To Understand Recursion…

Recursion – Real Life Examples

<noun phrase> = <noun>, or
<adjective> <noun phrase>, or
<adverb> <noun phrase>

Example:

terrible horrible no-good very bad day

Recursion – Real Life Examples

ancestor(p) = parent(p), or
parent(ancestor(p))

great great great great great great great great great great great great grandmother.

0! = 1
n! = n * (n-1)!
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 36288000, 362880000, 39916800, 479001600...

Sum the digits in a non-negative integer

/** = sum of digits in n. */
public static int sumDigs(int n) {
    if (n < 10) return n;
    // { n has at least two digits }
    // return first digit + sum of rest
    return n%10 + sum(n/10);
}

sum(7) = 7
sum(8703) = 3 + sum(8700)
    = 3 + 8 + sum(70)
    = 3 + 8 + 7 + sum(0)
Two different questions, two different answers

1. How is it executed?
   (or, why does this even work?)

2. How do we understand recursive methods?
   (or, how do we write/develop recursive methods?)

Stacks and Queues

Stack: list with (at least) two basic ops:
* Push an element onto its top
* Pop (remove) top element
Last-In-First-Out (LIFO)

Like a stack of trays in a cafeteria

Queue: list with (at least) two basic ops:
* Append an element
* Remove first element
First-In-First-Out (FIFO)

Stack Frame

A “frame” contains information about a method call:

- **local variables**
- **parameters**
- **return info**

At runtime Java maintains a stack that contains frames for all method calls that are being executed but have not completed.

Method call: push a frame for call on stack. Assign argument values to parameters. Execute method body. Use the frame for the call to reference local variables and parameters.

End of method call: pop its frame from the stack; if it is a function leave the return value on top of stack.

Questions about local variables

In a call m(...) when is local variable d created and when is it destroyed? Which version of procedure m do you like better? Why?

Frames for methods sum main method in the system

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:
1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack.

When executing method body look in frame for call for parameters and local variables.
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}
```

```java
public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Frame for method in the system that calls method main: main is then called

n >= 10 sum calls sum: n >= 10. sum calls sum:

n < 10 sum stops: frame is popped and n is put on stack:

Using return value 8 stack computes 2 + 8 = 10 pops frame from stack puts return value 10 on stack
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n % 10 + sum(n/10);
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 10 stack computes 4 + 10 = 14 pops frame from stack puts return value 14 on stack

Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n % 10 + sum(n/10);
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 14 main stores 14 in r and removes 14 from stack

Poll time!

```java
public static int sumDigs(int n) {
    if (n < 10) {
        return n;
    }
    return n % 10 + sumDigs(n / 10);
}
```

Assume my program’s main method calls `sumDigs(1837420)` During this call, what is the maximum number of stack frames above (not including `main`’s stack frame)?

Two different questions, two different answers

1. How is it executed?
   (or, why does this even work?)
   It’s not magic! Trace the code’s execution using the method call algorithm, drawing the stack frames as you go. Use only to gain understanding / assurance that recursion works.

2. How do we understand recursive methods?
   (or, how do we write/develop recursive methods?)
   This requires a totally different approach.

Back to Real Life Examples

Factorial function:

0! = 1
n! = n * (n-1)! for n > 0
e.g.: 4! = 4*3*2*1 = 24

Exponentiation:
b^0 = 1
b^c = b * b^(c-1) for c > 0

```
public static int exp(int b, int c) {
    if (c == 0) return 1;
    return b * exp(b, c-1);
}
```

Easy to make math definition into a Java function!

```
public static int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n-1);
}
```

spec says that the value of a call equals the sum of the digits of n

How to understand what a call does

Make a copy of the method spec, replacing the parameters of the method by the arguments

```java
public static int sumDigs(int n) {  // sum of the digits of n
    if (n < 10) return n;  // n has at least two digits
    return n % 10 + sumDigs(n/10);
}
```
Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s): That is, cases where the parameter is small enough that the result can be computed simply and without recursive calls.

If \( n < 10 \) then \( n \) consists of a single digit.

Looking at the spec we see that that digit is the required sum.

```java
/** = sum of the digits of n. * Precondition: n >= 0 */
public static int sumDigs(int n) {
    if (n < 10) return n; // n has at least two digits
    return n%10 + sumDigs(n/10);
}
```

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the method spec and verify that the correct result is then obtained.

\[ \text{return } n\%10 + \text{(sum of digits of } n/10) \]  
// e.g. \( n = 843 \)

Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s).

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.

Two different questions, two different answers

1. How do we write/develop recursive methods?

   Step 1. Have a precise spec!
   
   Step 2. Write the base case(s): Cases in which no recursive calls are needed. Generally for “small” values of the parameters.
   
   Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind. Done suitably, point 4 (about termination) is automatically satisfied.
   
   Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.

2. How do we understand recursive methods?

   (or, how do we write/develop recursive methods?)

   Step 1. Have a precise spec!
   
   Step 2. Check that the method works in the base case(s).
   
   Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the spec and verify correctness.
   
   Once you get the hang of it this is what makes recursion easy! This way of thinking is based on math induction which we don’t cover in this course.
   
   Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.

Writing a recursive method

Step 1. Have a precise spec!

Step 2. Write the base case(s): Cases in which no recursive calls are needed. Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind. Done suitably, point 4 (about termination) is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.
Examples of writing recursive functions

For the rest of the class we demo writing recursive functions using the approach outlined below. The java file we develop will be placed on the course webpage some time after the lecture.

Step 1. Have a precise spec!
Step 2. Write the base case(s).
Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind.
Step 4. Make sure recursive calls are “smaller” (no infinite recursion).

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward:

- `isPal("racecar")` → `true`
- `isPal("pumpkin")` → `false`

A String with at least two characters is a palindrome if
- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

```
e.g. AMANAPLANACANALPANAMA
      have to be the same
      have to be a palindrome
A recursive definition!
```

Example: Is a string a palindrome?

```java
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1) return true;
    // { s has at least 2 chars }
    int n = s.length() - 1;
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1, n));
}
```

The Fibonacci Function

Mathematical definition:

```
fib(0) = 0  two base cases!
fib(1) = 1
fib(n) = fib(n - 1) + fib(n - 2)  n ≥ 2
```

Fibonacci sequence: 0 1 1 2 3 5 8 13 ...

```
/** = fibonacci(n). Prev: n >= 0 */
static int fib(int n) {
    if (n <= 1) return n;
    // { 1 < n }
    return fib(n - 1) + fib(n - 2);
}
```

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward:

- `isPal("racecar")` → `true`
- `isPal("pumpkin")` → `false`

A String with at least two characters is a palindrome if
- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

```
e.g. AMANAPLANACANALPANAMA
      have to be the same
      have to be a palindrome
A recursive definition!
```

Example: Is a string a palindrome?

```java
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1) return true;
    // { s has at least 2 chars }
    int n = s.length() - 1;
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1, n));
}
```

Example: Count the e’s in a string

```java
/** = number of times c occurs in s */
public static int countEm(char c, String s) {
    if (s.length() == 0) return 0;
    // { s has at least 1 character }
    if (s.charAt(0) != c)
        return countEm(c, s.substring(1));
    int n = s.length() - 1;
    return s.charAt(0) == c && isPal(s.substring(1, n));
}
```