Recursion

Alexander Ferworn

Table of Contents

Contents

1 Recursion

Code Self-invocation

Joke: *What is the definition of recursion? SEE the definition for recursion.*

To be reasonably useful Recursive routines must generally have all of the following characteristics:

i) The code calls itself (recursive case) if certain conditions are met,
ii) The code can check for a stopping condition (base case) that allows it to end without calling itself.

- Recursive code allows for certain problems to be conceptualized much more easily. Mathematicians like it because it can be easily related to the concept of “recurrence” relationships in math.
- Recursive algorithms use a Computer Science concept called “divide and conquer”--Essentially, if a problem is too difficult, break it into many smaller problems that may be easier to solve. If they are too big, break them down as well until you have sufficiently small problems that can be solved and used as building blocks to solve the big problems.
- Recursion can always be replaced by iteration but not always easily
- Recursion sometimes has unintended side effects at execution time.

2 Example Factorial

The factorial of n can be defined as

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

This is an iterative definition where the value of n is decremented and multiplied with the decremented value of n (n-1). This keeps going until we hit 1.

The iterative Java code looks like this,

```
    static int fact(int n) {
        int answer=1;
        if(n <= 1)return 1;
        for(int i=n; i>=1 ;i--)
            answer = answer * i;
        return answer;
    }
```

The definition for n! can be rewritten recursively as,

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

This can be translated into Java relatively easily as,

```
    static int recurseFact(int n) {
        // Base Case:
        //    If n <= 1 then n! = 1.
        if (n <= 1) {
            return 1;
        }
        // Recursive Case:
        //    If n > 1 then n! = n * (n-1)!
        else {
            return n * recurseFact(n-1);
        }
    }
```

Both the iterative and the recursive definitions lead to the same answer but in some cases the recursive case may be easier to conceptualize. However, the iterative code is almost always faster (if you can write it).

3 Example Fibonacci numbers

By definition, the first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two. In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

Where $F_0=0$ and $F_1=1$

A recursive version of this relationship can be written in Java as the code below.

```
public class Fibonacci_recursive {
    public static int fib(int n) {
        if (n <= 1) return n; // base case
        else return fib(n-1) + fib(n-2);// recursive case
    }
    public static void main(String[] args) {
        int N = 1000;
        for (int i = 1; i <= N; i++)
            System.out.println(i + ": " + fib(i));
    }
}
```

However, the code can also be rewritten using iteration. The recursive version of the code closely resembles its mathematical equivalent. An argument could be made that this aids understanding of the problem. However, if one were to run the two version of the program the iterative approach will always execute much faster.

In practice it is sometime difficult to write an iterative version of a recursive algorithm. This just doesn't happen to be one of those times.

4 Example: Anagrams

Anagrams are words that are created from the rearrangement of the letters of other words (permutations).

For example, if we consider the word "east", an anagram program should list all 24 permutations, including eats, etas, teas, and non-words like tsae. If we want the program to work with any length of word, there is no straightforward way of performing this task without recursion.

Using recursion, we simply need to find the base case and the recursive case. If we had a four-letter word, a useful assumption that allows us to presume our recursive method knows how to handle all words with fewer than four letters. So what we might hope to do is to take each letter of the four-letter word, and place that letter at the front of all the three-letter permutations of the remaining letters. Given east, we would place e in front of all six permutations of ast — ast, ats, sat, sta, tas, and tsa — to arrive at east, eats, esat, esta, etas, and etsa. Then we would place a in front of all six permutations of est, then s in front of all six permutations of eat, and finally t in front of all six permutations of eas. Thus, there will be four recursive calls to display all permutations of a four-letter word.

Of course, when we're going through the anagrams of ast, we would follow the same procedure. We first display an a in front of each anagram of st, then an s in front of each anagram of at, and finally a t in front of each anagram of as. As we display each of these anagrams of ast, we want to display the letter e before it.

To translate this concept into Java code, our recursive method will need two parameters. The more obvious parameter will be the word whose anagrams to display, but we also need the letters that we want to print before each of those anagrams. At the top level of the recursion, we may want to print all anagrams of east, without printing any letters before each anagram. But in the next level, one recursive call will be to display all anagrams of ast, prefixing each with the letter e. And in the next level below that, one recursive call will be to display all anagrams of st, prefixing each with the letters ea.

The base case of our recursion would be when we reach a word with just one letter. Then, we just display the prefix followed by the one letter in question.

The Java code looks like this;

```
public static void printAnagrams(String prefix, String word) {
   if(word.length() <= 1) {
       System.out.println(prefix + word);
   }
   else{
       for(int i = 0; i < word.length(); i++) {
           String cur = word.substring(i, i + 1);
           String before = word.substring(0, i);
                          // letters before cur
           String after = word.substring(i + 1);
                          // letters after cur
           printAnagrams(prefix + cur, before + after);
       }
   }
```