Recursion example

Here is a recursive C function that returns the factorial of the integer number passed as an argument.

```c
int factorial(int n) {
        if(n > 1)
                return n * factorial(n-1);

        else
                return 1;
}
```

Here is one possible version of assembly code for this function.

```
factorial:      PUSH {R4, LR}
                CMP R0, #1
                BGT if
                BLE else

if:             MOV R4, R0
                SUB R0, R0, #1
                BL factorial
                MUL R0, R0, R4
                B return

else:           MOV R0, #1
                B return

return:         POP {R4, LR}
                BX LR
```

To help you get from the C to assembly, we highlight some points. The main point to keep in mind is that recurios is meant only to confuse you. As long as you follow the calling convention, the code will work, whether a recursive function/subroutine or not.

Some points that involve both C and assembly

- Arguments (in C) are contained in R0 – R3 (in assembly)

  So if we have a function in C: **int func(int a, int b, int c, int d);**

  Then a will be in R0, b in R1, c in R2, and d in R3
- Return value (in C) should be put in R0 (in assembly)
- An assembly subroutine that implements a C function will always end in **BX LR**

Some points that only involve assembly

- R0 – R3 can be modified without saving the contents first. While this makes using these registers convinient for an assembly subroutine, the subroutine has to be careful to not place values that it needs in these registers. If another subroutine is called in the middle, there is no guarantee that the values in these registers will be the same.
- If any register other than R0-R3 is used, it must be saved (put on the stack). This holds true for the LR (Link register), SP (Stack pointer) as well.

Let's look at the example above in light of these points.

- The first **PUSH** is there because both R4 and LR are used in the subroutine
- **CMP R0, #1** uses the fact that the argument is passed in R0
- **MOV R4, R0** is done to store the value of R0, since it will soon be modified
- **SUB R0, R0, #1** places the new argument (n-1) in the register R0
- **MUL R0, R0, R4** uses the fact that the return value of the factorial(n-1) function is in R0. This is multiplied with R4, which holds the previously stored value of n, and the result is placed back into R0, which is the return value register.
- **MOV R0, #1** is the other return case in the c code, where again the return value is placed in the return register R0
- Finally we **POP** R4 and LR to restore them back to the values that we pushed at the beginning of the subroutine, and then branch to the link register (BX LR).

Hopefully that example made sense. Now here is some slightly more advanced assembly code that implements the same function

```
factorial:      PUSH {CPSR}
                CMP R0, #1

                PUSHGT {R4, LR}
                MOVGT R4, R0
                SUBGT R0, R0, #1
                BLGT factorial
                MULGT R0, R0, R4
                POPGT {R4, LR}

                MOVLE R0, #1

                POP{CPSR}
                BX LR
```

Rather than have the labels **if** and **else** as before, we have used conditional execution for all instructions. The important difference here is that since the execution of all our instructions depends on the CPSR register, we cannot allow it to be corrupted and therefore must save it's contents within each sunroutine call.

You don't need to write code like shown in the second approach, and it may not even be better. The point of it is to give another example of using the stack and to show that conditions can be used on any instructions.

Finally, it is difficult to differentiate between a looped v/s recursive implementation in assembly. The main point to keep in mind here is that if you do not have a BL to the same subroutine within the subroutine, then it is not recursive. Recursive assembly subroutines will always have a structire similar to this:

```
subroutine_name:        ...
                        ...
                        BL subroutine_name
                        ...
                        ...
                        BX LR
```

Finally, here is the factorial function (in C) and subroutine (in assembly) implemented as a loop.

```c
int factorial(int n) {
        int retval = 1;

        while(n > 1) {
                retval = retval * n;
                n = n-1;
        }

        return retval;
}
```

```
factorial:      PUSH{R4}
                MOV R4, R0
                MOV R0, #1

loop:           CMP R4, #1
                BLE return
                MUL R0, R0, R4
                B loop

return          POP {R4}
                BX LR
```

It might prove to be a useful exercise to verify that the assembly version adheres to the calling convention as highlighted above.