2023-03-20

Recursive Algorithms

Base Case, Recursive Case

Base Case	Recursive Case
The **base case** of a recursive method is the case where it does not recursively call itself, that is, the method terminates. The base case is a problem that is so simple, we already know the answer to it!	The **recursive case**, or **general case**, is the case where the method calls itself. It's called the general case because it's the case that usually happens when a recursive algorithm is executing. For the algorithm to work, the recursive case must diminish the problem so that it eventually approaches the base case.

Factorial

The factorial n! of a non-negative integer n is the product of all of the integers 1..n multiplied together.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 1$$

By convention, $0! = 1$

One of the uses of factorial: It equals the number of permutations of n items, that is, the number of ways they can be ordered.

If you have items A, B and C, there are
3! = 3 • 2 • 1 = 6 permutations.

A B C

A C B

B A C

B C A

C A B

C B A

Recurrence Relations

Many mathematical functions can be defined in terms of **recurrence relations**.

A recurrence relation defines a mathematical function in terms of a smaller version of itself.

Factorial

1, 2, 6, 24, 120, 720, ...

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F(0) = 0, F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Factorial

With recursion, a recurrence relation can be translated pretty directly into code.

```
int factorial(int n) {
    if (n == 0) {
        // Base case: 0! = 1
        return 1;
    } else {
        // General case: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}
```

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Head Recursion vs. Tail Recursion

Head Recursion	Tail Recursion
If a method makes a recursive call to itself, but then does anything after that point, it is **head recursive**.	If the recursive call is the last thing the method does, then the method is said to be **tail recursive**. Some languages (but not Java) can optimize out tail recursion automatically, turning it into a loop. This makes the method more efficient, and eliminates the risk of a stack overflow.

Factorial head recursive, tail recursive

Our previous factorial example is head recursive. To make it tail recursive, we can introduce an accumulator that is passed along.

```
int factorial(int n) {

  if (n == 0) {

    // Base case: 0! = 1

    return 1;

  } else {

    // General case: n! = n * (n-1)!

    return n * factorial(n - 1);

  }

}
```

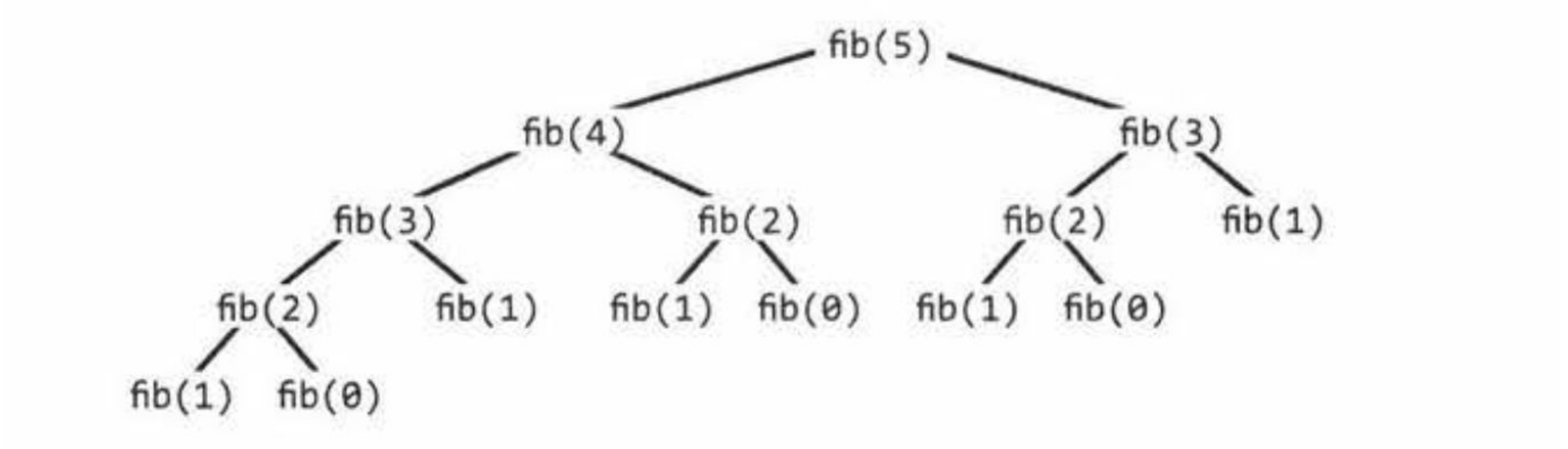
```
int factorialHelper(int n, int a) {

  if (n <= 1) {

    return a;

  } else {

    return factorialHelper(n - 1, n * a);

  }

}

int factorial(int n) {

  return factorialHelper(n, 1);

}
```

Fibonacci

```
int fibonacci(int n) {
    if (n == 0 || n == 1) {
        // Base cases: fibo(0) = 0, fibo(1) = 1
        return n;
    } else {
        // General case: fibo(n) = fibo(n-1) + fibo(n-2)
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Efficiency of recursive Fibonacci: $O(2^N)$ ... very poor!

While this formulation of Fibonacci is elegant, it's inefficient because the same Fibonacci numbers are calculated again and again.

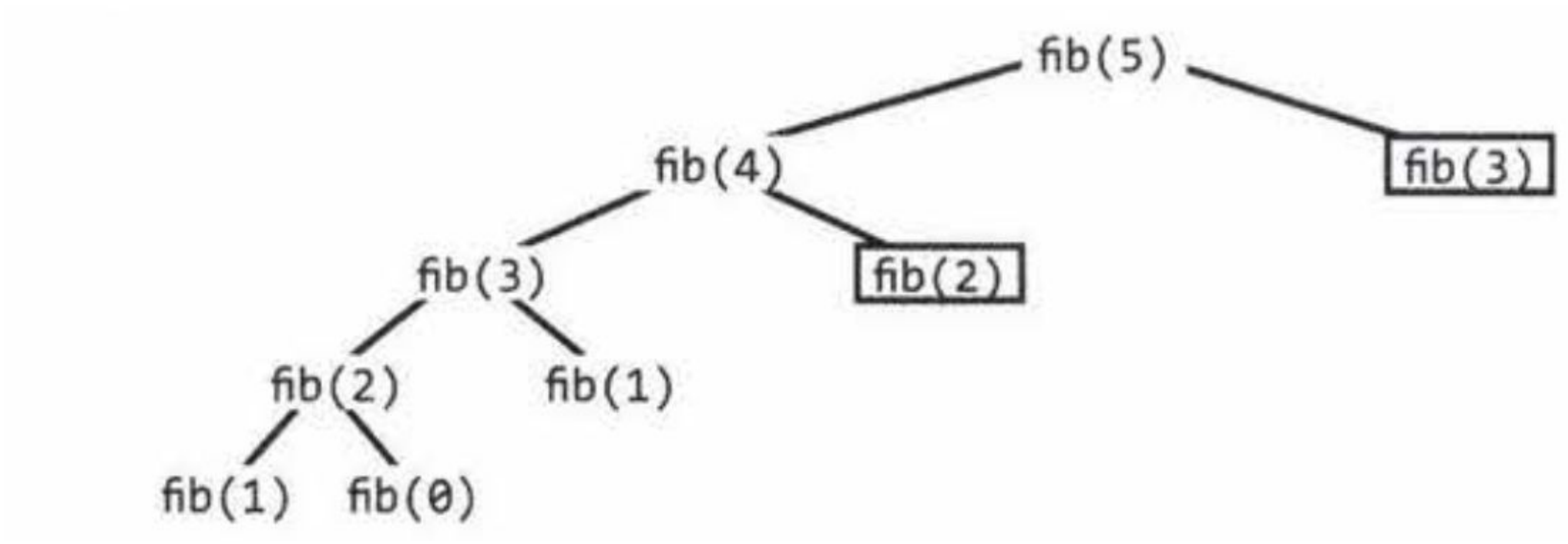


Memo-ization

```
private static long fibonacciCache[] = new long[1000];

// Precondition: n >= 0 && n < 1000

public long fibonacci(long n) {

  if (n == 0 || n == 1) {

    // Base cases: fibo(0) = 0, fibo(1) = 1

    return n;

  } else if (fibonacciCache[n] != 0) {

    return fibonacciCache[n];

  } else {

    // General case: fibo(n) = fibo(n-1) + fibo(n-2)

    return fibonacciCache[n] = fibonacci(n-1) + fibonacci(n-2);

  }

}
```

Efficiency of Fibonacci with memo-ization

The memo-ization brings the running time down to O(N), as we never calculate the same Fibonacci number more than once.



When to use recursion?

Factorial and Fibonacci can be expressed naturally using recursion, but they're not necessarily more efficient that way.

There are, however, many algorithms that are easier to write and understand in a recursive fashion.

Walking directory trees

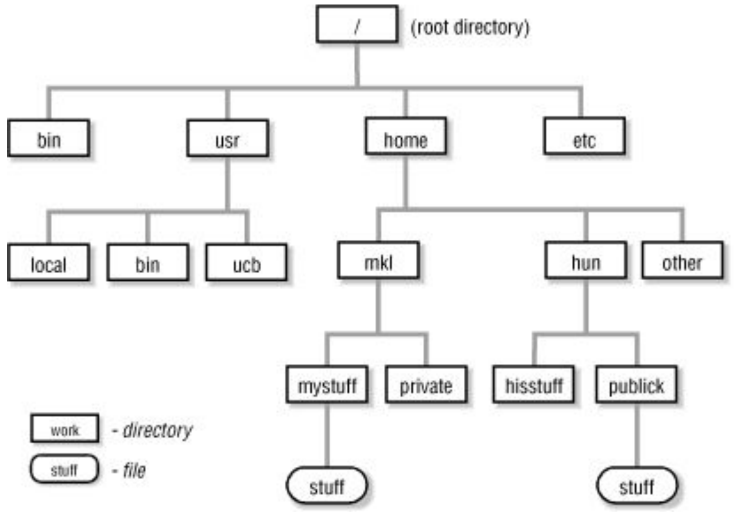
Subdirectories on your hard drive are recursive, since subdirectories nest inside each other.

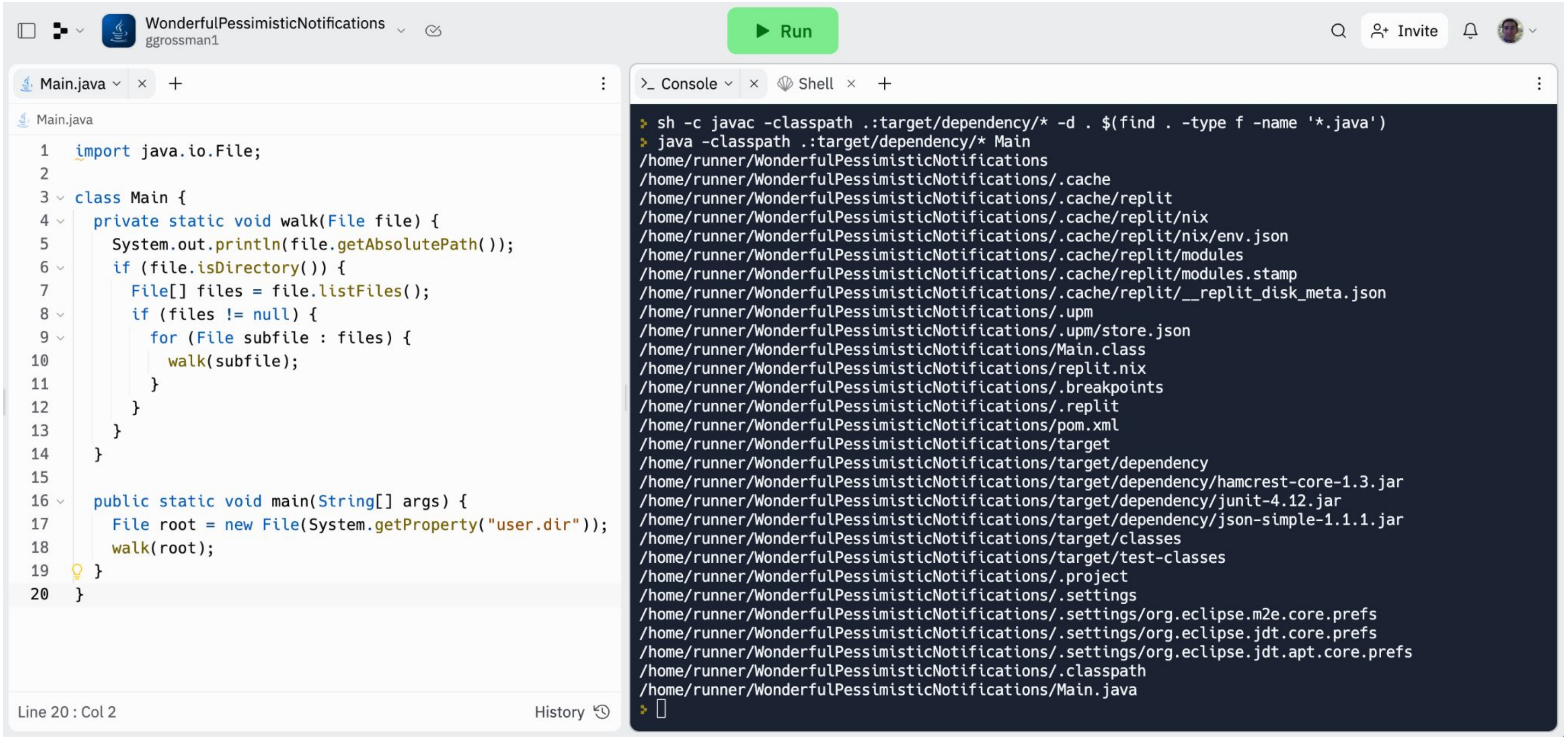
Subdirectories are what's called a **tree structure**.

Going through all of the directories recursively until you hit the ends is called **walking** the directory tree.

```
def walk(directory):
  for all files in directory:
    if this file is a subdirectory, walk(file)
```

This algorithm is also known as **depth-first search** and is used for many things other than directory trees, such as solving mazes.





```java
import java.io.File;

class Main {
  private static void walk(File file) {
    System.out.println(file.getAbsolutePath());
    if (file.isDirectory()) {
      File[] files = file.listFiles();
      if (files != null) {
        for (File subfile : files) {
          walk(subfile);
        }
      }
    }
  }

  public static void main(String[] args) {
    File root = new File(System.getProperty("user.dir"));
    walk(root);
  }
}
```

```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath .:target/dependency/* Main
/home/runner/WonderfulPessimisticNotifications
/home/runner/WonderfulPessimisticNotifications/.cache
/home/runner/WonderfulPessimisticNotifications/.cache/replit
/home/runner/WonderfulPessimisticNotifications/.cache/replit/nix
/home/runner/WonderfulPessimisticNotifications/.cache/replit/nix/env.json
/home/runner/WonderfulPessimisticNotifications/.cache/replit/modules
/home/runner/WonderfulPessimisticNotifications/.cache/replit/modules.stamp
/home/runner/WonderfulPessimisticNotifications/.cache/replit/__replit_disk_meta.json
/home/runner/WonderfulPessimisticNotifications/.upm
/home/runner/WonderfulPessimisticNotifications/.upm/store.json
/home/runner/WonderfulPessimisticNotifications/Main.class
/home/runner/WonderfulPessimisticNotifications/replit.nix
/home/runner/WonderfulPessimisticNotifications/.breakpoints
/home/runner/WonderfulPessimisticNotifications/.replit
/home/runner/WonderfulPessimisticNotifications/pom.xml
/home/runner/WonderfulPessimisticNotifications/target
/home/runner/WonderfulPessimisticNotifications/target/dependency
/home/runner/WonderfulPessimisticNotifications/target/dependency/hamcrest-core-1.3.jar
/home/runner/WonderfulPessimisticNotifications/target/dependency/junit-4.12.jar
/home/runner/WonderfulPessimisticNotifications/target/dependency/json-simple-1.1.1.jar
/home/runner/WonderfulPessimisticNotifications/target/classes
/home/runner/WonderfulPessimisticNotifications/target/test-classes
/home/runner/WonderfulPessimisticNotifications/.project
/home/runner/WonderfulPessimisticNotifications/.settings
/home/runner/WonderfulPessimisticNotifications/.settings/org.eclipse.m2e.core.prefs
/home/runner/WonderfulPessimisticNotifications/.settings/org.eclipse.jdt.core.prefs
/home/runner/WonderfulPessimisticNotifications/.settings/org.eclipse.jdt.apt.core.prefs
/home/runner/WonderfulPessimisticNotifications/.classpath
/home/runner/WonderfulPessimisticNotifications/Main.java
>
```

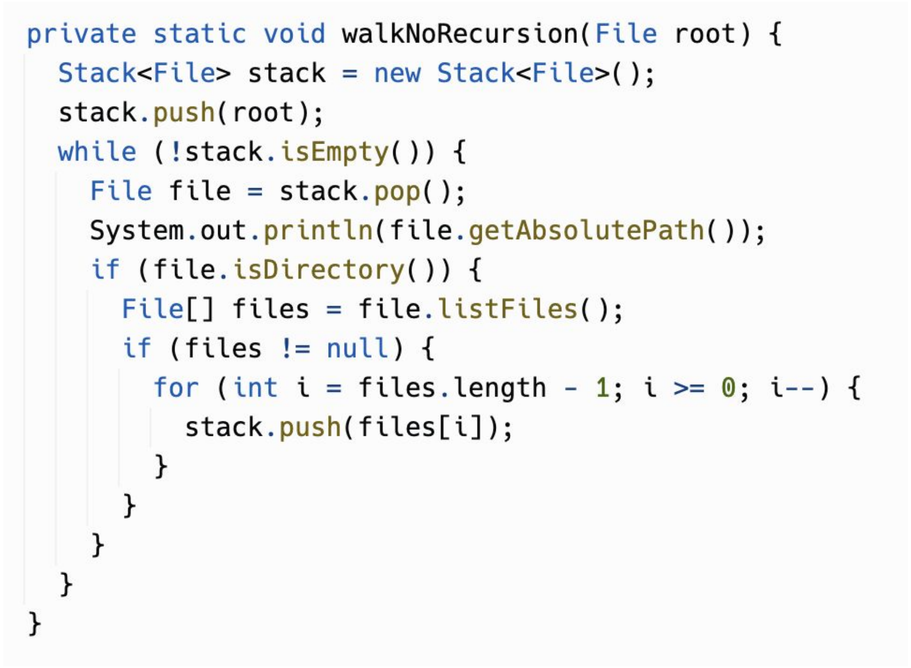
Here's how you could walk the directory tree without using recursion.

Recursion lets you take advantage of the built-in stack used to track method calls.

If you don't use recursion here, you need to provide your own stack!

Using your own stack can make sense if the recursion would get too deep and cause a stack overflow.

Here, recursion is more natural and filesystem directories are usually not thousands of levels deep... if they are, we call that **pathologically** deep. People occasionally do something like that as a joke, or if they're trying to hack something.



```java
private static void walkNoRecursion(File root) {
  Stack<File> stack = new Stack<File>();
  stack.push(root);
  while (!stack.isEmpty()) {
    File file = stack.pop();
    System.out.println(file.getAbsolutePath());
    if (file.isDirectory()) {
      File[] files = file.listFiles();
      if (files != null) {
        for (int i = files.length - 1; i >= 0; i--) {
          stack.push(files[i]);
        }
      }
    }
  }
}
```

Java itself is recursive

Most programming languages are recursive. Their building blocks are expressions and statements, and expressions can usually contain nested sub-expressions, and statements contain nested statements.

Each set of parentheses is a sub-expression which is evaluated recursively:

```
Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
```

if statements and other statements can nest within if statements, because the then and else clause are "statement":

```
if (x == 1) {
  if (y == 2) {
    ...
  }
}
```

Simplifying somewhat, but somewhere in a Java compiler, there is a method like "parseStatement" that calls itself!

Java and recursion

Java is an **imperative** language: Java statements are commands to the computer, executed in order. These commands often update the running state of the program (variables, etc.)

Java code tends to primarily use iteration. Recursion is used when it’s the best fit for a particular algorithm. (Like walking a directory tree.)

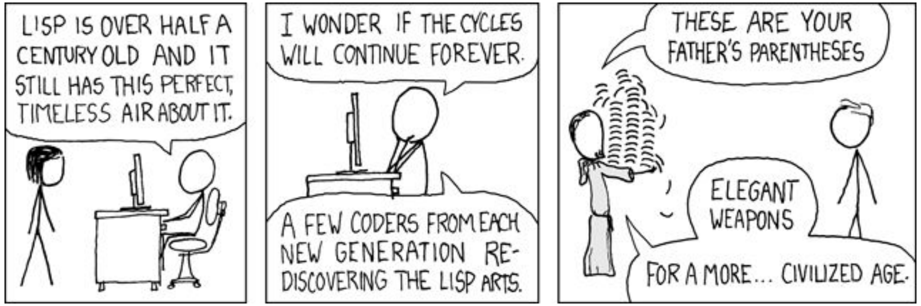
Other imperative languages: C/C++, C#

Functional programming

Another programming paradigm is **functional programming.** In FP languages (Lisp, Haskell, Scheme, Clojure, Scala), recursion is often preferred to iteration and more in the spirit of the language.

In functional programming, a program is applying and composing functions, ideally "pure" functions that have no *side effects* like changing variables. Calling yourself with new values is seen as better than mutating variables!

```
(defun factorial (n)
  (if (< n 2)
      1
      (* n (factorial (1- n)))))
```


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

Limits of recursion in Java

Some programming languages, particularly ones where recursion is fundamental, have a feature called Tail Call Optimization. TCO automatically converts tail recursion into iteration.

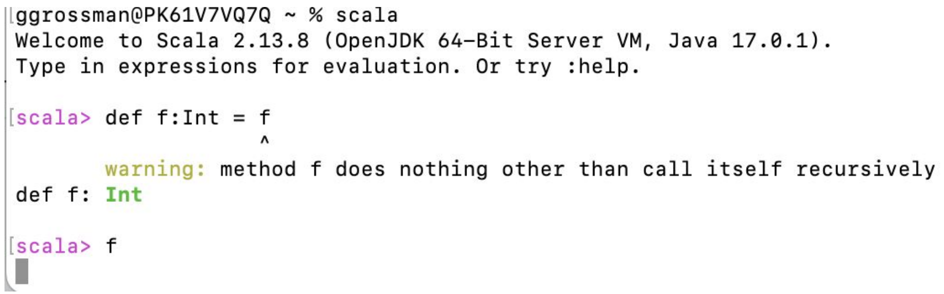
Java doesn't have this feature, although there is talk of adding it.

Because Java lacks TCO, even a tail-recursive method will blow up the stack if it goes through enough levels.

1MB stack is typical. If your recursive method uses a 100 byte stack frame, it will crash with StackOverflowError after 1m/100 = ~10,000 recursions.

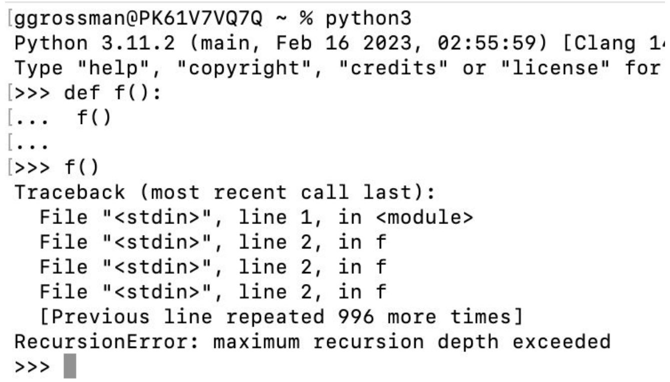
Limits of recursion in Java

Scala has TCO, hangs forever



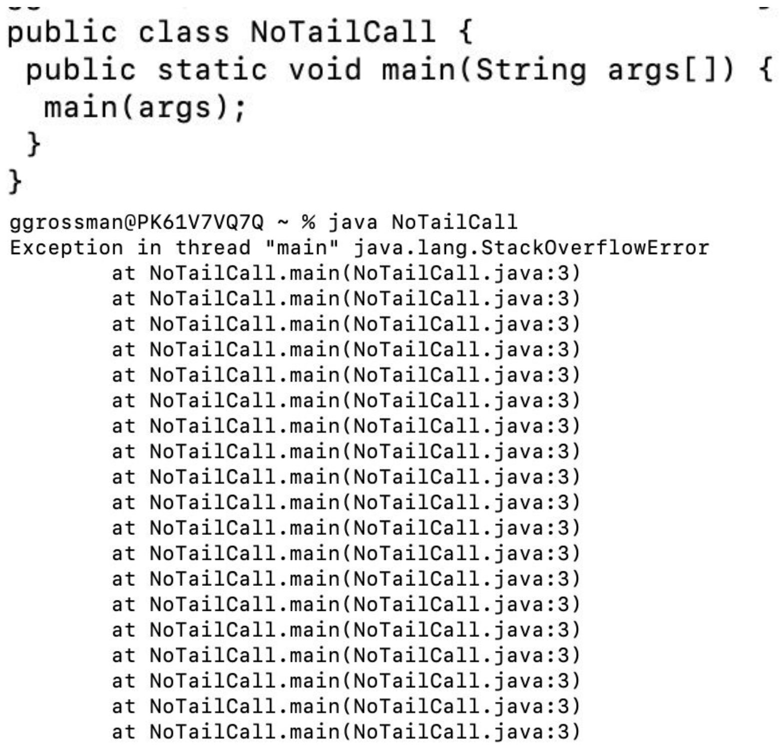
```
[ggrossman@PK61V7VQ7Q ~ % scala
Welcome to Scala 2.13.8 (OpenJDK 64-Bit Server VM, Java 17.0.1).
Type in expressions for evaluation. Or try :help.

[scala> def f:Int = f
                    ^
       warning: method f does nothing other than call itself recursively
def f: Int

[scala> f
█
```

Python: No TCO, stack overflow



```
[ggrossman@PK61V7VQ7Q ~ % python3
Python 3.11.2 (main, Feb 16 2023, 02:55:59) [Clang 1
Type "help", "copyright", "credits" or "license" for
[>>> def f():
[...   f()
[...
[>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in f
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
>>> █
```

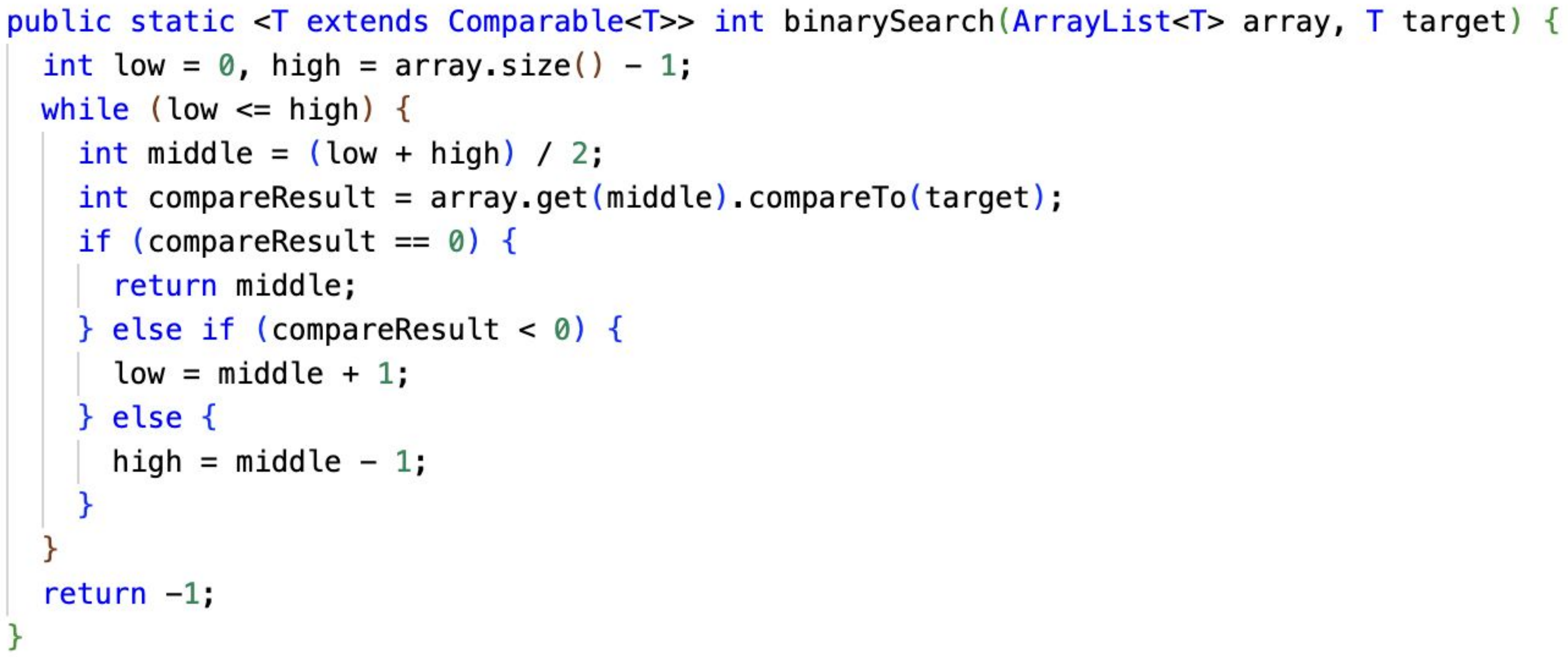
Java has no TCO, stack overflow



```
public class NoTailCall {
 public static void main(String args[]) {
  main(args);
 }
}
```

```
ggrossman@PK61V7VQ7Q ~ % java NoTailCall
Exception in thread "main" java.lang.StackOverflowError
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
        at NoTailCall.main(NoTailCall.java:3)
```

Review: Binary Search (iterative approach)



```java
public static <T extends Comparable<T>> int binarySearch(ArrayList<T> array, T target) {
  int low = 0, high = array.size() - 1;
  while (low <= high) {
    int middle = (low + high) / 2;
    int compareResult = array.get(middle).compareTo(target);
    if (compareResult == 0) {
      return middle;
    } else if (compareResult < 0) {
      low = middle + 1;
    } else {
      high = middle - 1;
    }
  }
  return -1;
}
```

Recursive Binary Search

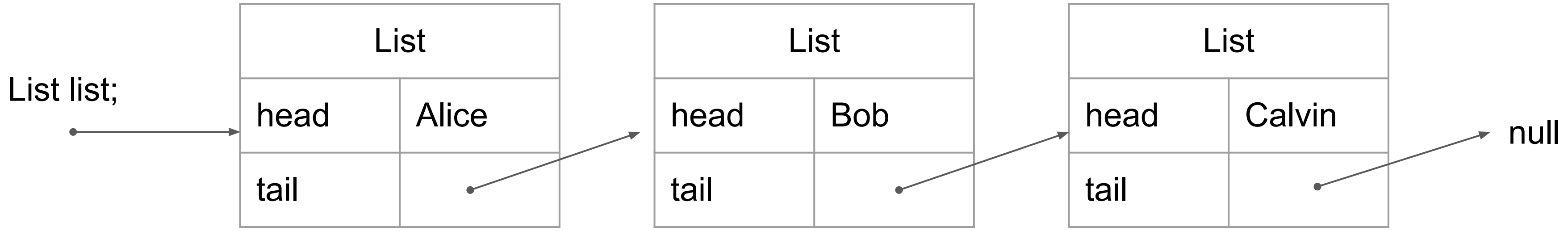
```java
public static <T extends Comparable<T>> int binarySearchHelper(ArrayList<T> array, T target, int low, int high) {
  if (low > high) {
    return -1;
  }
  int middle = (low + high) / 2;
  int compareResult = array.get(middle).compareTo(target);
  if (compareResult == 0) {
    return middle;
  } else if (compareResult < 0) {
    return binarySearchHelper(array, target, middle+1, high);
  } else {
    return binarySearchHelper(array, target, low, middle-1);
  }
}

public static <T extends Comparable<T>> int recursiveBinarySearch(ArrayList<T> words, T target) {
  return binarySearchHelper(words, target, 0, words.size() - 1);
}
```

Binary Search: Iterative or Recursive?

- The recursive solution is ever-so-slightly slower due to method call overhead.
- However, you may find it easier to understand... or harder to understand! Some programmers love recursion, some don't.
- A binary search is unlikely to get deep enough to cause a stack overflow, so it's not a problem to use the recursive method.
- If you go into the source code for the Java standard library implementation of binary search, it is probably iterative.
- The iterative version of binary search is essentially the recursive version with tail call optimization manually applied!

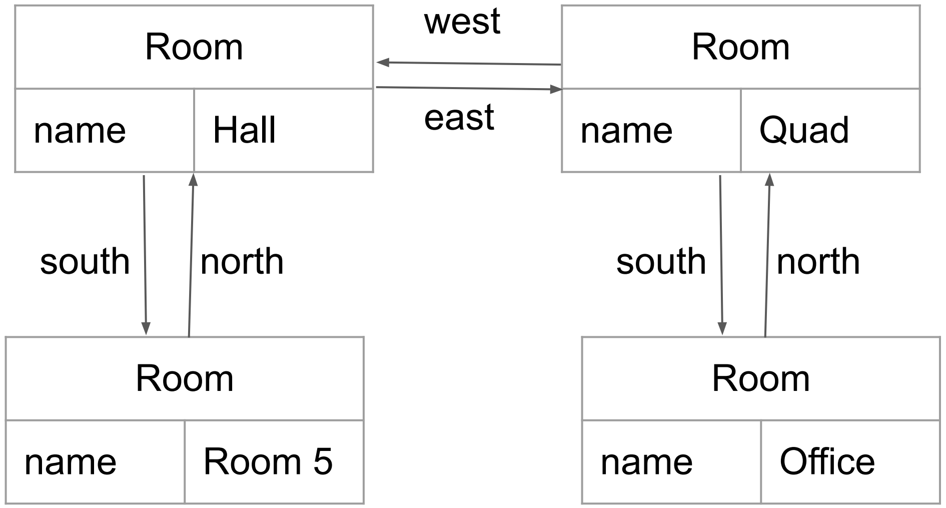
Linked List - a recursive data structure

```
class List<T extends Comparable<T>> {
  // The object being stored in this node of the linked list
  private T head;

  // A reference to a linked list which is the continuation of this one
  private List<T> tail;

  ...
}
```



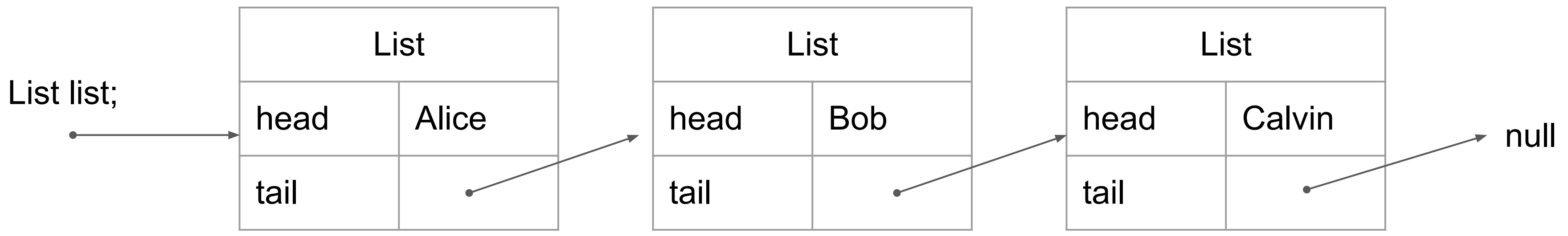
Linked List - a recursive data structure

You've already dealt with a recursive data structure similar to a Linked List: The Room class in the Adventure project. The map in the Adventure project is an example of a data structure called a **directed graph**, a set of vertices connected by directed edges.



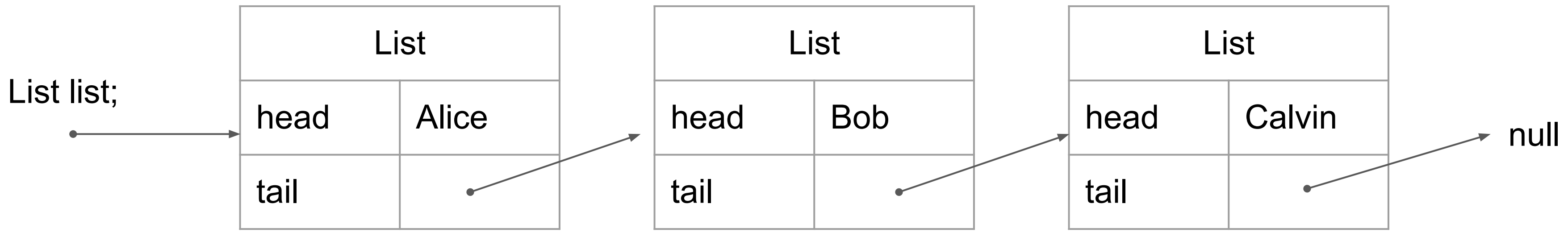
Linked List - Why not just use an array?

- Often, you do just use an array or ArrayList, but sometimes a linked list is the best fit.
- Linked lists have O(1) insertion and deletion time and never need resizing.
- Linked lists do NOT support random access... to get to the n'th element, you have to walk the list from the beginning.
- Even finding out the size of a linked list requires traversing the whole thing.



Linked List - Repl.it

- You'll be implementing three methods. These **could** be done in iterative style, but try to write them recursively!
  - Count number of elements in the linked list
  - Return whether a particular value is in the linked list
  - Prepend (add to the beginning) a value to the linked list
- Note that methods are implemented as static methods of List, since **null** is empty list.


List list;
List
head
Alice
tail
List
head
Bob
tail
List
head
Calvin
tail
null