Lecture 8: Recursion

Bohyung Han
CSE, POSTECH
bhhan@postech.ac.kr

Recursion

- Divide a problem into subproblems of the same type
- Recursive function: a method (function) calling itself

A classic example
- Factorial function
  \[ n! = 1 \cdot 2 \cdot 3 \cdots (n - 1) \cdot n \]
- Recursive definition
  \[
  f(n) = \begin{cases} 
  1 & \text{if } n = 0 \\ 
  n \cdot f(n - 1) & \text{otherwise}
  \end{cases}
  \]

Recursive Function

```java
int Factorial(int n) {
    if (n == 0) return 1; // base case
    else return n * Factorial(n - 1); // recursive case
}
```

- Base case(s)
  - Values of the input variables for which we perform no recursive calls
  - There should be at least one base case
  - Every possible chain of recursive calls must reach a base case.
- Recursive case(s)
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

```
pubic void main(String args[]) {
    int fac;
    fac = Factorial(4);
    System.out.println("fac = "+fac);
}
```

Output: fac = 24
Reversing an Array

Algorithm `ReverseArray(A, i, j)`:
- **Input:** An array A and nonnegative integer indices i and j
- **Output:** The reversal of the elements in A starting at index i and ending at j
  - if i < j then
    - `ReverseArray(A, i+1, j-1)`
    - Swap A[i] and A[j]
  - return

<table>
<thead>
<tr>
<th>A</th>
<th>3</th>
<th>6</th>
<th>1</th>
<th>9</th>
<th>4</th>
<th>2</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ReverseArray(A, 8, 8)
ReverseArray(A, 7, 9)
ReverseArray(A, 6, 10)
ReverseArray(A, 5, 11)

Reversing an Array by Tail Recursion

- **Tail Recursion**
  - A linearly recursive method makes its recursive call as its last step.
  - More efficient than ordinary recursions

Algorithm `ReverseArray(A, i, j)`:
- **Input:** An array A and nonnegative integer indices i and j
- **Output:** The reversal of the elements in A starting at index i and ending at j
  - if i < j then
    - Swap A[i] and A[j]
    - `ReverseArray(A, i+1, j-1)`
  - return

<table>
<thead>
<tr>
<th>A</th>
<th>3</th>
<th>6</th>
<th>1</th>
<th>9</th>
<th>4</th>
<th>2</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ReverseArray(A, 5, 11)
ReverseArray(A, 6, 10)
ReverseArray(A, 7, 9)

Recursion vs. Loop

- **Equivalence between recursion and loop**
  - A recursive function can be implemented with a loop.

Algorithm `IterativeReverseArray(A, i, j)`:
- **Input:** An array A and nonnegative integer indices i and j
- **Output:** The reversal of the elements in A starting at index i and ending at j
  - while i < j do
    - Swap A[i] and A[j]
    - i = i + 1
    - j = j - 1
  - return

<table>
<thead>
<tr>
<th>A</th>
<th>7</th>
<th>6</th>
<th>1</th>
<th>9</th>
<th>4</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ReverseArray(A, 5, 11)
ReverseArray(A, 6, 10)
ReverseArray(A, 7, 9)

Computing Powers

- **Objective**
  - Compute the power function, \( p(x, n) = x^n \)
  - Define recursively
    \[
    p(x, n) = \begin{cases} 
    1 & \text{if } n = 0 \\
    x \cdot p(x, n - 1) & \text{otherwise}
    \end{cases}
    \]

- **Time complexity**
  - \( O(n) \): making \( n \) recursive calls
  - We can do better than this, however.
Recursive Squaring

• A better method to compute powers
  ▪ We can derive a more efficient linearly recursive algorithm by using repeated squaring

\[
p(x, n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  x \cdot p(x, (n-1)/2)^2 & \text{if } n \text{ is odd} \\
  p(x, n/2)^2 & \text{if } n \text{ is even}
\end{cases}
\]

• For example,

\[
2^5 = 2 \cdot (2^2)^2 = 2 \cdot (2^4)^2 = 2 \cdot (2^2)^2 = 2 \cdot 4^2 = 2 \cdot 16 = 32
\]

Recursion Squaring

Algorithm \text{Power}(x, n):

\text{Input:} A number x and integer n = 0
\text{Output:} The value } x^n

\begin{align*}
\text{if } n = 0 & \text{ then return } 1 \\
\text{if } n \text{ is odd} & \text{ then return } x \cdot \text{Power}(x, (n-1)/2) \\
\text{else} & \text{ return Power}(x, n/2) \cdot y \cdot y \cdot y
\end{align*}

Each time we make a recursive call we halve the value of n; hence, we make $\log_2 n$ recursive calls. That is, this method runs in $O(\log n)$ time.

Binary Recursion

• What is binary recursion?
  ▪ There are two recursive calls for each non-base case.

Example

▪ Fibonacci number

\[
f(n) = \begin{cases} 
  n & \text{if } n = 0 \text{ or } n = 1 \\
  f(n-1) + f(n-2) & \text{if } n \geq 2
\end{cases}
\]

\begin{verbatim}
int Fibonacci(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
\end{verbatim}

Fibonacci Number

\[
\text{Time complexity: } O(2^n)
\]

However, there are too many duplicate calls!
A Better Fibonacci Algorithm

Algorithm LinearFibonacci(k)
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers \( (F_k, F_{k-1}) \)

if \( k = 1 \) then
    return \( (k, 0) \)
else
    \((i, j) = \text{LinearFibonacci}(k - 1)\)
    return \( (i+j, i) \)

No duplicate calls!

Time complexity: \( O(n) \)