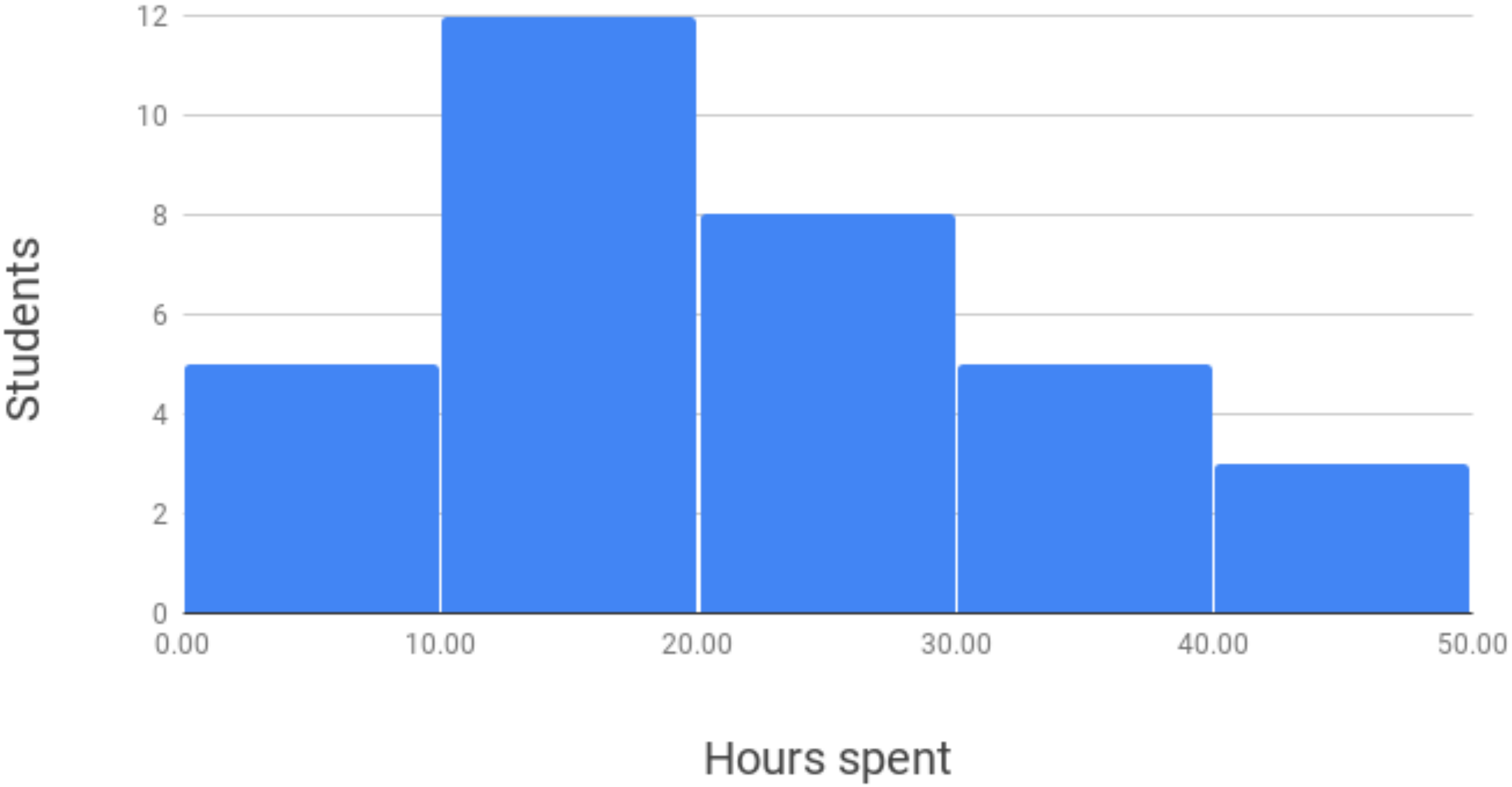
CSCI 241

Lecture 4:
Intro to Runtime Analysis
Recursion
Mergesort intuition

Announcements

- Lab 2 and A1 are out!
- Lab 2 is done in the same repo as A1
- A1 is due a week from Friday
- A1 is bigger than you think.

Hours spent on A1 last time I taught this class



Goals

- Know how to count primitive operations to determine the runtime of an algorithm.
- Understand how recursive methods are **executed**.
- Be able to **understand** and **develop** recursive methods *without* thinking about the details of how they are executed.
- Gain intuition for how merge sort works

“Primitive” Operations

Things the computer can do in a “fixed” amount of time.

“fixed” - doesn’t depend on the input size (n)

A non-exhaustive list:

- Get or set the value of a variable or array location

  a[7]   count=4

- Evaluate a simple expression

  2+4   a<b

- Return from a method

  return 4

Strategies for counting primitive operations

Easiest case:

1. Identify all primitive operations
2. Determine how many time each one happens
3. Add them all up.

n = a.length

findMax - runtime

get
set
evaluate
return

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

set get

1· set

eval (n-1)

eval set (n-1)

eval get

set get

return

1 →

(n-1)

(n-1)

1 →

1 · 2

1 + (n-1)3

(n-1) 2

(n-1) 2

2
1+ 3(n - 1)
2(n-1)
2(n-1)
1

$7(n-1)+4$

1

$7(n-1)+4$

$7n-7+4$

$\boxed{7n-3}$

findMax - runtime

get
set
evaluate
return

```
                          public int findMax(int[] a) {
                                                 set    get
                      1      int currentMax = a[0];              set
                                        set       eval            get
1 + (N-1) + 2(N-1)           for (int i = 1; i < a.length; i++) {
                                                   eval  get
             2(N-1)             if (currentMax < a[i]) {
                                                 set   get
             2(N-1)                currentMax = a[i];

                                }

                             }
                                       return
                  1          return currentMax;

     = 7N-4              }
```

sillyFindMax - runtime

get
set
evaluate
return

```
public int sillyFindMax(int[] a) {

  for (int i = 0; i < a.length; i++) {

    // check if anything is bigger than a[i]
    boolean isMax = true;

    for (int j = 0; j < a.length; j++) {

      if (a[j] > a[i]) {
        isMax = false; // found something bigger
      }
    }
    if (isMax) {
      return a[i];
    }
  }
}
```

$1 + 3n$

n

$n (1 \quad 1 \quad 1n \quad 2n)$

$n + 3n^2$

$1 \quad 1 \quad 1$

$3n^2$

n^2

n

1

$7n^2 + 6n + 2$

sillyFindMax

```
public int sillyFindMax(int[] a) {
  for (int i = 0; i < a.length; i++) {
    // check if anything is bigger than a[i]
    boolean isMax = true;
    for (int j = 0; j < a.length; j++) {
      if (a[j] > a[i]) {
        isMax = false;  // found something bigger
      }
    }
    if (isMax) {
      return a[i];
    }
  }
}
```

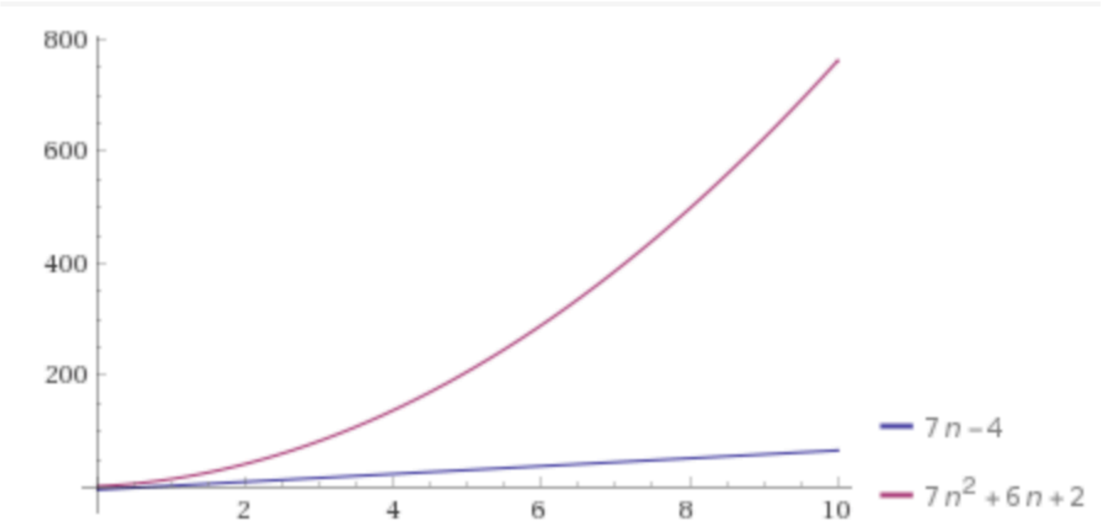
sillyFindMax

```
public int sillyFindMax(int[] a) {
  for (int i = 0; i < a.length; i++) {          1 + N + 2N
    // check if anything is bigger than a[i]
    boolean isMax = true;                                N
    for (int j = 0; j < a.length; j++) {       N (1+N+2N)
      if (a[j] > a[i]) {                            N (3N)
        isMax = false;  // found something bigger      N*N
      }
    }
    if (isMax) {                                         N
      return a[i];                                       1
    }
  }
}
```

$2 + 5N + 6N^2$

Comparing findMaxes

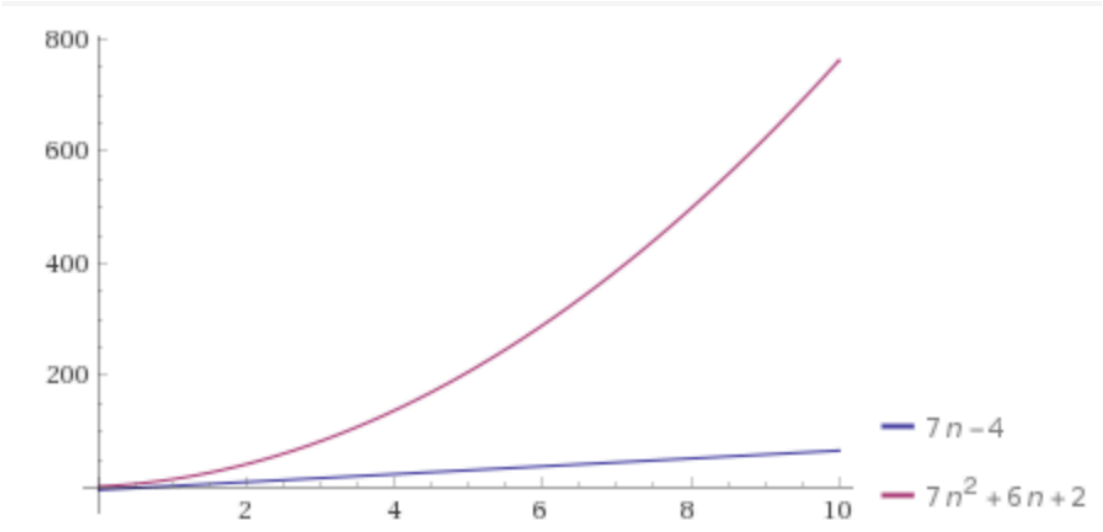
- findMax: 7N-4
- sillyFindMax: $7N^2 + 6n + 2$



Comparing findMaxes

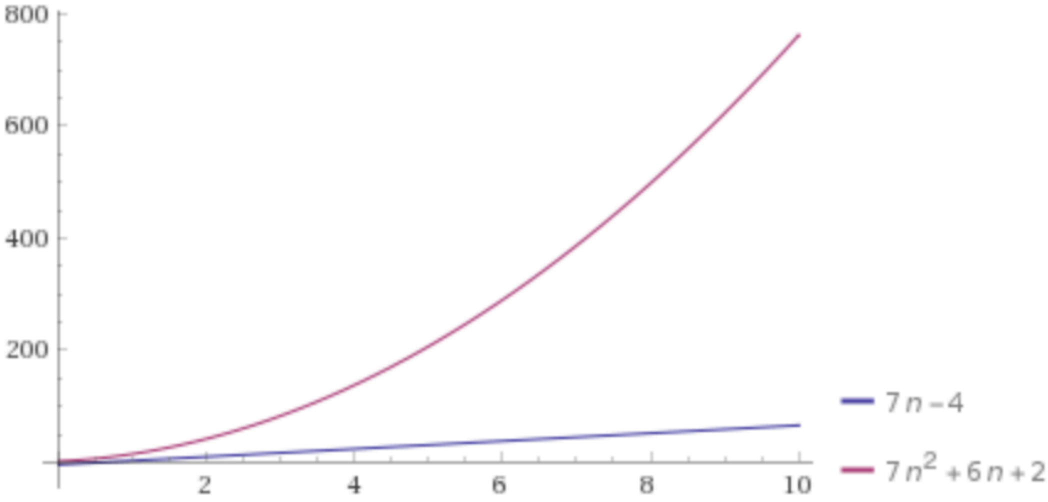
- findMax: 7N-4
- sillyFindMax: $7N^2 + 6n + 2$

sillyFindMax is $O(n^2)$



Comparing findMaxes

- findMax: 7N-4
- sillyFindMax: $7N^2 + 6n + 2$



sillyFindMax is $O(n^2)$

findMax is $O(n)$

Strategies for counting primitive operations

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.

Insertion Sort: Runtime

```
// 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++
```

1: j=0 .. n

2: j=1 .. n

2 .. n

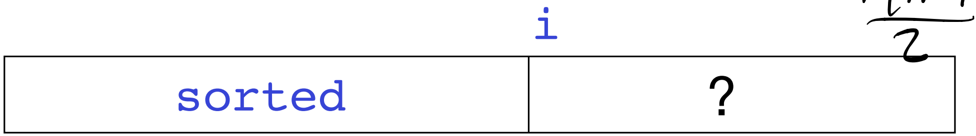
⋮

n

$$\begin{array}{r} n \\ n-1 \\ n-2 \\ \vdots \\ +\quad 1 \\ \hline \end{array}$$

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

Invariant: A



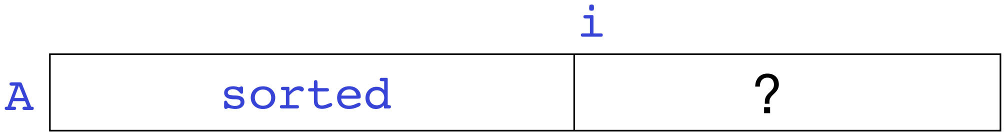
AT MOST How many times do we call swap() during iteration i?

$$\underset{\underset{O(n^2)}{\uparrow}}{\frac{1}{2}n^2} - \frac{1}{2}n$$

Insertion Sort: Runtime

```
// 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:



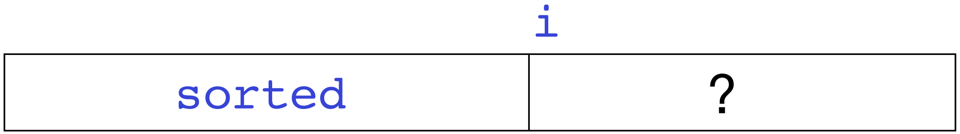
AT MOST How many times do we call swap() during iteration i?

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

Insertion Sort: Runtime

```
// 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?

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

Let's talk about recursion.

Why are we talking about recursion, I thought we were learning about sorting?

Why are we talking about recursion, I thought we were learning about sorting?

```
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end + start)/2
  mergeSort(A, start, mid)
  mergeSort(A, mid, end)
  merge(A, start, mid, end)
```

How do we execute recursive methods?

How do we execute non-recursive methods?

```
x = max(1,3)
```

How do we execute non-recursive methods?

```
x = max(1,3)
    => 3
```

How do we execute non-recursive methods?

```
x = max(1,3)
          3
```

How do we execute non-recursive methods?

`x = 3`

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
   if n == 0:
         return 1
    return n * fact(n - 1)
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
    if n == 0:
            return 1
     return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
  return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * fact(0)
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
  return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * fact(0)
                     => 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * fact(0)
                          1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
                  1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
         => 2 * 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
              2
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * 2
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 6
```

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

Problem 1: If I call `fib(3)`,

A. How many times is `fib` called?

B. What value is returned?

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

1A - ABCD:

A. 3
B. 4
C. 5
D. 6

Problem 1: If I call `fib(3)`,

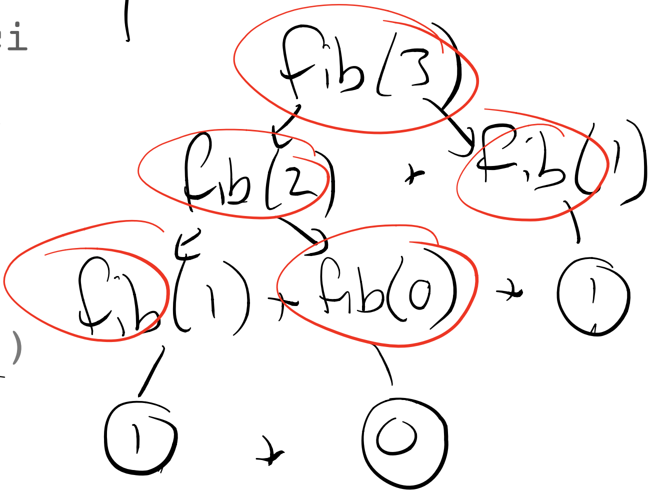
A. How many times is `fib` called?
B. What value is returned?

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci
  * number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```



If I call `fib(3)`,

A. How many times is `fib` called? 5

B. What value is returned? 2

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

Problem 2: If I call `fib(4)`,

A. How many times is `fib` called?

B. What value is returned?

How do we understand recursive methods?