Recursion

Chapter 8

Chapter Contents

• What Is Recursion?
• Tracing a Recursive Method
• Recursive Methods That Return a Value
• Recursively Processing an Array
• Recursively Processing a Linked Chain
• The Time Efficiency of Recursive Methods
  – Time Efficiency of \texttt{countDown}
• A Simple Solution to a Difficult Problem
• A Poor Solution to a Simple Problem

What Is Recursion?

• It is a problem-solving process
• Breaks a problem into identical but smaller problems
• Eventually you reach a smallest problem
  – Answer is obvious or trivial
• Using that solution enables you to solve the previous problems
• Eventually the original problem is solved
• An alternative to iteration
  • An iterative solution involves loops

What Is Recursion?

• A method that calls itself is a recursive method

```java
/** Task: Counts down from a given positive integer. *
 * @param integer an integer > 0 */
 public static void countDown(int integer)
 { System.out.println(integer);
   if (integer > 1)
     countDown(integer - 1);
 } // end countDown
```

What Is Recursion?

• A method that calls itself is a recursive method

```java
/** Task: Counts down from a given positive integer. *
 * @param integer an integer > 0 */
 public static void countDown(int integer)
 { System.out.println(integer);
   if (integer > 1)
     countDown(integer - 1);
 } // end countDown
```

When Designing Recursive Solution

• Four questions for construction recursive solutions
  – How can you define the problem in terms of a smaller problem of the same type?
  – How does each recursive call diminish the size of the problem?
  – What instance of the problem can serve as the base case?
  – As the problem size diminishes, will you reach this base case?
When Designing a Recursive Solution

- Method definition must provide parameter
  - Leads to different cases
  - Typically includes an if or a switch statement
- One or more of these cases should provide a non-recursive solution
  - The base or stopping case
- One or more cases includes recursive invocation
  - Takes a step towards the base case

Tracing a Recursive Method

- Given:
  ```java
class CountDown {
    public static void countDown(int integer) {
        System.out.println(integer);
        if (integer == 1) {
            return;
        } else {
            countDown(integer - 1);
        }
    }
}
```

- The effect of method call `countDown(3)`

Tracing the recursive method with stack activation records:

- Each call to a method generates an activation record that captures the state of the method’s execution and is placed into a stack.
- The activation-record stack remembers the history of incomplete method calls.
- The topmost activation record holds the data values for the currently executing method.
- When the topmost method finishes, its activation record is popped.
Tracing a Recursive Method

• Too many recursive calls can cause the error message “stack overflow”. Stack of activation records has become full. Method has used too much memory.

• Infinite recursion or large-size problems are the likely cause of this error.

Recursive Methods That Return a Value

• Task: Compute the sum $1 + 2 + 3 + \ldots + n$ for an integer $n > 0$

```java
public static int sumOf(int n) {
    int sum;
    if (n == 1) {
        sum = 1; // base case
    } else {
        sum = sumOf(n - 1) + n; // recursive call
    }
    return sum;
} // end sumOf
```

Recursively Processing an Array

• When processing array recursively, divide it into two pieces
  – Last element one piece, rest of array another
  – First element one piece, rest of array another
  – Divide array into two halves

• A recursive method part of an implementation of an ADT is often private
  – Its necessary parameters make it unsuitable as an ADT operation

Recursively Processing a Linked Chain

• To write a method that processes a chain of linked nodes recursively
  – Use a reference to the chain’s first node as the method’s parameter

• Then process the first node
  – Followed by the rest of the chain

```java
public void display() {
    displayChains(list.firstNode(), System.out.print());
}
```

Time Efficiency of Recursive Methods

• For the `countDown` method

```java
public static void countDown(int integer) {
    System.out.println(integer);
    if (integer > 1) {
        countDown(integer - 1);
    }
} // end countDown
```

– The efficiency is $O(n)$
A Simple Solution to a Difficult Problem

The initial configuration of the Towers of Hanoi for three disks

The sequence of moves for solving the Towers of Hanoi problem with three disks.

Continued →

The smaller problems in a recursive solution for four disks

Algorithm for solution with 1 disk as the base case

```python
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
    solveTowers(numberOfDisks-1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks-1, tempPole, startPole, endPole)
```

Rules for the Towers of Hanoi game

1. Move one disk at a time. Each disk you move must be a topmost disk.
2. No disk may rest on top of a disk smaller than itself.
3. You can store disks on the second pole temporarily, as long as you observe the previous two rules.
Recursion Efficiency

• How many moves occur for n disks?
• m(1) = 1 for n>1, two recursive calls to solve problems that have n-1 disks.
• m(n) = m(n-1) + m(n-1) + 1 = 2*m(n-1) + 1
• Let’s evaluate the recurrence for m(n) for a few values of n:
  - m(1) = 1; m(2) = 3; m(3) = 7; m(4) = 15; m(5) = 31; m(6) = 63....
  - m(n) = 2^n - 1

Mathematical Induction

• Prove this conjecture m(n) = 2^n - 1 by using mathematical induction:
• We know that m(1) = 1, which equals to 2^1-1=1, so the conjecture is true for n =1.
• Now assume that it is true for n=1,2,...,k, and consider m(k+1).
• m(k+1) = 2*m(k) + 1 (use the recurrence relation)
  - =2*(2^k-1) + 1 = 2^(k+1) - 1 (we assume that m(k) = 2^k-1)
• Since the conjecture is true for n=k+1, it is true for all n>=1

Mathematical Induction

• Assume you want to prove some statement P, P(n) is true for all n starting with n = 1. The Principle of Math Induction states that, to this end, one should accomplish just two steps:
  1). Prove that P(1) is true.
  2). Assume that P(k) is true for some k. Derive from here that P(k+1) is also true.

• look P(1) is true and implies P(2). Therefore P(2) is true. But P(2) implies P(3). Therefore P(3) is true which implies P(4) and so on.

Multiplying Rabbits (The Fibonacci Sequence)

• “Facts” about rabbits
  - Rabbits never die
  - A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  - Rabbits are always born in male-female pairs
    • At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair

Multiplying Rabbits (The Fibonacci Sequence)

• Problem
  – How many pairs of rabbits are alive in month n?
  – Month  Month No calculation rabbit couples
    – January  1  1 + 0 = 1
    – February 2  1 + 0 = 1
    – March  3  1 + 1 = 2
    – April  4  2 + 1 = 3
    – May  5  3 + 2 = 5
    – June  6  5 + 3 = 8
    – July  7  8 + 5 = 13
    – August  8  13 + 8 = 21
• Recurrence relation
  rabbit(n) = rabbit(n-1) + rabbit(n-2)

A Poor Solution to a Simple Problem

• Fibonacci numbers
  – First two numbers of sequence are 1 and 1
  – Successive numbers are the sum of the previous two
    – 1, 1, 2, 3, 5, 8, 13, ...
  – This has a natural looking recursive solution
    – Turns out to be a poor (inefficient) solution
A Poor Solution to a Simple Problem

• The recursive algorithm

\textit{Algorithm Fibonacci(n)}
\begin{algorithmic}
\Function{Fibonacci}{n}
\If{\(n \leq 1\)}
\State \textbf{return} 1
\EndIf
\If{\(n \gt 1\)}
\State \textbf{return} \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)
\EndIf
\EndFunction
\end{algorithmic}

The computation of the Fibonacci number \(F_6\)

\begin{enumerate}
\item recursively;
\item iteratively
\end{enumerate}

Time efficiency grows exponentially with \(n\), which is \(k^n\)

Iterative solution is \(O(n)\)