Chapter 17

Recursion
Recursion

♦ Outline

- Consider some recursive solutions to non-computer problems
- Compare iterative and recursive solutions
- Identify the recursive case and the base cases in some simple recursive algorithms
- Implement recursive methods (methods that call themselves)
Recursion can describe everyday examples

- Show everything in a folder and all its subfolders
  - show everything in top folder
  - show everything in each subfolder in the same manner
- How to look up a word in a dictionary
  - look up a word (use alphabetical ordering) or
  - look up word to define the word you are looking up
- Take attendance:
  
  if you are in the last row
    return #students in your row
  
  else
    return #students behind you + #students in your row
Recursive definition:  
Arithmetic Expression

- Arithmetic expression is defined as
  - a numeric constant
  - an numeric identifier
  - an arithmetic expression enclosed in parentheses
  - 2 arithmetic expressions with a binary operator like + - / or *

Note: The term arithmetic expression is defined with the term arithmetic expression
  - but the first two bullets do not
Mathematical Examples

♦ Consider the factorial method (0!=1)

\[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]

♦ The recursive definition:

\[
f(n) = \begin{cases} 
  n \geq 1 & \Rightarrow n \times f(n - 1) \\
  n = 0 & \Rightarrow 1 
\end{cases}
\]

♦ What is \( f(2) \)? _____________

♦ What is \( f(3) \)? _____________
Recursive and non-recursive solutions

// Non-recursive solution, using a loop
// precondition: n >= 0
public long factRep(int n) {
    long result = 1;
    for(int lcv = 2; lcv <= n; lcv++)
        result = lcv * result;
    return result;
}

// Recursive solution
// precondition: n >= 0
public long factRec(int n) {
    if(n == 0)
        return 1;  // base case
    else
        return n * factRec(n - 1);  // Recursive case
}  // Don't call factRec(n + 1)!

@Test
given void testFacorialMethods() {  
    assertEquals(1, factRep(0));
    assertEquals(1, factRec(0));
    assertEquals(1, factRep(1));
    assertEquals(1, factRec(1));
    assertEquals(2, factRep(2));
    assertEquals(2, factRec(2));
    assertEquals(6, factRep(3));
    assertEquals(6, factRec(3));
    assertEquals(24, factRep(4));
    assertEquals(24, factRec(4));
    assertEquals(3628800, factRep(10));
    assertEquals(3628800, factRec(10));
}
Trace factRec(4)

- Method calls itself until base case is reached

```
factRec( 0 ) = 1
```

```
factRec( 1 ) = 1 * 1
```

```
factRec( 2 ) = 2 * 1
```

```
factRec( 3 ) = 3 * 2
```

```
factRec( 4 ) = 4 * 6
```

24 replaces original method call
Return num to the $b$ power

- Write $\text{pow}(\text{int } a, \text{ int } b)$ using this recursive definition

$$\text{pow}(\text{num, pow}) = \begin{cases} 
\text{pow} == 0 \Rightarrow 1 \\
\text{pow} == 1 \Rightarrow \text{num} \\
\text{pow} \geq 2 \Rightarrow \text{num} \times \text{pow}(\text{num, pow - 1})
\end{cases}$$
Determine base and recursive cases

♦ When writing recursive methods
  — Make sure there is at least one base case
    • at least one situation where a recursive call is *not* made. There could be more than one base case
      – The method might return a value, or do nothing at all
  — There could be one or more recursive cases
    • a recursive call must be a simpler version of the same problem
      – the recursive call should bring the method closer to the base case *perhaps pass n-1 rather than n.*
Palindrome

♦ Palindrome is a String that equals itself when reversed: "racecar" "abba" "12321"
♦ Write a recursive method that returns true if a given string is a palindrome
  — What is/are the base case(s)?

```java
public void testIsPalindrome() {
    assertTrue(isPalindrome("A"));
    assertTrue(isPalindrome("racecar"));
    assertFalse(isPalindrome("not"));
    assertFalse(isPalindrome("Aba"));
    assertFalse(isPalindrome("1231"));
    assertFalse(isPalindrome("1233 21"));
}
```

— What is the recursive case?
Recursion = Recursion( Again-1 );
A Combinatorial method

♦ This example of a recursive solution comes from the field of Combinatorics
♦ Problem: A D.J. plays 10 songs each hour. There are 40 different songs. How many different one hour sets are possible?
♦ Or in general given 40 different things, how many sets of size 10 can be chosen
♦ Or using the standard notation – \( n \) choose \( k \)

\[
\binom{n}{k}
\]
Recursive n choose k (con.)

- In any given hour, we could play "Stairway"
- All the possible sets are those that have "Stairway" and those that don't (sum below)
- After picking this one song--now a simpler problem--the DJ must pick 9 more from the remaining 39 songs (from 39 choose 9)
- Possible sets are those with "Stairway", and those without

\[
\binom{40}{10} = \binom{39}{9} + \binom{39}{10}
\]
From 5 choose 2

- Here is simpler problem, from 5 letters choose 2

\[
\begin{array}{cccc}
A & B & C & D \\
\end{array}
\quad \begin{array}{c}
\binom{5}{2} = \binom{4}{1} + \binom{4}{2}
\end{array}
\]

- All the sets with A (from 4 choose 1):
  - AB AC AD AE

- Those without A (from 4 choose 2, no A):
  - BC BD BE CD CE DE
Recursive $n$ choose $k$ (con.)

♦ Or, in general, we get

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]

♦ Rewritten as a method named $c$ that has two parameters $n$ and $k$:

\[
c(n, k) = c(n - 1, k - 1) + c(n - 1, k)
\]

♦ We're getting closer but where is the base case?
The base cases for \( n \) choose \( k \)

- First, \( n \) must be at least as big as \( k \):
  - We cannot choose a 10 song set from 9 songs
- When \( n = k \), there is only one choice
  - only one possible 10 song set from 10 songs
- To be meaningful, \( k \) must be at least 1
  - We're not interested in sets with 0 songs
- When \( k \) is 1, there are \( n \) ways to choose
  - If we only play 1 song sets, and we have 10 songs to choose from, we get \( n \), or 10, possible sets
Finally, here is the recursive definition of \( n \) choose \( k \)

The recursive definition of \( n \) choose \( k \) summarizes all of these points:

\[
c(n,k) = \begin{cases} 
k = 1 \Rightarrow n 
n = k \Rightarrow 1 
n > k \& \& k > 1 \Rightarrow c(n-1,k-1) + c(n-1,k) 
\end{cases}
\]

What is \( c(5, 2) \)? ____________
What is \( c(4, 1) \)? ____________
What is \( c(4, 2) \)? ____________
What is \( c(6, 3) \)? ____________
How many poker hands are possible?_____
We don’t need recursion

♦ Could also use the factorial method
  — at least for not large arguments

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]

♦ Example: From 4, choose 2

\[
\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{24}{2 \times 2} = 6
\]
How Recursion works

♦ Method calls generate activation records
  – Depending on the system, the activation record might store
    • all parameter values and local values
    • return point -- where to go after the method finishes
  – imagine all this is in a box
  – the data is stored in an activation frame (a box) and pushed onto a stack -- one on top of the other
  – when the method finishes, the activation frame stack is popped
    • it disappears, control returns to where it was called
A method that calls itself

```java
public void forward(int n){
    if(n > 1)
        forward(n - 1); // recursive call: n goes toward 0
    System.out.print(n);
}

@Test
public void showRecursion(){
    int arg = 3;
    forward(arg);
    // RP# SHOW
    arg = 999;
}
```

```
start in showRecursion
arg 3

arg 3

arg 3

arg 3

arg 3

arg 3

arg 3

arg 3
```
The base case is reached

♦ Several activation frames are stacked
♦ When parameter \((n) == 1\), there is no recursive call.

- (1) execute the base case when \(n == 1\)
  
  ```java
  System.out.print(n);
  ```
- The output is 1 and the method is done

```
public void forward(int n){
    if(n > 1)
        forward(n - 1); // recursive call
    System.out.print(n);
}
```
Returning back to SHOW

— (2) Return to previous call and pop box on top
  • continue from RP# FORWARD, print 2
— (3) Return to previous call and pop box on top
  • continue from RP# FORWARD, print 3
— (4) Return to showRecursion, pop box on top
  • execute arg = 999 (to indicate we're back in SHOW)

The SHOW method is done, Output is **123**
Infinite Recursion

- A recursive method will keep calling itself until the base case is reached
  - there must be at least one base case
    - this could be to do nothing
  - each call should make progress towards a base case
    - call a simpler version
      - could be a smaller or larger argument than the parameter
      - could be a smaller portion of an array or linked structure

- How many method calls are in the next code?
Infinite Recursion

```java
assertEquals(2, (factRec(2))); // Stopped when n == 10301 on one machine

public long factRec(int n) {
    System.out.println(n);
    if (n == 0) {
        return 1; // base case
    } else {
        return n * factRec(n + 1);
    }
}

java.lang.StackOverflowError
```
Refactorings

♦ Refactoring: Making small changes to code without changing the meaning

♦ Two related refactorings from Martin Fowler's online refactoring catalog
  – Replace recursion with iteration  
  – Replace iteration with recursion  