CSCI 241

Lecture 4:
Recursion

Announcements

- First programming assignment (A1) out today(ish)

Today

- Runtime of InsertionSort and SelectionSort
- Recursion: how to execute it
- Recursion: how to think about it

```
selectionSort(A):
  i = 0;
  while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
```

ABCD: What's the best and worst-case asymptotic runtime complexity of selectionSort?

	Best	Worst
A	O(n)	O(n)
B	$O(n^2)$	O(n)
C	O(n)	$O(n^2)$
D	$O(n^2)$	$O(n^2)$

```
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++
```

ABCD: What's the best and worst-case asymptotic runtime complexity of insertionSort?

	Best	Worst
A	$O(n)$	$O(n)$
B	$O(n^2)$	$O(n)$
C	$O(n)$	$O(n^2)$
D	$O(n^2)$	$O(n^2)$

Why is this best-case runtime interesting?

```
insertionSort1(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++
```

```
insertionSort2(A):
  i = 0;
  while i < A.length:
    j = i;
    tmp = A[i];
    while j > 0 and tmp < A[j-1]:
      A[j] = A[j-1]
      j--
    i++
```

ABCD: What's the best and worst-case asymptotic runtime complexity of insertionSort2?

	Best	Worst
A	$O(n)$	$O(n)$
B	$O(n^2)$	$O(n)$
C	$O(n)$	$O(n^2)$
D	$O(n^2)$	$O(n^2)$

Why are we talking about recursion, I thought we were learning how to sort things?

```
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)
```

Goals:

- Understand how recursive methods are **executed**.
- Be able to **understand** and **develop** recursive methods *without* thinking about the call stack.

How do we execute recursive methods?

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 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
```

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)
=> 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

```
/** 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? (show your work)
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

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

A. How many times is `fib` called? (show your work)
B. What value is returned?

How do we understand recursive methods?

1. Make sure it has a **precise specification**.
2. Make sure it works in the **base case**.
3. Ensure that each recursive call makes **progress** towards the base case.
4. Replace each **recursive call** with the **spec** and verify overall behavior is correct.

How do we understand recursive methods?

```
def count_e(s):
   """ returns # of 'e' in string s
   """
   if len(s) == 0:
      return 0
   first = 0
   if s[0] == 'e':
      first = 1

   return first + count_e(s[1..end])
```

1. **spec**

2. **base case**

3. **progress**

4. **recursive call —> spec**

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

This code has **at least one** bug:

```
dup(String s):
  if s.length == 0:
     return s

  return s[0] + s[0] + dup(s)
```

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
   if s.length == 0:
       return s

   return s[0] + s[0] + dup(s)
```

1. **spec!**

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
  if s.length == 0:
     return s

  return s[0] + s[0] + dup(s)
```

3. **progress!**

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
  if s.length == 0:
     return s

  return s[0] + s[0] + dup(s[1..s.length])
```

3. **progress!**

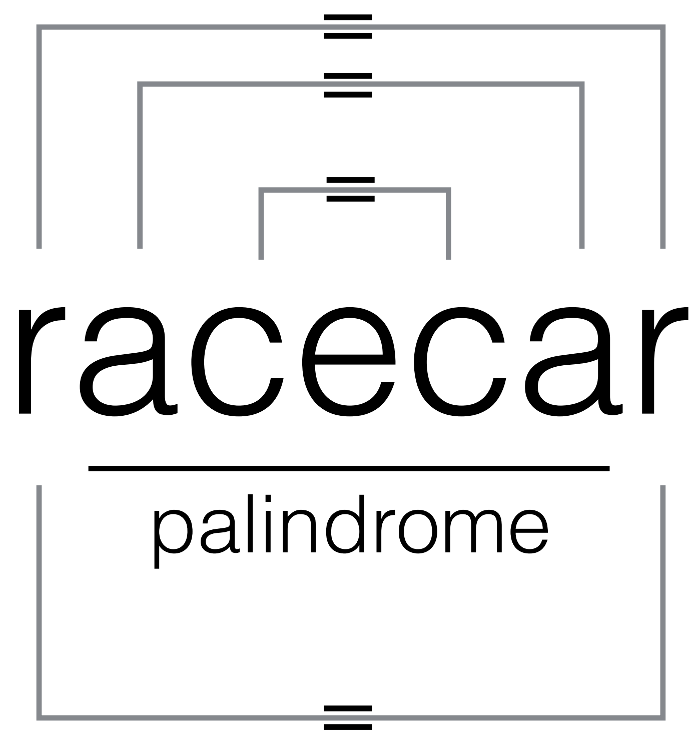
How do we develop recursive methods?

1. Write a **precise specification**.
2. Write a **base case** without using recursion.
3. Define all other cases in terms of **subproblems** of the same kind.
4. Implement these definitions using the **recursive call** to compute solutions to the subproblems.

Palindromes

Examples:

- civic
- radar
- deed
- racecar



Recursive definition: A string s is a palindrome if

- `s.length < 2`, OR
- `s[0] == s[end-1]` AND s[1..end-2] is a palindrome

racecar

palindrome

=

Recursive definition: A string so is a palindrome if

- `s.length < 2`, OR
- `s[0] == s[end-1]` AND s[1..end-2] is a palindrome

Problem 3: Write a recursive palindrome checker:

```
/** return true iff s[start..end]
  * is a palindrome */
public boolean isPal(s, start, end) {
  // your code here
}
```