Recursion

Lecture 16
ICOM 4075
What is recursion?

**Definition**: A function or procedure is said to be **recursively defined** if it is defined in terms of itself

- For a function $f$, this means that at least one value $f(x)$ is defined in terms of another value $f(y)$, where $x \neq y$

- For a procedure $P$, this means that the action of $P$ on some input $x$ are defined in terms of the action of $P$ on another input $y$, $x \neq y$
Illustration

Example of a function recursively defined:
Let $f : \text{Nat} \rightarrow \text{Nat}$ be defined by

$$f(0) = 0$$
$$f(n) = f(n-1) + n$$

An evaluation:

$$f(3) = f(2) + 3$$
$$= f(1) + 2 + 3 = f(1) + 5$$
$$= f(0) + 1 + 5 = 0 + 6 = 6$$
Illustration

To implement the previous function as a program (or algorithmic function), we may write

Type: $F: \text{Nat} \rightarrow \text{Nat}$

Pseudo-code:

$F(n)$:

If $n = 0$

Return 0

Else

Return $F(n-1) + n$

How can we implement such a procedure?
Recursion in a computer

The computer performs recursion by constructing a stack of pending function calls:

<table>
<thead>
<tr>
<th>n</th>
<th>Fifth</th>
<th>F(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Forth</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Third</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Second</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>First</td>
<td></td>
</tr>
</tbody>
</table>
Recursion in a computer

The computer performs recursion by constructing a stack of **pending function calls**:

<table>
<thead>
<tr>
<th>n</th>
<th>Fifth</th>
<th>F(0)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Forth</td>
<td>F(1)</td>
<td>1 + 0 = 1</td>
</tr>
<tr>
<td>2</td>
<td>Third</td>
<td>F(2)</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>3</td>
<td>Second</td>
<td>F(3)</td>
<td>3 + 3 = 6</td>
</tr>
<tr>
<td>4</td>
<td>First</td>
<td>F(4)</td>
<td>6 + 4 = 10</td>
</tr>
</tbody>
</table>

5 function calls

Filling the stack

Returning

Returned value
Counting function calls and evaluations

The time it takes for an algorithmic function to complete a computation is roughly proportional to the number of steps it take to complete the computation on a given input.

Example: Procedure F computes F(4) in a total of 10 functions calls and evaluations.

It is easy to see that, given a general input n, F computes F(n) in 2(n + 1) function calls and evaluations.
Table of counts for F(n)

Some values are:

<table>
<thead>
<tr>
<th>n</th>
<th>2(n + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>42</td>
</tr>
<tr>
<td>30</td>
<td>62</td>
</tr>
<tr>
<td>40</td>
<td>82</td>
</tr>
</tbody>
</table>

Linear graph:

Roughly speaking, the execution time of F must grow as a linear function of n.
Computing Fibonacci numbers recursively

Recall the inductive definition:
Fib(0)=0, Fib(1) = 1 and Fib(n) = Fib(n-1)+Fib(n-2), n > 1

We discussed an **inductive method** for computing Fib(n):

\[
\text{Inductive}\_\text{Fibo}(n)
\]

1. \text{If } n = 0 \text{ return 0}
2. \text{If } n = 1 \text{ return 1}
3. \text{U } \leftarrow 1
4. \text{V } \leftarrow 0
5. \text{Z } \leftarrow 1
6. \text{For } j = 2 \text{ to } n
7.     \text{V } \leftarrow \text{U}
8.     \text{U } \leftarrow \text{Z}
9.     \text{Z } \leftarrow \text{U} + \text{V}
10. \text{Return } Z
Computing Fibonacci numbers recursively

Recall the inductive definition:
Fib(0)=0, Fib(1) = 1 and Fib(n) = Fib(n-1)+Fib(n-2), n > 1

A **recursive method** for computing Fib(n) is

Fib(n):

If n = 0  
Return 0

Else if n = 1  
Return 1

Else  
Return Fib(n-1) + Fib(n-2)
Computing Fibonacci numbers recursively

Recall the inductive definition:
Fib(0)=0, Fib(1) = 1 and Fib(n) = Fib(n-1)+Fib(n-2), n > 1

A recursive method for computing Fib(n) is

Fib(n):
- If n = 0
  - Return 0
- Else if n = 1
  - Return 1
- Else
  - Return Fib(n-1) + Fib(n-2)

Much shorter and “elegant”
Counting pending calls in the recursive Fibonacci method

However, the structure of the function calls is now much complicated

Let’s take \( n = 4 \)

A TOTAL OF 9 FUNCTION CALLS
Counting function calls and evaluations in recursive Fibonacci

As illustrated in the previous example, recursion involve as many calls as nodes in a binary tree. More precisely, if the input is n, the numbers of function calls is approximately $2^{n-1} + 1$

This is, the number of calls and function evaluations grows exponentially

Indeed, for $n = 40$ there will be $549,755,813,889$ function calls!
One more example: Finding a common prefix

**Definition:** A prefix of a string $x$ is a substring $p$ of $x$ such that $x = ps$. String $s$ is said to be a suffix.

**Example:** Let $x = 010010$. Then, the prefixes of $x$ are:

- $p = 0$ (here $s = 10010$)
- $p = 01$ (here $s = 0010$)
- $p = 010$ (here $s = 010$)
- $p = 0100$ (here $s = 10$)
- $p = 01001$ (here $s = 0$), and
- $p = 010010$ (here $s = \lambda$)
A string matching problem

**Problem**: Given two strings $s$ and $t$ over the same alphabet, find the longest common prefix

**Example**: Suppose we are given

\[ s = 010010, \text{ and } \]
\[ t = 011001100110 \]

Then, by comparing their prefixes we see that the longest common prefix is $p = 01$
Recursive solution

Assume the given strings \( s \) and \( t \) are strings over the alphabet \( \{0, 1\} \). The recursive function:

\[
\text{LongestPrefix}(s, t):
\]

If \( s = \lambda \) or \( t = \lambda \) return \( \lambda \)

Else if \( s = 0x \) and \( t = 0y \)

\[
\text{Return } 0\text{LongestPrefix}(x, y)
\]

Else if \( s = 1x \) and \( t = 1x \)

\[
\text{Return } 1\text{LongestPrefix}(x, y)
\]

Else return \( \lambda \)

solves the longest common string problem
Some computations

Let $s = 010010$, and $t = 011001100110$. Then,

$$\text{LongestPrefix}(010010, 011001100110) = 0$$
$$\text{LongestPrefix}(10010, 11001100110) = 0$$
$$\text{LongestPrefix}(0010, 1001100110) = 01$$

The answer is $p = 01$ (as expected)
Bounding the count: worst case

In this case, the number of steps may vary from 0 to $n$, where $n$ is the shortest of the two sequences. In these cases, the complexity is estimated by

Counting pending function calls and evaluation in the **Worst-case time scenario.** This is, the largest possible number of steps involved in the process. Such number is called **upper complexity bound**

The upper complexity bound of LongestPrefix grows linearly
Summary

• Definition of recursion
• Recursive functions and recursive procedures
• Recursion in a computer: stacks of function calls
• Time complexity
• Linear growth
• Fibonacci numbers by recursion
• Exponential growth
• A string matching problem
• Upper complexity bounds
Exercises

1. Given the following procedure:

   LengthList(L):

   If L = <> Return 0

   Else Return 1 + LengthList(tail(L))

   a) Show the steps in the computation of L = <a, b, c, d, e>

   b) Decide whether the time complexity of this procedure is linear or exponential. Justify your answer
Exercises

2. Consider the procedure defined for binary trees:

   \[\text{SearchNode( Tree, node)}\]
   
   \[\text{If root(Tree) = node return “node found”}\]
   
   \[\text{Else}\]
   
   \[\text{SearchNode(Left subtree, node)}\]
   
   \[\text{SearchNode(Right subtree, node)}\]
   
   Return “node not found”

a) Given the Tree = \[<<<<D>,B,<E>>, A ,<<F>,C,>>\], show the steps for computing SearchNode(Tree, F) and the steps for computing SearchNode(Tree, K)

b) Estimate the worst-case time complexity
Exercises

3. Construct a recursive procedure for computing each of the following functions:

a) The function \( f : \text{Nat} \rightarrow \text{Nat} ; f(n) = \sum_{i=0}^{n} \left\lfloor \frac{i}{3} \right\rfloor \)

b) The function \( f : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} ; f(n, k) = \sum_{i=0}^{n} k + 2i \)

c) The map
\[
 f : \text{Strings} \times \text{Strings} \rightarrow \{0, 1\}; f(x, y) = \begin{cases} 
0 & \text{if } x \neq y \\
1 & \text{otherwise}
\end{cases}
\]

d) The map \( f : \text{Strings} \times \text{Strings} \rightarrow \text{Strings} \)
\[
f(x, y) = \begin{cases} 
1, & \text{if } x \text{ is a prefix of } y \\
0, & \text{otherwise}
\end{cases}
\]

Give the time complexity in each case.
Exercises

4. Count the actual number of pending function calls and call evaluations for the recursive Fibonacci algorithm, for \( n = 2, 3, 4, 5, 6, \) and \( 7. \) Draw the graph of \( n \) vs. the addition of the number of pending calls and function evaluations and compare it with the graph of \( f(n) = 2^{n-1} + 1 \) for the same values of \( n. \) Do you think \( f(n) \) is a good approximation to the actual count?