# Lab 1 Report

### Iterative Exponentiation

**a1/r0 : x, a2/r1: n; returns a1/r0 = x^n**

This is quite simple; here, we move x from R0 to R2 to use as a loop pointer, and move 1 to R0; In our loop, we first decrement A2 and check if the result is negative, then multiply R0 by x.

This means that if we start with n set to 0, we will return from our loop immediately without requiring a second n == 0 check outside. Because we have already moved 1 to R0 to multiply by x n times, our return value is already correct.

We are only using the scratch registers, and the function is iterative, so we don't need to push anything to the stack.

### Recursive factorialization

**r0: n; returns r0 = n!**

Implementing this recursively is rather unfortunate, as an iterative or tail-recursive approach is very easy and removes all of these unessecary memory accesses. We do not need to save any registers upon entering the program, as we only use the scratch registers. First, we check for n < 2, and for both base cases 1! and 0! we will return 1, so only a single check is needed. Otherwise, we will need to push n (R0) and the link register to the stack, because we recurse into n-1!. Once this is done, we pop n into R1 (to not overwrite the result returned by n-1!, which will be in R0), multiply and return by popping the link register into program counter.

Here is a tail recursive implementation, the fac function sets the accumulator before calling the recursive function:

```
fac:
    mov a2, a1
    mov a1, #1
facr:
    cmp a2, #2
    bxlt lr
    mul a1, a1, a2
    sub a2, a2, #1
    b facr
```

### Exponentiation By Squaring

**a1/r0 : x, a2/r1: n; returns a1/r0 = x^n**

The first thing we calculate is whether n is even or odd, and store 1 if it is even, and x if it is odd. This serves two purposes: first, if $n < 2$, we can just return this value; if n were 0, we would return 1 as n is even, if it is 1, we return x because it is odd. Secondly, in the recursive case we can use this value to multiply in our exponentiation by squaring; when n is odd, we need $(x^2)(n/2)$, when it is even $x*(x^2)(n/2)$. If we are recursing into another function call, we will push this value into the stack, and then pop to a different register after our recursive function returns to avoid overwriting the return value.

Here is a tail recursive implementaiton for exponentiation by squaring, which operates similarly but stores the value to multiply by multiplying it into the accumulator:

```
exp:
    mov a3, a1
    mov a1, #1
expr:
    tst a2, #1
    mulne a1, a1, a3
    cmp a2, #0
    bxle lr
    mul a3, a3, a3
    lsr a2, a2, #1
    b expr
```

**Quicksort**

**a1/r0 : array pointer, a2/r1: start index, a3/r2: end index**

The swap function needs to push 2 saved registers to the stack because each memory address needs to be loaded into a register before being swapped.

The quicksort function uses 2 saved registers, for loading the values of the pivot and the array element we index through; we will also push a3 to the stack because we will need to get the end position of the array again at the very end of our function, before calling quicksort recursively from the pointer position + 1 to the end of the array, and in the meantime this value will get overwritten.

In the quicksort function, there are a few things worth noting: firstly, it is unessecary to iterate i until the end of the list, but merely until it is greater than j; this is because once i surpasses j, no further swaps will be preformed, so any elements it finds in the remainder of the list are completely irrelevant. The value of i is not used anywhere else. Secondly, note that the 3 while loops in the c code template have been condensed into merely 2 labels here: this is because the outer while loop immediately begins the execution of the first loop to iterate i. Next, notice that at the very end of our quicksort function, we need to call quicksort twice, but after the second call the our quicksort function immediately ends; this means that we can call the second quicksort tail recursively, because

there is nothing more to do in the parent recursive function.

Next, we can show through strong induction that the quicksort function will preserve the value of the end index and thus avoid having to store this value in between our two quicksort calls: if the length of the array is 0, the function immediately returns, so the 3rd argument will not be modified. Then, we use our inductive hypothesis that for an array of length 0...n, our quicksort function will preserve the value of A3. Then consider what happens in the case of a list of length n+1: we will call quicksort recursively twice on lists of length at most n (because even worse case, where the pivot is the greatest element in the list, we do not include the pivot's position in subsequent calls). Our second recursive quicksort call runs on the array between the pivot position + 1 and the end index of the n+1 quicksort, and we tail recurse into this function; thus, if this function preserves the value of a3, which we have by our inductive hypothesis, so must quicksort for a list of length n+1. We use this to avoid needing to push the index of our pivot before calling the first recursive quicksort call, which goes from 0 to pivot position - 1, and merely add 2 to this value while moving it into a2 to get pivot position + 1, the starting index of our second quicksort call.

I created a further optimized version of the quicksort function by having the swap functionality inline, rather than calling it as a helper function. Here is that implementation, which needs to use an additional saved register as a temporary value for swapping:

```
quicksort:
    cmp a2, a3
    bxge lr
    push {a3, v1, v2, v3, lr}
    mov a4, a2 // a2 = pivot location
    ldr v2, [a1, a2,lsl#2] // v2 = pivot value
qsiloop: // find leftmost number greater than pivot, save index to a4
    ldr v3, [a1, a4,lsl#2] // v1 = possible out of order value
    cmp v3, v2
    bgt qsjloop
    add a4, a4, #1
    cmp a4, a3 // comparing to the end of the list vs comparing to j is the same
    blt qsiloop
qsjloop: // find rightmost number less than pivot at, save index to a3
    ldr v1, [a1, a3,lsl#2]
    cmp v1, v2
    subgt a3, a3, #1
    bgt qsjloop
    cmp a4, a3
    // swap if out of order
    bge qsloopout
    str v3, [a1, a3,lsl#2]
    str v1, [a1, a4,lsl#2]
```

```
    b qsiloop // repeat if maybe more out of order elements
qsloopout:
    //mov a2, a4
    // swap pivot with last number smaller than it
    str v2, [a1, a3,lsl#2]
    str v1, [a1, a2,lsl#2]
    sub a3, a3, #1
    bl quicksort
    add a2, a3, #2
    pop {a3, v1, v2, v3, lr}
    b quicksort
```

This runs with in 311 instructions executions, 79/46 loads/stores, and is 184 B, so I am fairly content with how optimized it is.