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

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

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>B</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>C</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>D</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
`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++
```

ABC: What’s the best and worst-case asymptotic runtime complexity of insertionSort?

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>B</td>
<td>O(n²)</td>
<td>O(n)</td>
</tr>
<tr>
<td>C</td>
<td>O(n)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>D</td>
<td>O(n²)</td>
<td>O(n²)</td>
</tr>
</tbody>
</table>

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]:
            j--
        i++

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

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>B</td>
<td>O(n^2)</td>
<td>O(n)</td>
</tr>
<tr>
<td>C</td>
<td>O(n)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>D</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
</tbody>
</table>
Why are we talking about recursion, I thought we were learning how to sort things?

```python
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?

\[ x = \text{max}(1, 3) \]
\[ \Rightarrow 3 \]
How do we execute non-recursive methods?

\[ x = \max(1, 3) \]
How do we execute recursive methods?

```python
/** return n!; pre: n >= 0 */

def 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?

```python
/** 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?

```python
/** 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?

```python
/** 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?

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

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

Fibonacci:

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>

/** 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:

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>

`/** 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
/** return the nth fibonacci number
 * precondition: n >= 0 */

fib(n):

if n <= 1:
    return n

return fib(n-1) + fib(n-2)

Fibonacci:

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>

Problem 2: If I call fib(4),
A. How many times is fib called? (show your work)
B. What value is returned?

1A - ABCD:
A. 3
B. 4
C. 5
D. 6
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
This code has **at least one** bug:

\[
\text{dup(String s):}
\]

\[
\text{if s.length == 0:}
\]

\[
\text{return s}
\]

\[
\text{return s[0] + s[0] + dup(s)}
\]
/** 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
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?

```java
/** 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])
```

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

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.
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
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:

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