Data Structures - Recursion 1

Dr. TGI Fernando [1] [2]

[1] Email: tgi.fernando@gmail.com
[2] URL: http://tgifernando.wordpress.com/

What is recursion?

Sometimes a problem is too difficult or too complex to solve because it is too big. If the problem can be broken down into smaller versions of itself, we may be able to find a way to solve one of these smaller versions and then be able to build up to a solution to the entire problem.

This is the idea behind recursion; recursive algorithms break down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results.

What is recursion? (Contd.)

- Recursion is a programming technique in which a method (function) calls itself.
- Produces repetitions without using **LOOPS**.
- Produces substantial results from very little code.
- Allows elegantly simple solutions to difficult problems.
- But it can also be misused, producing inefficient code.
- Recursive code is usually produced from recursive algorithms.

The Factorial Function

If n is a non-negative number, the factorial of n is mathematically defined as

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

This is a recursive definition because the factorial "recurs" on the right side of the equation. The function is defined in terms of itself.

n	0	1	2	3	4	5	6	7	8	9
$n!$	1	1	2	6	24	120	720	5040	40310	362880

The first 10 values of the factorial function

The first value, 0! is computed by the upper half of the above definition.
The other values are computed by the lower half of the above definition.

Recursive implementation of the Factorial Function

When a function is defined recursively, its implementation is usually a direct translation of its recursive definition (a).

```
int factorial (int n) {
  if (n==0) // basis
    return 1;
  else // recursive part
    return n*factorial(n-1);
}
```

(a) Recursive

```
int factorial (int n) {
  int f = 1;
  for (int i=2; i<=n; i++) {
    f = f*i;
  }
  return f;
}
```

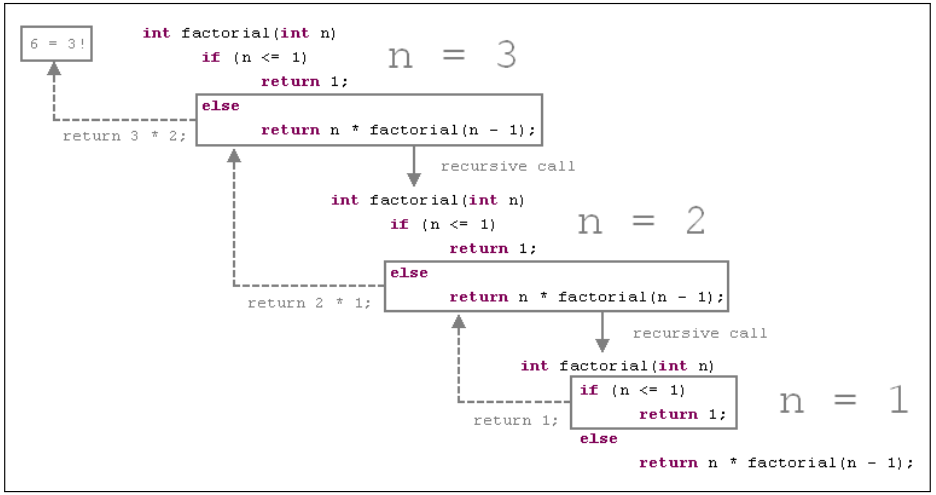
(b) Iterative

Simple test driver for the factorial method

```
public static void main(String[] args) {
  for (int n=0; n<10; n++) {
    System.out.println("f("+n+") = "+f(n));
  }
}
```

What's really happening?

Calculation of 3! in details



Basis and Recursive Parts

To work correctly, every recursive function must have a basis and a recursive part. The **basis** is what stops the recursion. The **recursive part** is where the function calls itself.

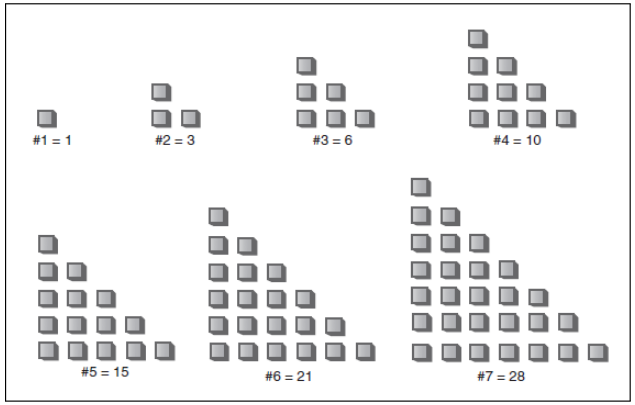
Note:

- Every recursive function must have at least one base case (many functions have more than one).
  If it doesn't, your function will not work correctly most of the time, and will most likely cause your program to crash in many situations, definitely not a desired effect.
- Diminishing the size of problem
  On each recursive call the problem must be approaching the base case. If the problem isn't approaching the base case, we'll never reach it and the recursion will never end.

Triangular Numbers

1, 3, 6, 10, 15, 21, ...
CAN you find the next member of this series?
WHAT is the recursive definition of these numbers?

Triangular Numbers (Contd.)



nth triangle = n + *(n-1)*th triangle

Recursive definition:

$$nth\ number = \begin{cases} 1 & if\ n = 1\ (basis) \\ (n-1)th\ number + n & if\ n > 1 \end{cases}$$

The Fibonacci Numbers

The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

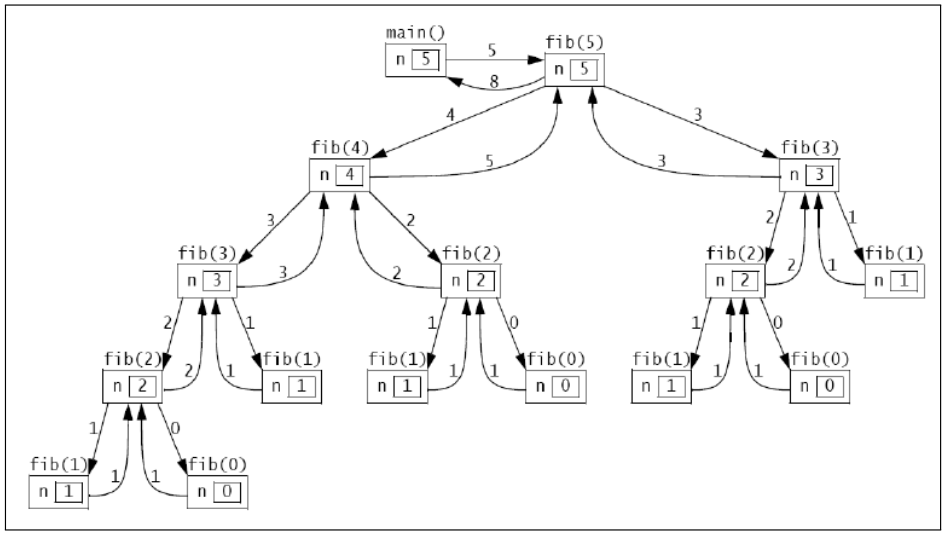
Each number after the second is the sum of the two preceding numbers.

This is a naturally recursive definition:

$$F_n = \begin{cases} 0, & if\ n = 0\ (basis) \\ 1, & if\ n = 1\ (basis) \\ F_{n-1} + F_{n-2} & if\ n > 1\ (recursive\ part) \end{cases}$$

The Fibonacci function is more heavily recursive than the factorial function because it includes two recursive calls.

Tracing the recursive Fibonacci function



Characteristics of Recursive Methods

- It calls itself.
- When it calls itself, it does so to solve a smaller problem.
- There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself.

Avoiding Circularity

- Another problem to avoid when writing recursive functions is circularity.
- Circularity occurs when you reach a point in your recursion where the arguments to the function are the same as with a previous function call in the stack.
- If this happens you will never reach your base case, and the recursion will continue forever, or until your computer crashes, whichever comes first.

E.g.

```
int f (int n){
  if (n==1) return 1; // basis
  else if (n % 2 != 0) return f(n/2);
  else return 1 + f(3*n + 1);
}
```

What is f(4)?

The Call Stack

Frame - When a function is called, a certain amount of memory is set aside for storing local variables, function's address in memory (this allows the program to return to the proper place after an another function call), etc. This memory is called as a frame.

The Call Stack - Every function has its own frame that is created when the function is called. Since functions can call other functions, often more than one function is in existence at any given time, and therefore there are multiple frames to keep track of. These frames are stored on the call stack, an area of memory devoted to holding information about currently running functions.

The Call Stack (Contd.)

- In the call stack, frames are put on top of each other in the stack.
- When a new function is called (meaning that the function at the top of the stack calls another function), that new function's frame is pushed onto the stack and becomes the active frame.
- When a function finishes, its frame is destroyed and removed from the stack, returning control to the frame just below it on the stack (the new top frame).

The Call Stack (Contd.)

Example:

```
void main () {
  ranjan ();
}

void ranjan () {
  int thisYear = 2017;
  int ageFabrication = 10;
  int trueAgeGeetha = geetha (thisYear) + ageFabrication;
  int trueAgeSwarna = swarna (thisYear) + ageFabrication;
}

int geetha (int curYear) {
  int bornYear = 1965;
  int age = curYear - bornYear;
  return age;
}

int swarna (int curYear) {
  int bornYear = 1958;
  int age = curYear - bornYear;
  return age;
}
```

The Call Stack (Contd.)

(a) First, the program calls `main()` and so the `main()` frame is placed on the stack.

(b) The `main()` calls the function `ranjan()`. So `ranjan()`'s frame push onto the stack.

(c) Next `ranjan()` calls `geetha()`. `geetha()`'s frame also push onto the stack.

(d) When the function `geetha()` is finished executing, its frame is deleted from the stack and control returns to the `ranjan()`'s frame.

The Call Stack (Contd.)

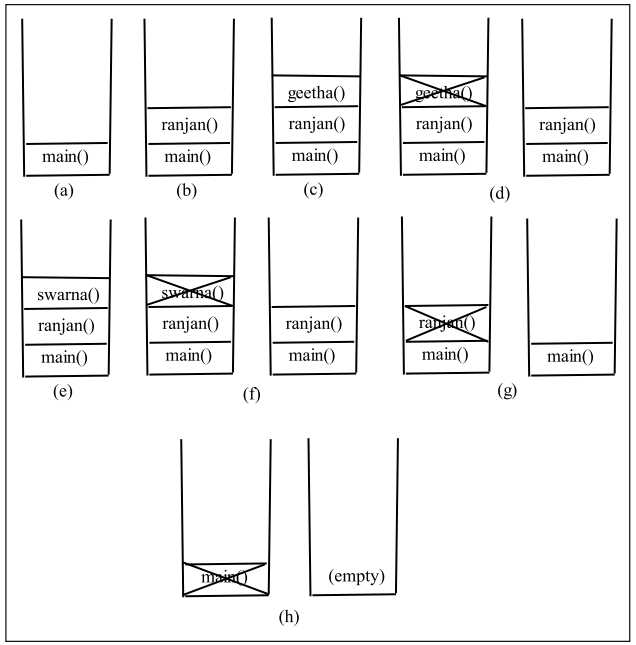
(e) After regaining control, `ranjan()` then calls `swarna()`. Thus, `swarna()`'s frame push onto the stack.

(f) When the function `swarna()` is finished executing, its frame is deleted from the stack and control returns to `ranjan()`.

(g) When `ranjan()` is finished, its frame is deleted and control returns to `main()`.

(h) When the `main()` function is done, it is removed from the call stack. As there are no more functions on the call stack, and thus no where to return to after `main()` finishes, the program is finished.

The Call Stack (Contd.)



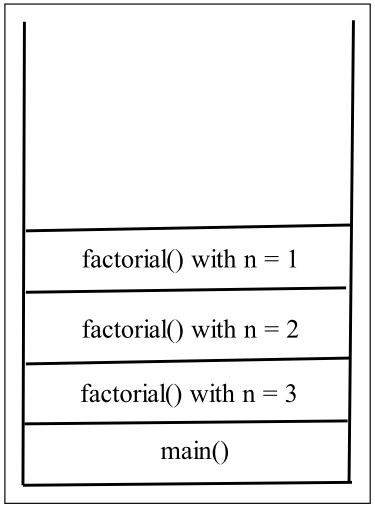
Efficiency

Recursion might not be the most efficient way to implement an algorithm.

- Each time a function is called, there is a certain amount of "overhead" that takes up memory and system resources.
- When a function is called from another function, all the information about the first function must be stored so that the computer can return to it after executing the new function.

Recursion and the call stack

- When using recursive techniques, functions "call themselves."
- For example, the function `factorial()` calls itself during the course of its execution.
- However, as mentioned before, it is important to realize that every function called gets its own frame, with its own local variables, its own address, etc.
- As far as the computer is concerned, a recursive call is just like any other call.



Overhead of recursion

- Imagine what happens when you call the factorial function on some large input, say 500.
- The first function will be called with input 500.
- It will call the factorial function on an input of 499, which will call the factorial function on an input of 498, etc.
- Keeping track of the information about all active functions can use many system resources if the recursion goes many levels deep.
- In addition, functions take a small amount of time to be instantiated, to be set up.
- If you have a lot of function calls in comparison to the amount of work each one is actually doing, your program will run significantly slower.

Is recursion necessary?

- Often, you'll decide that an iterative implementation would be more efficient and almost as easy to code (sometimes they'll be easier, but rarely).
- It has been proven mathematically that any problem that can be solved with recursion can also be solved with iteration, and vice versa.
- However, in some cases, recursion is a blessing and in these instances you should use recursion without any hesitation.

  E.g. Recursion is often useful tool when working with data structures such as `trees`.

When ... recursion?

(1) **The problem is much more clearly solved using recursion.**
There are many problems where the recursive solution is clearer, cleaner, and much more understandable. As long as the efficiency is not the primary concern, or if the efficiencies of the various solutions are comparable, then you should use the recursive solution.

(2) **Some problems are much easier to solve through recursion.**
There are some problems which do not have an easy iterative solution. Here you should use recursion. The *Towers of Hanoi problem* is an example of a problem where an iterative solution would be very difficult.