Recursion

In lecture, we have talked about how to write functions. Unlike in C, calling a function in MIPS requires more than just invoking it. Before calling a function, the function *making* the call (known as the *caller*) needs to save values of registers that the function may use and overwrite (these are known as *caller-saved registers*). The called function, known as the *callee* also needs to save values of some registers, which are appropriately named *callee-saved registers*.
Following the register-saving convention becomes even more critical when using recursive functions, which is the topic of this discussion.

Some Definitions

A *recursive function* is a function that calls itself. To ensure that the self-calling eventually terminates, there are two parts to every recursive function, *base case* and *recursive case*. The base case defines the condition which terminates the recursion. The *recursive case* calls the function recursively, but with different input(s) that come progressively closer to the *base case* condition. Reaching the *base case* results in termination of the recursion and unwinding of the stack of recursive calls.

Example

Let's write a recursive function that does simple math. Given two integers, n and k, where $n \leq k$, find the sum of integers from n to k, inclusive (*e.g.*, if $n = 2$, and $k = 4$, our program will add 2 + 3 + 4)

Recursion in high level languages

First, let's try to write this function in our favorite C-like language. There are two inputs to this function, so the signature is:

```
int mySum(int n, int k);
```

The function proceeds to count numbers from n up to k by incrementing n by 1 each time. The *recursive case* needs to perform the addition using a recursive call:

```
return n + mySum (n + 1, k);
```

The *base case* of the algorithm occurs when $n = k$. Note that we're assuming that $n \leq k$ in the beginning.

```
if(n == k)
     return n;
```

So, here's the complete function:

```
int mySum (int n, int k) {
     if (n == k)
          return n;
     return n + mySum (n + 1, k);
}
```

It is short and elegant, which is one of the appeals of recursive functions.

Implementing recursion in MIPS

Now, let's write the MIPS code, doing tiny steps, eventually arriving at the correct solution.

1. *Review register conventions and name variables.*

   First, a reminder about registers: `$a0-$a3` are used to pass parameters to functions, while `$v0` and `$v1` are used for return values. In our case, `$a0` corresponds to n, `$a1` is k and the return value will be stored in `$v0`.

2. *Convert the code for the base case.*

   Converting the *base case* (lines 2 & 3 of C code) is easy, but not trivial. Remember to check for the "else" clause, i.e. if $n \neq k$, because the label must point to the *else (recursive) case*:

```
mySum:
     bne   $a0, $a1, recurse
     move  $v0, $a0
     jr    $ra
```

   Note that, since the *base case* does not have any function calls, it is not necessary to save/restore any values to the stack. The *recursive case* (line 4 of C code) is more complex and requires multiple steps:

3. *Save callee- and caller-saved registers on the stack. How much stack space has to be allocated?*

   Callee-saved registers are `$s`-registers and `$ra`. Other registers are caller-saved. Recursive functions are **both caller and callee**, so we need to save all registers. Even if we're sure that some registers will not change (e.g., `$a1`), it's still **good programming practice** to save them.

```
recurse:
     sub  $sp, $sp, 12   # allocate stack space: 3 values * 4 bytes each
     sw   $ra, 0($sp)
     sw   $a0, 4($sp)
     sw   $a1, 8($sp)    # add this just for completeness
```

4. *Call mySum recursively.*

   Before we can perform the addition (line 4 of C code), we need to obtain the result of the function call. Since `$a1` already has the right value, we only need to modify `$a0` and call `mySum`.

```
     addi $a0, $a0, 1
     jal  mySum
```

5. *Clean up the stack and return the result.*

   To perform the addition, we need to use the old value of `$a0` (n, not $n + 1$), which we fortunately stored on the stack.

```
     lw   $a0, 4($sp)
     add  $v0, $v0, $a0
```

Now `$v0` has the correct return value, but the stack pointer is still not reset and we don't know where to return.

```
lw   $ra, 0($sp)
addi $sp, $sp, 12
jr   $ra
```

That was exciting. Now it's your turn ...

---

1 Recursion in MIPS

Implement the Fibonacci function in MIPS given the following C code.

```
int fib (int n) {
     if (n <= 1)
          return n;
     else
          return fib (n - 1) + fib (n - 2);
}
```

Note that this code contains **two** recursive calls. Be careful and save the result of the first `fib` before calling it again.

1. *Assign register names to variables and determine which is base case and which is recursive.*

2. *Convert the code for the base case.*

3. *Save callee- and caller-saved registers on the stack.*

4. *Call `fib` recursively.*

5. *Call `fib` recursively* ***again***.

6. *Clean up the stack and return the result.*

2 MIPS to C

In the following MIPS assembly code, the value in register `$a0` is an input and the value in register `$v0` is the output.
Note:

1. You must **NOT** use **gotos**. You must use only **conditional if-else**, **Switch case statements** and **loops** to alter control flow.
2. You must be able to figure out and write **correct prototypes** for C functions you translate by looking at the MIPS code.
3. You must use array indexing and not translate addresses literally using pointer arithmetic. For example:
   The MIPS code given below can be translated in two possible ways:

   ```
   lw $t0, 8($a0)
   ```

Proper Translation	Improper Translation!
`i = a[2]`	`i = *(a + 2×4)`

1. *Translate the following MIPS function into an equivalent C function:*

```
func:
        addi   $t0, $zero, 1
        addi   $v0, $zero, 1
Loop:   sle    $t1, $t0, $a0
        beq    $t1, $zero, Exit
        mul    $v0, $v0, $t0
        addi   $t0, $t0, 1
        j      Loop
Exit:
        jr $ra
```

2. *What mathematical function does this code perform?*