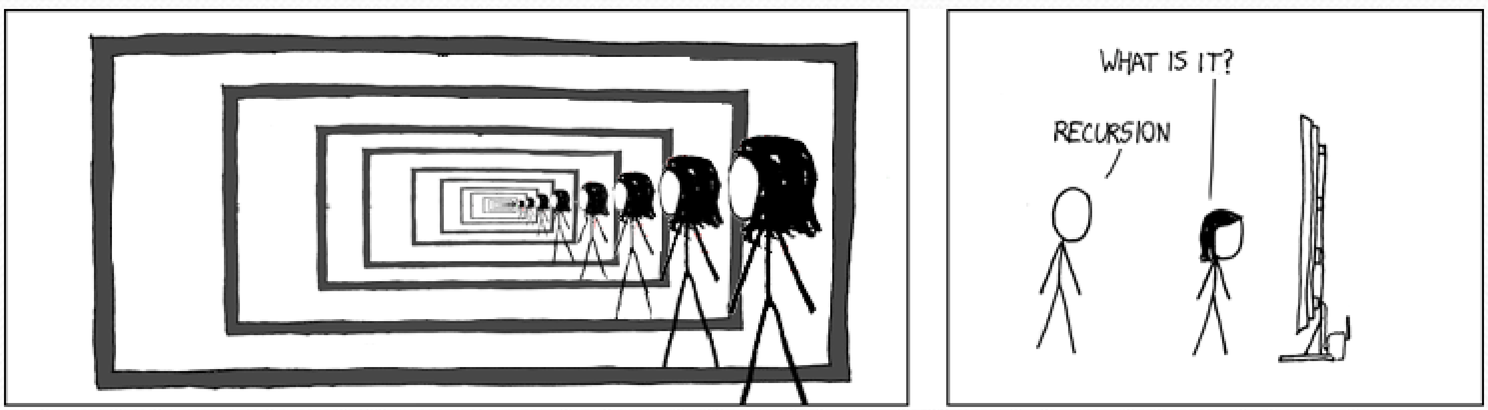
CSE 143

Lecture 9: introduction to recursion

reading: 12.1



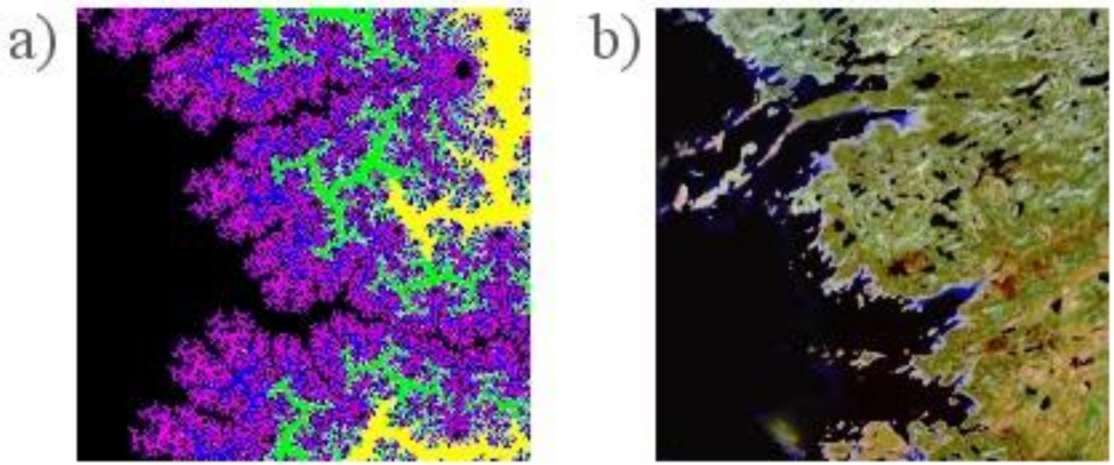
Recursion

- **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


ALOE
POLYPHYLLA
SPIRAL ALOE



a) Part of the Mandelbrot set.

b) Part of the North American coastline near Hudson Bay.

Getting down stairs

- Need to know two things:
  - Getting down one stair
  - Recognizing the bottom

- Most code will look like:

```
if (simplest case) {
    compute and return solution
} else {
    divide into similar subproblem(s)
    solve each subproblem recursively
    assemble the overall solution
}
```

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:

```
// 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.
  - Hint: Your solution should print just one star at a time.

"Recursion Zen"

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

```
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.

  (A CSE 143 informal term)

Exercise

- Write a recursive method `reverseLines` that accepts a file `Scanner` and prints the lines of the file in reverse order.
  - Example input file:

```
I have eaten
the plums
that were in
the icebox
```

→ Expected console output:

```
the icebox
that were in
the plums
I have eaten
```

  - What are the cases to consider?
    - How can we solve a small part of the problem at a time?
    - What is a file that is very easy to reverse?

Tracing our algorithm

- **call stack**: The method invocations currently running

```
reverseLines(new Scanner("poem.txt"));
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "I have eaten"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "the plums"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "that were in"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "the icebox"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {   // false
        ...
    }
}
```

```
I have eaten
the plums
that were in
the icebox
```

```
the icebox
that were in
the plums
I have eaten
```

Recursive tracing

- Consider the following recursive method:

```
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?

    `mystery(648)`

A recursive trace

```
mystery(648):
▪ int a = 648 / 10;        // 64
▪ int b = 648 % 10;        //  8
▪ return mystery(a + b);   // mystery(72)

    mystery(72):
    ▪ int a = 72 / 10;         // 7
    ▪ int b = 72 % 10;         // 2
    ▪ return mystery(a + b);   // mystery(9)

        mystery(9):
        ▪ return 9;
```

Recursive tracing 2

- Consider the following recursive method:

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

  - What is the result of the following call?

  `mystery(348)`

A recursive trace 2

```
mystery(348)
  int a = mystery(34);
    int a = mystery(3);
      return (10 * 3) + 3;     // 33
    int b = mystery(4);
      return (10 * 4) + 4;     // 44
    return (100 * 33) + 44;    // 3344

  int b = mystery(8);
    return (10 * 8) + 8;       // 88

  return (100 * 3344) + 88;    // 334488
```

- What is this method really doing?