Recursion

Readings: RS Chapter 12

Recursion: The definition of an operation in terms of itself.
- Solving a problem using recursion depends on solving smaller occurrences of the same problem.

Recursive Programming: Writing methods that call themselves to solve problems recursively.
- An equally powerful substitute for iteration (loops)
- Particularly well-suited to solving certain types of problems
Why learn recursion?

- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)

The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?
Recursive algorithm

- Number of people behind me:
  - If there is someone behind me, ask him/her how many people are behind him/her.
    - When they respond with a value $N$, then I will answer $N + 1$.
  - If there is nobody behind me, I will answer $0$.

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **Base Case**: A simple occurrence that can be answered directly.
  - **Recursive Case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

- Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
- A crucial part of recursive programming is identifying these cases.
Recursion in Java

• Consider the following method to print a line of * characters:

```java
// Prints a line containing the given number of stars.
// Precondition: n >= 0
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();  // end the line of output
}
```

• Write a recursive version of this method (that calls itself).
  – Solve the problem without using any loops.

A basic case

• What are the cases to consider?
  – What is a very easy number of stars to print without a loop?

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```
Handling more cases

- Handling additional cases, with no loops (in a bad way):

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...;
}
```

Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1);    // prints "*
    } else if (n == 3) {
        System.out.print("*");
        printStars(2);    // prints "**
    } else if (n == 4) {
        System.out.print("*");
        printStars(3);    // prints "***"
    } else ...
}
```
Using recursion properly

- Condensing the recursive cases into a single case:

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

"Recursion Zen"

- The real, even simpler, base case is an n of 0, not 1:

```java
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

---

- **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.
• Consider the following recursive method:

```java
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

- What is the result of the following call?
  ```java
  mystery(648)
  ```

• Consider the following recursive method:

```java
public static int mystery2(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery2(n / 10);
        int b = mystery2(n % 10);
        return (100 * a) + b;
    }
}
```

- What is the result of the following call?
  ```java
  mystery2(348)
  ```
Exercise

• Write a recursive method \( \text{pow} \) accepts an integer base and exponent and returns the base raised to that exponent.
  - Example: \( \text{pow}(3, 4) \) returns 81

  - Solve the problem recursively and without using loops.

An optimization

• Notice the following mathematical property:

\[
3^{12} = 531441 = 9^6 = (3^2)^6 \\
531441 = (9^2)^3 = ((3^2)^2)^3
\]

  - When does this "trick" work?

  - How can we incorporate this optimization into our \( \text{pow} \) method?

  - What is the benefit of this trick if the method already works?
```java
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else if (exponent % 2 == 0) {  // recursive case 1:  x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {  // recursive case 2:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

**Verification of Recursion**

- How do we know a recursive method works?
  - Static analysis (static)
  - Testing (dynamic)
  - Formal verification (static)

- Formal Verification
  - Correctness proofs using principles learned in Discrete Math
    - Shows correctness of the algorithm on all valid inputs
  - What type of proof strategy is recursion similar to?
Verification of Recursion (2)

- When proving an operation implementation is correct, the verifier has:
  - Preconditions: assume true at the beginning of the method (initial state(s))
  - Post Conditions (Ensures/Goal): what you must prove to be true at the end of the method (final state(s))

- When proving a call of an operation is correct
  - Preconditions: prove that the input(s) to the operation is(are) correct before calling the operation
  - Post Conditions: assume operation’s post conditions are true after the call

Example

```java
int sum(int j, int k)
// Precondition: j >= 0
// Goal/Ensures: result = j + k
{
    if (j == 0) {
        return k;
    } else {
        j--;
        int r = sum(j, k);
        return r + 1;
    }
}
```

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project
Reasoning Pattern

- Similar to an inductive proof
- Base case (e.g., \( j == 0 \))
  - Reason code works for the base case
- Recursive case (e.g., \( j > 0 \))
  - Assume that the recursive call \( j--; r = \text{sum}(j, k) \) works
  - Reason that the code works for the case of \( j \)
  - Show assumption is legit, i.e., show termination

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project

Recursion: Base case

```c
int sum(int j, int k)
// Precondition: j >= 0
// Goal: result = j + k
{
    if (j == 0) {
        return k;
        // Given/Assume: (j = 0) ^ (result = k)
        // Confirm Goal: result = 0 + k
    } else {
        ...
    }
}
```

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project
Recursion: Inductive Assumption

```c
int sum(int j, int k)
// Precondition: j >= 0
// Goal: result = j + k
{
    if (j == 0) {
        ...
    } else {
        j--;
        int r = sum(j, k);
        // Given/Assume: r = (j – 1) + k
        return r + 1;
    }
}
```

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project

Recursion: Inductive Proof Step

```c
int sum(int j, int k)
// requires j >= 0
// ensures result = j + k
{
    if (j == 0) {
        ...
    } else {
        j--;
        int r = sum(j, k);
        return r + 1;
        // Given/Assume: (r = (j – 1) + k) ^
        // (result = r + 1)
        // Confirm Goal: result = j + k
    }
}
```

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project
Reasoning: Recursive Case

- For the inductive proof to be legit, the inductive assumption must be legit
- This requires showing that an argument passed to the recursive call is strictly smaller
- This is the proof of termination
- To prove termination automatically, programmers need to provide a progress metric (\( j \) decreases in the example)

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project

Reasoning about iterative code

- Also appeals to induction
- Loops need to include an “invariant” and a progress metric for termination
- Invariant is established for the base case (i.e., before the loop is entered)
- Invariant is assumed at the beginning of an iteration and confirmed at the beginning of the next
- Also needs a progress metric and a proof of termination

Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project
Exercise

• Write a recursive method `generateBinary` that accepts an integer and returns that number’s representation in binary (base 2).
  - Example: `generateBinary (7)` returns 111
  - Example: `generateBinary (12)` returns 1100
  - Example: `generateBinary (42)` returns 101010

<table>
<thead>
<tr>
<th>place</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>place</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• Write the method recursively and without using any loops.

Case analysis

• Recursion is about solving a small piece of a large problem.
  - What is 69743 in binary?
    • Do we know anything about its representation in binary?

  - Case analysis:
    • What is/are easy numbers to print in binary?
    • Can we express a larger number in terms of a smaller number(s)?

  - Suppose we are examining some arbitrary integer N.
    • if N's binary representation is 10010101011
    • (N / 2) 's binary representation is 1001010101
    • (N % 2) 's binary representation is 1
Basic Math Strings

- Unlike sets, strings have order
  - Example: Str(Z) for String of integers

- Notations
  - Empty string (written empty_string or Λ)
  - Concatenation ( alpha o beta )
  - Length ( |alpha| )
  - String containing one entry ( <5> )

```java
// Prints the given integer's binary representation.
// Precondition: n >= 0
public static String generateBinary(int n) {
    if (n < 2) {
        // base case; same as base 10
        return "" + n;
    }
    else {
        // recursive case; break number apart
        return generateBinary(n / 2) +
               generateBinary(n % 2);
    }
}
```
Exercise

• Write a recursive method `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.
  - `isPalindrome("madam")` → true
  - `isPalindrome("racecar")` → true
  - `isPalindrome("step on no pets")` → true
  - `isPalindrome("able was I ere I saw elba")` → true
  - `isPalindrome("Java")` → false
  - `isPalindrome("rotater")` → false
  - `isPalindrome("byebye")` → false
  - `isPalindrome("notion")` → false

Exercise

• Write a method `crawl` accepts a `File` parameter and prints information about that file.
  - If the `File` object represents a normal file, just print its name.
  - If the `File` object represents a directory, print its name and information about every file/directory inside it, indented.

```java
// cse143
handouts
  syllabus.doc
  lecture_schedule.xls
homework =
  l-sortedintlist
    ArrayIntList.java
    SortedIntList.java
  index.html
  style.css
```

- **recursive data:** A directory can contain other directories.
File objects

A File object (from the java.io package) represents a file or directory on the disk.

<table>
<thead>
<tr>
<th>Constructor/method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File(String)</td>
<td>creates File object representing file with given name</td>
</tr>
<tr>
<td>canRead()</td>
<td>returns whether file is able to be read</td>
</tr>
<tr>
<td>delete()</td>
<td>removes file from disk</td>
</tr>
<tr>
<td>exists()</td>
<td>whether this file exists on disk</td>
</tr>
<tr>
<td>getName()</td>
<td>returns file's name</td>
</tr>
<tr>
<td>isDirectory()</td>
<td>returns whether this object represents a directory</td>
</tr>
<tr>
<td>length()</td>
<td>returns number of bytes in file</td>
</tr>
<tr>
<td>listFiles()</td>
<td>returns a File[] representing files in this directory</td>
</tr>
<tr>
<td>renameTo(File)</td>
<td>changes name of file</td>
</tr>
</tbody>
</table>

Public/private pairs

We cannot vary the indentation without an extra parameter:

```java
public static void crawl(File f, String indent) {
```

Often the parameters we need for our recursion do not match those the client will want to pass.

In these cases, we instead write a pair of methods:

1) a public, non-recursive one with the parameters the client wants
2) a private, recursive one with the parameters we really need