Chapter 6+6: Recursion

These are the slides of the textbook, modified/corrected where necessary. The chapters have been re-enumerated according to CMPE160.

Chapter outline

- thinking recursively
  - recursive algorithms
  - iteration vs. recursion
- recursive functions
  - integer exponentiation (pow)
  - infinite recursion
  - tracing recursive methods
  - greatest common divisor (GCD)
- recursive graphics (optional)

Recursive thinking

- **recursion**: a programming technique in which a method can call itself to solve a problem
- **recursive definition**: one which uses the word or concept being defined in the definition itself
  - In some situations, a recursive definition can be an appropriate or elegant way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively

reading: 12.1 - 12.2
Recursive definitions

Consider the following list of numbers:
24 \rightarrow 88 \rightarrow 40 \rightarrow 37

A list can be defined recursively

Either \hspace{1cm} LIST = null
or \hspace{1cm} LIST = element \rightarrow LIST

That is, a LIST is defined to be either empty (null), or an element followed by a LIST

(The concept of a LIST is used to define itself)

How would we confirm that null is a LIST? That one element is a LIST? That three elements are a LIST?

More recursive definitions

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

Note: The term arithmetic expression is defined by using the term arithmetic expression!
- (not the first bullet)

Recursive algorithms

**recursive algorithm**: description for a way to solve a problem, that refers to itself

Show everything in a folder and all it subfolders:
1. show everything in top folder
2. show everything in each subfolder in the same manner

Look up a word in a dictionary:
1. look up the word using the alphabetical ordering.
2. if all words in the definition are known to you, stop.
3. else, for each unknown word in the definition, look up that word

A recursive algorithm

Consider the task of finding out what place you are in a long line of people.

- If you cannot see the front of the line, you could ask the person in front of you.
- To answer your question, this person could ask the person in front of him/her, and so on.
A recursive algorithm

- Once the front person answers their place in line (first), this information is handed back, one person at a time, until it reaches you.
- This is the essence of recursive algorithms; many invocations of the same method each solve a small part of a large problem.

Recursive programming

- A method in Java can call itself; if written that way, it is called a recursive method.

- A recursive method solves some problem. The code of a recursive method should be written to handle the problem in one of two ways:
  - base case: a simple case of the problem that can be answered directly; does not use recursion.
  - recursive case: a more complicated case of the problem, that isn’t easy to answer directly, but can be expressed elegantly with recursion; makes a recursive call to help compute the overall answer.

Factorial example

- The factorial for any positive integer N, written N!, is defined to be the product of all integers between 1 and N inclusive.

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

// not recursive
public static long factorial(int n) {
    long product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}

Recursive factorial

- factorial can also be defined recursively:

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

- A factorial is defined in terms of another factorial until the basic case of 0! is reached.

// recursive
public static long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
Recursion vs. iteration

- every recursive solution has a corresponding iterative solution
  - For example, $N!$ can be calculated with a loop

- recursion has the overhead of multiple method invocations

- however, for some problems recursive solutions are often more simple and elegant than iterative solutions

- you must be able to determine when recursion is appropriate

Recursive power example

- Write method `pow` that takes integers `x` and `y` as parameters and returns $x^y$.
  - $x^y = x \times x \times x \times \ldots \times x$ (y times, in total)

- An iterative solution:

  ```java
  // not recursive
  public static int pow(int x, int y) {
      int product = 1;
      for (int i = 0; i < y; i++) {
          product = product * x;
      }
      return product;
  }
  ```

Recursive power function

- Another way to define the power function:
  - $\text{pow}(x, 0) = 1$
  - $\text{pow}(x, y) = x \times \text{pow}(x, y-1)$, $y > 0$

- A recursive solution:

  ```java
  // recursive
  public static int pow(int x, int y) {
      if (y == 0) {
          return 1;
      } else {
          return x * pow(x, y - 1);
      }
  }
  ```
How recursion works

- each call sets up a new instance of all the parameters and the local variables
- as always, when the method completes, control returns to the method that invoked it (which might be another invocation of the same method)

\[
pow(4, 3) = 4 \times pow(4, 2) \\
= 4 \times 4 \times pow(4, 1) \\
= 4 \times 4 \times 4 \times pow(4, 0) \\
= 4 \times 4 \times 4 \times 1 \\
= 64
\]

Infinite recursion

- a definition with a missing or badly written base case causes **infinite recursion**, similar to an infinite loop
- avoided by making sure that the recursive call gets closer to the solution (moving toward the base case)

```
public static int pow(int x, int y) {
    return x * pow(x, y - 1);  // Oops! Forgot base case
}
```

pow(4, 3) = 4 \times pow(4, 2) \\
= 4 \times 4 \times pow(4, 1) \\
= 4 \times 4 \times 4 \times pow(4, 0) \\
= 4 \times 4 \times 4 \times 4 \times pow(4, -1) \\
= 4 \times 4 \times 4 \times 4 \times 4 \times pow(4, -2) \\
= ... crashes: Stack Overflow Error!

Activation records

- **activation record**: memory that Java allocates to store information about each running method
- return point ("RP"), argument values, local variable values
- Java stacks up the records as methods are called; a method's activation record exists until it returns
- drawing the act. records helps us trace the behavior of a recursive method

```
<table>
<thead>
<tr>
<th>x = [ 4 ]   y = [ 0 ]</th>
<th>pow(4, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 1 ]</td>
<td></td>
</tr>
<tr>
<td>[ 2 ]</td>
<td></td>
</tr>
<tr>
<td>[ 3 ]</td>
<td></td>
</tr>
<tr>
<td>[ main ]</td>
<td></td>
</tr>
</tbody>
</table>
```

Tracing recursive methods

Consider the following method:

```
public static int mystery1(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return mystery1(x - y, y); 
    }
}
```

For each call below, indicate what value is returned:

mystery1(6, 13)  ____________
mystery1(14, 10)  ____________
mystery1(37, 10)  ____________
mystery1(8, 2)  ____________
mystery1(50, 7)  ____________
Tracing recursive methods

```java
public static void mystery2(int n) {
    if (n <= 1) {
        System.out.print(n);
    } else {
        mystery2(n / 2);
        System.out.print(", " + n);
    }
}
```

For each call below, indicate what output is printed:
- `mystery2(1)` ____________
- `mystery2(2)` ____________
- `mystery2(3)` ____________
- `mystery2(4)` ____________
- `mystery2(16)` ____________
- `mystery2(30)` ____________
- `mystery2(100)` ____________

Tracing recursive methods

```java
public static int mystery3(int n) {
    if (n < 0) {
        return -mystery3(-n);
    } else if (n < 10) {
        return n;
    } else {
        return mystery3(n/10 + n % 10);
    }
}
```

For each call below, indicate what value is returned:
- `mystery3(6)` ____________
- `mystery3(17)` ____________
- `mystery3(259)` ____________
- `mystery3(977)` ____________
- `mystery3(-479)` ____________

Tracing recursive methods

```java
public static void mystery4(String s) {
    if (s.length() > 0) {
        System.out.print(s.charAt(0));
        if (s.length() % 2 == 0) {
            mystery4(s.substring(0, s.length() - 1));
        } else {
            mystery4(s.substring(1, s.length()));
        }
        System.out.print(s.charAt(s.length() - 1));
    }
}
```

For each call below, indicate what output is printed:
- `mystery4("")` ____________
- `mystery4("a")` ____________
- `mystery4("ab")` ____________
- `mystery4("bc")` ____________
- `mystery4("abcd")` ____________

Recursive numeric problems

- **Problem**: Given a decimal integer \( n \) and a base \( b \), print \( n \) in base \( b \).
  
  (Hint: consider the \(/\) and \(\%\) operators to divide \( n \).)

- **Problem**: Given integers \( a \) and \( b \) where \( a \geq b \), find their greatest common divisor ("GCD"), which is the largest number that is a factor of both \( a \) and \( b \). Use Euclid's formula, which states that:

  \[
  \text{GCD}(a, b) = \text{GCD}(b, a \mod b) \\
  \text{GCD}(a, 0) = a
  \]

  (Hint: What should the base case be?)
Recursive printing problem

- **Problem**: Write a method `starString` that takes an integer `n` as an argument and returns a string of stars (asterisks) $2^n$ long (i.e., 2 to the `n`th power). For example:

  - `starString(0)` should return "*" (because $2^0 == 1$)
  - `starString(1)` should return "**" (because $2^1 == 2$)
  - `starString(2)` should return "*****" (because $2^2 == 4$)
  - `starString(3)` should return "********" (because $2^3 == 8$)
  - `starString(4)` should return "****************" ($2^4 == 16$)

Recursive string problems

- **Problem**: Write a recursive method `isPalindrome` that takes a string and returns whether the string is the same forwards as backwards. (Hint: examine the end letters.)

- **Problem**: Write a recursive method `areAnagrams` that takes two strings `w1` and `w2` and returns whether they are anagrams of each other; that is, whether the letters of `w1` can be rearranged to form the word `w2`.

Recursion can perform badly

- The **Fibonacci numbers** are a sequence of numbers $F_0, F_1, ... F_n$ such that:
  - $F_0 = F_1 = 1$
  - $F_i = F_{i-1} + F_{i-2}$ for any $i > 1$

- **Problem**: Write a method `fib` that, when given an integer `i`, computes the `i`th Fibonacci number.

- Why might a recursive solution to this problem be a bad idea? (Let's write it...)
  - Can we fix it? If so, how?

Revisiting Fibonacci...

- Recursive Fibonacci was expensive because it made many, many recursive calls
  - `fibonacci(n)` recomputed `fibonacci(n-1 ... 1)` many times in finding its answer!

  - this is a common case of "overlapping subproblems" or "divide poorly and reconquer", where the subtasks handled by the recursion are redundant with each other and get recomputed
Fractal images

- **fractal**: A mathematically generated, self-similar image.
  - Created by B. Mandelbrot in 1975
  - Many can be drawn elegantly using recursive algorithms

Fractal levels

- A fractal can be drawn at many different *levels*.
  - Each level is another layer of self-similarity.
  - The larger figure is decomposed into smaller occurrences of the same figure.
  - The smaller figures can themselves be decomposed, and so on.
  - Let’s write a program to draw the fractal below, an image called the Sierpinski Triangle.

Fractal code

- We can write a recursive method to draw the triangle figure at a certain level.
  ```java
  public static void drawFigure(int level, Graphics g) {
    ...
  }
  ```
  - The recursive aspect is that `drawFigure` for a given level should call itself for other levels as part of its work.
  - But these smaller levels appear at different positions, so we should require the triangle’s 3 corner points as parameters.

  ```java
  public static void drawFigure(int level, Graphics g, Point p1, Point p2, Point p3) {
    ...
  }
  ```
As usual, we begin with the base case.

- The easiest figure to draw is a triangle at level 1.
- We can use the `fillPolygon` method of `Graphics g` to do this.

```java
public static void drawFigure(int level, Graphics g, Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        ...}
}
```

A key observation: The end points needed to draw the smaller triangles (p4, p5, p6) are the midpoints between the larger triangle's endpoints (p1, p2, p3).

- We can write a method to compute the midpoint between two `Point` objects.

```java
// returns the point halfway between p1 and p2
public static Point midpoint(Point p1, Point p2) {
    return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
}
```

Complete solution method

```java
public static void drawFigure(int level, Graphics g, Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        Point p4 = midpoint(p1, p2);
        Point p5 = midpoint(p2, p3);
        Point p6 = midpoint(p1, p3);

        // recurse on 3 triangular areas
        drawFigure(level - 1, g, p1, p4, p6);
        drawFigure(level - 1, g, p4, p2, p5);
        drawFigure(level - 1, g, p6, p5, p3);
    }
}
```