The academic forum is an opportunity for students to provide feedback to the department on all aspects of the undergraduate academic experience, ranging from course evaluation policies and course content to program graduation requirements. CSUS will be moderating a roundtable discussion between students and faculty and provide snacks.

Academic Forum

Wednesday January 31, 2:30-4pm
ENGMC 103
COMP250: Thinking Recursively. Examples.

Jérôme Waldispühl
School of Computer Science
McGill University

Based on slides from (Hatami), (Bailey), (Stepp & Martin)
Course credits

c(x) = total number of credits required to complete course x

c(COMP462) = ?

= 3 credits + \#credits for prerequisites

COMP462 has 2 prerequisites: COMP251 & MATH323

= 3 credits + c(COMP251) + c(MATH323)

The function c calls itself twice

c(COMP251) = ? c(MATH323) = ?

c(COMP251) = 3 credits + c(COMP250) COMP250 is a prerequisite

Substitute c(COMP251) into the formula:

c(COMP462) = 3 credits + 3 credits + c(COMP250) + c(MATH323)

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)
Course credits

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)
  c(COMP250) = ?  c(MATH323) = ?
  c(COMP250) = 3 credits  \# no prerequisite

c(COMP462) = 6 credits + 3 credits + c(MATH323)
  c(MATH323) = ?
  c(MATH323) = 3 credits + c(MATH141)

c(COMP462) = 9 credits + 3 credits + c(MATH141)
  c(MATH141) = ?
  c(MATH141) = 4 credits  \# no prerequisite

c(COMP462) = 12 credits + 4 credits = 16 credits
A noun phrase is either
- a noun, or
- an adjective followed by a noun phrase

\(<\text{noun phrase}\> \rightarrow \text{noun} \text{ OR } \text{adjective} \text{ <noun phrase>}

```
<noun phrase>
  /     \
<adjective> <noun phrase>
     /     \
<adjective> <noun phrase>
      /     \
  big  black  dog
```
Definitions

Recursive definition:
A definition that is defined in terms of itself.

Recursive method:
A method that calls itself (directly or indirectly).

Recursive programming:
Writing methods that call themselves to solve problems recursively.
Why using recursions?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- Recursion is often a good alternative to iteration (loops).
Iterative algorithms

Definition (iterative algorithm): Algorithm where a problem is solved by iterating (going step-by-step) through a set of commands, often using loops.

**Algorithm**: power(a,n)

**Input**: non-negative integers a, n

**Output**: $a^n$

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>1</td>
<td>a</td>
<td>$a \cdot a = a^2$</td>
<td>$a^2 \cdot a = a^3$</td>
<td>$a^3 \cdot a = a^4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>1</td>
<td>a</td>
<td>$a \cdot a = a^2$</td>
<td>$a^2 \cdot a = a^3$</td>
<td>$a^3 \cdot a = a^4$</td>
</tr>
</tbody>
</table>
Recursive algorithms

Definition (Recursive algorithm): algorithm is recursive if in the process of solving the problem, it calls itself one or more times.

Algorithm: power(a,n)
Input: non-negative integers a, n
Output: $a^n$
if (n=0) then
  return 1
else
  return a * power(a,n-1)
Example

- power(7,4) calls
  - power(7,3) calls
    - power(7,2) calls
      - power(7,1) calls
        - power(7,0) returns 1
          - returns 7 * 1 = 7
            - returns 7 * 7 = 49
              - returns 7 * 49 = 343
                - returns 7 * 343 = 2041
Algorithm structure

Every recursive algorithm involves at least 2 cases:

**base case:** A simple occurrence that can be answered directly.

**recursive case:** A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

Some recursive algorithms have more than one base or recursive case, but all have at least one of each.

A crucial part of recursive programming is identifying these cases.
Algorithm binarySearch(array, start, stop, key)

Input: - A sorted array
- the region start...stop (inclusively) to be searched
- the key to be found

Output: returns the index at which the key has been found, or returns -1 if the key is not in array[start...stop].

Example: Does the following sorted array A contains the number 6?

A =  

1 1 3 5 6 7 9 9

Call: binarySearch(A, 0, 7, 6)
Binary search example

1 1 3 5 6 7 9 9

Search [0:7]

5 < 6 ⇒ look into right half of the array

1 1 3 5 6 7 9 9

Search [4:7]

7 > 6 ⇒ look into left half of the array

1 1 3 5 6 7 9 9

Search [4:4]

6 is found. Return 4 (index)
Binary Search Algorithm

```c
int bsearch(int[] A, int i, int j, int x) {
    int e = (i+j)/2; // Find middle point
    if (A[e] > x) {  // key is lower, look to left half
        return bsearch(A,i,e-1,x);
    } else if (A[e] < x) {  // key is bigger, look to right half
        return bsearch(A,e+1,j,x);
    } else {  // value x is found
        return e;
    }
}
```
Binary Search Algorithm

```c
int bsearch(int[] A, int i, int j, int x) {
    if (i <= j) {  // the region to search is non-empty
        int e = \[(i+j)/2\];
        if (A[e] > x) {
            return bsearch(A, i, e-1, x);
        } else if (A[e] < x) {
            return bsearch(A, e+1, j, x);
        } else {
            return e;
        }
    } else { return -1; }  // value not found
}
```
Fibonacci numbers

\[
\begin{align*}
\text{Fib}_0 &= 0 & \text{base case} \\
\text{Fib}_1 &= 1 & \text{base case} \\
\text{Fib}_n &= \text{Fib}_{n-1} + \text{Fib}_{n-2} \text{ for } n > 1 & \text{recursive case}
\end{align*}
\]

<table>
<thead>
<tr>
<th>i</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>
</tr>
</thead>
<tbody>
<tr>
<td>Fib\text{i}</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>
</tr>
</tbody>
</table>
Recursive algorithm

Compute Fibonacci number n (for $n \geq 0$)

```java
public static int Fib(int n) {
    if (n <= 1) {
        return n;  // Can handle both base cases together
    }
    // {n > 0}
    return Fib(n-1) + Fib(n-2);  // Recursive case (2 recursive calls)
}
```

**Note:** The algorithm follows almost exactly the definition of Fibonacci numbers.
Recursion is not always efficient!

Question: When computing Fib(n), how many times are Fib(0) or Fib(1) called?