 0; int j = i+4; int k = i*j;
    ```
  - Returning from a method. `return k;`

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
    - Reading/writing from/to a variable or array location.

      ```
      int i = 2; int b = 4; a[i] = b;
      ```

    - Evaluating an arithmetic or boolean expression.

      ```
      int i = 0; int j = i+4; int k = i*j;
      ```

    - Returning from a method.

      ```
      return k;
      ```

Key intuition:

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
  - Reading/writing from/to a variable or array location.

    ```
    int i = 2; int b = 4; a[i] = b;
    ```

  - Evaluating an arithmetic or boolean expression.

    ```
    int i = 0; int j = i+4; int k = i*j;
    ```

  - Returning from a method.

    ```
    return k;
    ```

Key intuition:

- These don’t take identical amounts of time, but the times are within a **constant factor** of each other.

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
    - Reading/writing from/to a variable or array location.

      ```
      int i = 2; int b = 4; a[i] = b;
      ```
    - Evaluating an arithmetic or boolean expression.

      ```
      int i = 0; int j = i+4; int k = i*j;
      ```
    - Returning from a method. `return k;`

Key intuition:

- These don’t take identical amounts of time, but the times are within a **constant factor** of each other.
- Same for running the **same** operation on a **different** computer.

Counting Operations

What’s **not** a constant-time operation?

- Anything that **does** depend on the input size, e.g.:
  - Looping over all values in an array of size n.
  - Recursively checking whether a string is a palindrome
  - Sorting an array
  - Most nontrivial algorithms / data structure operations we’ll cover in this class.

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- We have to specify whether we want worst-case, average-case (aka expected-case), or best-case runtime.

```
public int findMax(int[] a) {
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++) {
    if (currentMax < a[i]) {
      currentMax = a[i];
    }
  }
}
```

~~false~~ always true

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- We have to specify whether we want worst-case, average-case (aka expected-case), or best-case runtime.

```
public int findMax(int[] a) {
   int currentMax = a[0];
   for (int i = 1; i < a.length; i++) {
      if (currentMax < a[i]) {
         currentMax = a[i];
      }
   }
}
```

# times executed depends on contents of a!

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- Worst-case is usually the important one, with notable exceptions for algorithms that beat asymptotically faster algorithms in practice.
- Quicksort is worst-case O(n^2) but often beats MergeSort in practice

Counting Strategies Review:
1. Simple counting

```
/** A singly linked list node */
public class Node {
  int value;
  Node next;
  public Node(int v) {
    value = v;
  }
}

/** Insert val into the list in after pred.
 * Precondition: pred is not null */
public void addAfter(Node pred, int val) {
  Node newNode = new Node(val);      1
  new_node.next = pred.next;         1
  pred.next = newNode;               1
}
```

$O(1)+O(1)+O(1)$ is $O(1)$

Counting Strategies Review:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:

int i = 0;
while (i < n) {
    loopBody(i);
    i++;
}
```

$1 + loopBody \cdot n + n$

$(loopBody) \cdot n$

$n \cdot loopBody$

$0..n$ $n-0$ iterations

`int i = 0;` — 1

`while (i < n) {` — 1 per iteration $n \cdot O(1)$

`loopBody(i);` — 1 per iteration $n \cdot$ runtime of loopBody

`i++;` — 1 per iteration $n \cdot O(1)$

Counting Strategies Review:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:

int i = 0;              1
while (i < n) {         1 per iteration
   loopBody(i);         1 per iteration
   i++;                 1 per iteration
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies Review: 1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:

int i = 0;                  1
while (i < n) {             n
    loopBody(i);            n * runtime of loopBody
    i++;                    n
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies Review: 1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:
```

Total runtime:
1 + 2n + n*[runtime of loopBody]

```
int i = 0;                1
while (i < n) {           n
   loopBody(i);           n * runtime of loopBody
   i++;                   n
}
```

How many iterations?
i takes on values 0..n, of which there are n.

Counting Strategies:
2. Aggregate Analysis

Not as easy case:

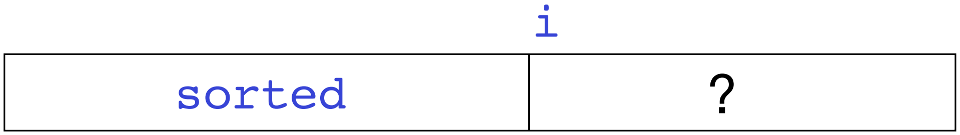
1. Identify all primitive operations

2. Trace through the algorithm, reasoning about the loop bounds in order to count the worst-case number of times each operation happens.

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Invariant: A

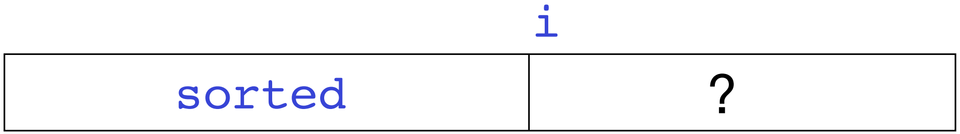


AT MOST How many times do we call swap() during iteration i?

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
   i = 0;
   while i < A.length:
      j = i;
      while j > 0 and A[j] < A[j-1]:
         swap(A[j], A[j-1])
         j--
      i++
```

Invariant: A



i

sorted

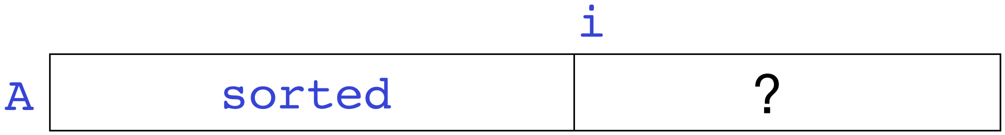
?

AT MOST How many times do we call swap() during iteration i?

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Invariant:



AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that’s as many as i swaps at iteration i

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
   i = 0;
   while i < A.length:
      j = i;
      while j > 0 and A[j] < A[j-1]:
         swap(A[j], A[j-1])
         j--
      i++
```

O(1)

n

n-1

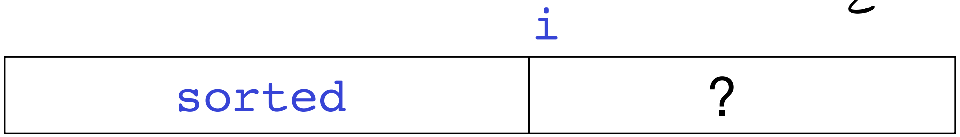
n-2

↓

$\frac{n(n-1)}{2} = \frac{n^2 - \cancel{n}}{\cancel{2}}$

O(n²)

Invariant: A



AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that's as many as i swaps at iteration i

Number of swaps: 1 in 1st iteration + 2 in 2nd iteration + … + n in nth iteration
1 + 2 + 3 + … + n-1 + n = (n * (n-1)) / 2 = (n^2 - n) / 2

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that’s as many as i swaps at iteration i

Number of swaps: 1 in 1st iteration + 2 in 2nd iteration + … + n in nth iteration
1 + 2 + 3 + … + n-1 + n = (n * (n-1)) / 2 = (n^2 - n) / 2

Counting Strategies:
2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that's as many as i swaps at iteration i

Number of swaps: 1 in 1st iteration + 2 in 2nd iteration + … + n in nth iteration
$1 + 2 + 3 + \ldots + n-1 + n = (n * (n-1)) / 2 = (n^2 - n) / 2$

$(n^2 - n)/2 \Rightarrow n^2 / 2 - n / 2 \Rightarrow n^2 - n \Rightarrow O(n^2)$

What about recursion?

Much like loops:

1. How much work is actually done per call?
   (not counting recursive calls)
2. How many calls are made?
   - This is simpler when the work per call is the same.
   - Sometimes the work per call depends on n.

Operation Counting in Recursive Methods: Example

```
/** Prints the linked list starting at head */
public static void printList(Node head) {
                                     n
                       ← O(1)
  if (head != null) {
    System.out.println(head) ← O(1)
    printList(head.next) ← recursive
                  n-1
}
```

1. O(1)

2. n calls

↳ O(n)

You try one

What's the big-O runtime of this?

socrative.com

Room: CSCI241

```
for i in N..0:
  if A[i] == 5:
   return i;
```

O(1)

$\underbrace{n..0}_{n}$

$7, 6, 5 \ldots 1, \cancel{0}$

A. O(1)

[✓ B. O(n)]

C. O(n-5)

D. O(n^2)

Another!

What's the big-O runtime of this?

```
public static void fn(int n):
    if n < 0:
        return n

    return fn(n-2);
```

A. O(1)

[✓ B. O(n)]

C. O(n log n)

D. $O(n^2)$

1. How much work per call? (not recursive)

$O(1)$

2. How many calls?

int div?

$\frac{n}{2} + 1$

fn(n)
↳ fn(n-2)
↳ fn(n-4)
↳ fn(n-6)

$\boxed{O(n)}$

10, 8, 6, 4, 2, 0 -2

0