Recursion

Definitions I

- A *recursive definition* is a definition in which the thing being defined occurs as part of its own definition
- Example:
  - An atom is a name or a number
  - A list consists of:
    - An open parenthesis, "("
    - Zero or more atoms or lists, and
    - A close parenthesis, ")“
    - → (atom1, atom2, list1, atom3)

Definitions II

- *Indirect recursion* is when a thing is defined in terms of other things, but those other things are defined in terms of the first thing
- Example: A list is:
  - An open parenthesis,
  - Zero or more S-expressions, and
  - A close parenthesis
- An S-expression is an atom or a list

Understand Recursion

- A child couldn't sleep, so her mother told a story about a little frog,
  - who couldn't sleep, so the frog's mother told a story about a little bear,
    - who couldn't sleep, so bear's mother told a story about a little weasel
      - ...who fell asleep.
    - ...and the little bear fell asleep;
  - ...and the little frog fell asleep;
- ...and the child fell asleep.

Recursive functions/methods

- The mathematical definition of factorial is:

$$\text{factorial}(n) \text{ is } \begin{cases} 1, \text{ if } n <= 1 \\ n * \text{factorial}(n-1) \text{ otherwise} \end{cases}$$

- We can define this in Java as:
  - ```
    long factorial(long n) {
        if (n <= 1) return 1;
        else return n * factorial(n – 1);
    }
    ```
- This is a recursive function because it calls itself
- Recursive functions are completely legal in Java
    ```

Anatomy of a recursion

Base case: does some work without making a recursive call

```
long factorial(long n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}
```

Extra work to convert the result of the recursive call into the result of *this* call

Recursive case: recurs with a simpler parameter

Example 1

- The following fills an array with the numbers 0 through n-1
- ```
  void run() {
      int[ ] a = new int[10];
      fill(a, a.length - 1);
      System.out.println(Arrays.toString(a));
  }
  ```
- ```
  void fill(int[ ] a, int n) {
      if (n < 0) return;
      else {
          a[n] = n;
          fill(a, n - 1);
      }
  }
  ```
- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Anatomy of a Recursive Function

```
void fill(int[ ] a, int n) {
    if (n < 0) return;
    else {
        a[n] = n;
        fill(a, n - 1);
    }
}
```

Base case: does some work without making a recursive call

Recursive case: recurs with a simpler parameter

Extra work to convert the result of the recursive call into the result of *this* call

Improving the example

- The line `fill(a, a.length - 1);` just seems ugly
  - Why should we have ask the array how big it is, then tell the method?
  - Why can't the method itself ask the array?
- Solution: Put a "front end" on the method, like so:

```
void fill(int[ ] a) {
    fill(a, a.length - 1);
}
```

- Now in our `run` method we can just say `fill(a);`
- We can, if we want, "hide" the two-parameter version by making it `private`

The four rules

- Do the base cases first
- Recur only with simpler cases
- Don't modify and use non-local variables
  - You can modify them ***or*** use them, just not both
  - Remember, parameters count as local variables, but if a parameter is a reference to an object, only the *reference* is local—***not*** the referenced object
- Don't look down:
  - When you write or debug a recursive function, think about *this* level *only*. If you can get *this* level correct, you will automatically get *all* levels correct.

Base cases and recursive cases

- Every valid recursive definition consists of two parts:
  - One or more *base cases*, where you compute the answer directly, without recursion
  - One or more *recursive cases*, where you do *part* of the work, and recur with a simpler problem

Do the base cases first

- Every recursive function *must* have some things it can do without recursion
- These are the *simple*, or *base*, cases
- Test for these cases, and do them first
  - The important part here is *testing* before you recur; the actual work can be done in any order
  - ```
    long factorial(long n) {
        if (n > 1) return n * factorial(n - 1);
        else return 1;
    }
    ```
  - However, it's usually better style to do the base cases first
- This is just writing ordinary, nonrecursive code
    ```

Recur only with a simpler case

- If the problem isn't simple enough to be a base case, break it into two parts:
  - A *simpler* problem of the same kind (for example, a smaller number, or a shorter list)
  - *Extra work* not solved by the simpler problem
- *Combine* the results of the recursion and the extra work into a complete solution
- “Simpler” means “more like a base case”

Infinite recursion

- The following is the recursive equivalent of an infinite loop:
  - ```
    int toInfinityAndBeyond(int x) {
        return toInfinityAndBeyond(x);
    }
    ```
- This happened because we *recurred with the same case!*
- While this is obviously foolish, infinite recursions can happen by accident in more complex methods
  - ```
    int collatz(int n) {
        if (n == 1) return 1;
        if (n % 2 == 0) return collatz(n / 2);
        else return collatz(3 * n - 1);
    }
    ```

Don't modify *and* use non-local variables

- Consider the following code fragment:
  - ```
    int n = 10;
    ...
    int factorial() {
        if (n <= 1) return 1;
        else {
            n = n - 1;
            return (n + 1) * factorial();
        }
    }
    ```
- It is very difficult to determine (without trying it) whether this method works
- The problem is keeping track of the value of `n` at all the various levels of the recursion
    ```

OK to modify ***or*** use global variables

- When we change the value of a “global” variable, we change it for all levels of the recursion
  - Hence, we cannot understand a single level in isolation
- It’s okay to modify a global variable if we don’t also use it
  - For example, we might update a variable count as we step through a list
- It’s okay to use (read) a global variable if we don’t also try to change it
  - As far as our code is concerned, it’s just a constant
- The problem comes when we try to *both* modify a global variable *and* use it in the recursion

Using non-local variables

- ```
  int total = 0;
  int[ ] b = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  int sum(int n) {
      if (n < 0) return total;
      else {
          total += b[n];
          sum(n - 1);
          return total;
      }
  }
  ```
- `System.out.println("Total is " + sum(9));`
- The global array `b` is being used, but not changed
- The global variable `total` is being changed, but not used (at least, not in any way that affects program execution)
- This program works, and can be understood
  ```

It's **OK** to modify local variables

- A function has its own copy of
  - local variables
  - parameters passed by value (which are effectively local variables)
- Each level of a recursive function has *its own copy* of these variables and parameters
- Changing them at one level *does not change* them at other levels
- One level *can't* interfere with another level

It's **bad** to modify objects

- There is (typically) only one copy of a given object
- If a parameter is passed by reference, there is only one copy of it
- If such a variable is changed by a recursive function, it’s changed at *all levels*
  - Hence, it’s acting like a global variable (one accessible to all parts of the program)
- The various levels interfere with one another
- This can get *very* confusing
- Don’t let this happen to you!

Don't look down

- When you write or debug a recursive function, think about *this* level *only*
- Wherever there is a recursive call, *assume that it works correctly*
- If you can get *this* level correct, you will automatically get *all* levels correct
- You really can't understand more than one level at a time, so don’t even try

We have small heads*

- It's hard enough to understand *one level* of *one function* at a time
- It's almost impossible to keep track of *many levels* of the *same function* all at once
- But you *can* understand *one level* of *one* function at a time...
- ...and that's *all you need to understand* in order to use recursion well

*According to Edsger Dijkstra

Example: member

- ```
  // A façade method to test whether x occurs in a array
  boolean member(int x, int[ ] a) {
      return member(x, a, a.length - 1);
  }
  ```

- ```
  boolean member(int x, int[ ] a, int n) {
      if (a[n] == x) return true;     // one base case
      if (n < 0) return false;        // another base case
      return member(x, a, n - 1); // recursive case
  }
  ```

Proving that member is correct

- `boolean member(int x, int[] a, int n) {`
  - This is supposed to test if `x` is one of the elements `0..n` of the array `a`
- `    if (a[n] == x) return true;`
  - This says: If `x` is in location `n` of the array, then it's in the array
  - This is obviously true
- `    if (n < 0) return false;`
  - This says: If we've gone off the left end of the array, then `x` isn't in the array
  - This is true ***if:***
    - We started with the rightmost element of the array (true because of the front end), and
    - We looked at every element (true because we decrease `n` by `1` each time)
- `    return member(x, a, n - 1);`
  - This says: If `x` isn't in location `n`, then `x` is one of the elements `0..n` if and only if `x` is one of the elements `0..n-1`
- `}`
  - Did we cover all possible cases?
  - Did we recur only with simpler cases?
  - Did we change any non-local variables?
  - We're done!

Reprise

- Do the base cases first
- Recur only with a simpler case
- Don't modify and use nonlocal variables
- Don't look down

Exercise

- Create a recursive function to obtain n^{th} Fibonacci number.
  - Hint: Fibonacci : 0 1 1 2 3 5 8 13 21 …

Exercise

- Create recursive methods to print the following patterns.

Input : n = 5
Output :
* * * * *
* * * *
* * *
* *
*

Input : n = 7
Output :
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*