Exercise 'NIC programs'

Multiplication of an array

```
// Computes res = a[0] * a[1] * a[2] * ...

        word aSize 4 // length of array a
        word a 1 2 3 4 // the array
        word res

        // Compute start address
        loadc   r1 a        // r1 = &a[0] (the address of a[0])

        // Compute where to stop
        loadc   r0 a        // r1 = &a[0]
        load    r2 aSize    // r2 = number of elements in a
        add     r2 r2 r2    // r2 = 2*r2 = number of bytes in a
        add     r0 r0 r2    // r0 = &a[aLen] (first address after array)

        loadc   r5 1        // r5 used to compute res
Loop:   loadr   r2 r1       // r2 = a[i]
        mul     r5 r5 r2    // r5 = r5 * r2
        addc    r1 2        // i++
        jumpn   r1 Loop     // if &a[i] != &a[aSize] goto Loop

        store   r5 res
```

Moving around things in memory

```
// moves the word a value 0xff that is stored at position 0x80 to 0xff in RAM.
    word a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0xff

    loadc r1 a  // initialize array.


    addc r1 0x58    // add 0x58 to the index. We will end up at 0x80 after this.
    loadr r2 r1 // store the value, i.e. 0xff in register r2.
    move r2 r3  // create a copy of r2 in r3.
    loadc r2 0  // r2 := 0.
    storer r2 r1    // store zero (r2) at the previous position of 0xff.
    addc r1 126 // add 126 to the index count. this will be position 0xff in memory.
    storer r3 r1    // store 0xff value in r3 in memory position 0xff (in RAM)
```

Fibonacci generator

```
// computes fibonacci sequence and stores result in RAM

    word init 1 1   // the initial array needed to generate a sequence

    loadc r1 init   // load the initial array into r1

```

```
    loadc r0 11 // the number of iterations we want to make
    loadc r2 0  // starting value for the number of iterations

    addc r1 2   // set the index of array to 1 ( out of 0,1,2...)

Loop:   addc r1 -2  // decrease the index of array by 1
    loadr re r1 // initialize F_{n-2} item in array to re
    addc r1 2   // increase n by 1 (the index)
    loadr rf r1 // initialize F_{n-1} item in array to rf
    addc r1 2   // increase n by 1 (the index)
    add rd re rf    // add value at re and rf and put it into rd
    storer rd r1    // store the rd value in F_{n}
    addc r2 1   // increase the starting value by 1 (i.e. i++)
    jumpn r2 Loop   // loop if r2 != r0
```

Exercise 'wost case ordo and ordo in general'

Loop 1

I chose the for loop as the *loop invariance*. Then, $O(n)$.

Loop 2

The i is never increased. Thus the array will continue forever. There is no worst case then. NB: If this is a typo, and it should say $i++$ in the array, then $O(n)$.

Loop 3

The i is never increased, hence no worst case. If it's assumed that i is increased by 1, we have $O(n^2)$.

Loop 4

I assume that all for-loops have an implicit increment of 1 on each iteration for the variable given in the initialization. Then, $O(n^2)$

Loop 5

The same assumption regarding the dummy variable that keeps track of the index. Then, $O(n^4)$. We will sum $1+2+3+4+5+\cdots+x$ n^2 times.

Exercise 'explain ordo'

Statement We want to show that $(n+1)^3 = O(n^3)$.
Explanation Let's expand LHS.

$$(n+1)^3 = n^3 + 3n^2 + 3n + 1$$

The definition states that $\exists c > 0, \exists n_0 > 0 : f(n) \leq cg(n), \forall n \geq n_0$. In our case, we want to find positive constants c, n_0 such that $n^3 + 3n^2 + 3n + 1 \leq cn^3$ for all $n \geq n_0$.

Let's divide both sides by n^3, i.e.

$$\frac{n^3 + 3n^2 + 3n + 1}{n^3} \leq c$$
$$1 + \frac{3}{n} + \frac{3}{n^2} + \frac{1}{n^3} \leq c$$

Already at this stage we see that the constant will be finite given our restriction that $n > n_0$, where $n_0 > 0$. In fact, as $n \to \infty$, c can be as little as 1.$\square$

Exercise 'reverse algorithm analysis'

The algorithm will have $T(n) = O(n)$. The key operation is the number of times the for loop is being executed. That's because the operation involved inside the for loop have a constant time, i.e. $O(1)$, so it won't affect the final ordo expression. It's $O(n)$, even if the algorithm will take $T(n) = n/2$ times, assuming we count the number of times the loop is being executed.

If the array contains identical elements, then the swap operation is not going to be executed. Assuming it takes a constant amount of time to execute it, no noticeable change is going to be observed. Maybe, some more time, depending on the how much time the *swap* operation takes.