Maciej Kowalczyk

SORTING:

Project 1

```
array:
        .quad 10, 11, 25, 5, 33, 29, 9, 22, 6, 8

size:
        .quad 10

print_num:
        .ascii "%d\n\0"
        .end
```

```
_start:
        ldr     x3, =size //load size
        ldr     x10, [x3] //load size (as counter)
        ldr     x2, =array //load array
        BL      selectionsort //do selection sort
        B       print_end


selectionsort:
        cmp     x10, #1 //check if done (x10 = 1, since we do not want to overstep to a nonexistant 11th element)
        b.eq    SSEnd //if so, end
        ldr     x0, [x2] //load 1st element
        B       find_min //find min
```

We begin by loading each of the values we will need, such as array and the size of the array. The size of the array will be used as a counter in the actual selection sort procedure. First we want to check our base case, if the array is equal to 1. If it is, we don't do anything and just spit that value back out. The reason this is 1 as opposed to zero, is because we won't return anything, so we have to account for the case there is only 1 element. Also when there is more than 1 element, we dont want to accidentally overstep into some empty value when we compare i and i+1. After that check, we load up the first element, and we are ready to go.

```
find_min:
        ldr     x4, =size               //load size of array
        ldr     x5, [x4]                //load the int from size (this will be our counter)
        ldr     x6, [x2]                //load first element
        ldr     x4, =array              //load array

        cmp     x5, #0                  //check if counter is 0
        beq     find_min_end            //if so, end
        mov     x7, x6                  //since this is the first element, its currently our min
        add     x4, x4, #8              //we loaded the array second so that we can traverse through it like so.
        sub     x5, x5, #1              //subtract from counter

        Loop:
                cbz     x5, find_min_end        //check if we are done
                beq     find_min_end
                ldr     x6, [x4]                //load next element
                cmp     x6, x7                  //x6 < x7
                b.lt    setMin                  //if true, branch
        L1:
                add     x4, x4, #8              //increase offset
                sub     x5, x5, #1              //counter - 1
                B       Loop                    //restart loop

        setMin:
                mov     x7, x6                  //set x7 to be the new min
                b       L1                      //go back to loop

        find_min_end:
                b       L2
```

Here, find_min is written as an iterative loop. We start by grabbing the size of the array again and loading it into x5, so we can tell when we are done checking the whole array. We also load the array into x4 to traverse through it using x4. We start with a base case of checking if there even are any elements, if not, we go to the end. We will use x7 to store our min value, and x6 to represent current. When we start the procedure, we immediately set x7 to x6, since we haven't seen anything else yet. We add to the offset for checking future elements in the array and we subtract from the counter. Now we do another check to see if we have finished or not, if so, we go to the end. if not, we continue by loading the next element. By adding to x4 earlier, we are traversing through the array in the next calls. This continues until we find the min value in the array and we return it. The min value will be stored in x7 for now There are 2 errors here, which I will discuss at the end of describing the sort.

```
L2:
        mov     x1, x7 //load second element
        cmp     x1, x0 // x1 < x0
        b.lt    swap //go to swap
L3:
        add     x3, x3, #8 //add offset
        sub     x10, x10, #1 //decrement counter
        B       selectionsort //loop

swap:
        ldr     x12, [x2, x0, lsl #3] // load current element
        ldr     x13, [x2, x1, lsl #3] // load min element
        str     x12, [x2, x1, lsl #3] // swap min into currents old position
        str     x13, [x2, x0, lsl #3] // swap current into mins old position
        B       L3

SSEnd:
        BR      LR //return to main
```

After finishing up with finding the min value, we move onto swap. We start at L2, and store the min into x1. x0 is our current value. If our min is smaller than current, we go to swap. Swap simply sets current and min to temporary registers, and then loads them back in to their swapped positions. After we finish with that, we go to L3 and add to the offset and decrease from the counter. We then do selection sort again.

ERRORS: Onto the errors. The main problem is, when coding this, I was setting the registers to be the array values themselves, as opposed to the addresses of these values. As a result, the swap function doesn't actually do anything. Another error is that the find_min section doesn't account for when we finish an iteration of the sort. What that means is, we will constantly get the same min value as the counter is 10 every time, as opposed to going size = 10, size =9, etc. etc. These two errors are the reason that the error does not end up sorted when we print it, and instead is returned as given the first time.

PRINTING:

```
print_end:
        ldr     x11, =size //grab the size of the array
        ldr     x14, [x11] //set the int to be a counter in x14
        mov     x12, #0 //x12 will be used as the offset

        sub     sp, sp, #24 //theres some things we need to save on stack

        print_loop:
                str     x12, [sp, 16] //store the offset
                str     x2, [sp, 8] //store the array
                str     x14, [sp, 0] //store the counter
                ldr     x0, =print_num //set print_num
                cmp     x14, #0 //check if done (counter = 0)
                beq     Exit //if so, exit
                ldr     x1, [x2, x12] //load the current element

                bl      printf  //print the current element

                ldr     x12, [sp, 16] //load the offset
                ldr     x2, [sp, 8] //load the array
                ldr     x14, [sp, 0] //load the counter
                add     x12, x12, #8 //add to the offset
                sub     x14, x14, #1 //decrement from the counter

                B       print_loop //loop


Exit:
        add     sp, sp, #24 //add back to stack
        mov     x0, #0
        mov     w8, #93
        svc     #0
```

Printing is also done iteratively. We begin by loading the size of the array to use as a counter, so that we can tell when we finish printing the array. We then load the int 0 to x12, so that we may use the register as the offset value in printing. Before we print, we must save a few things onto the stack, since printf changes our register values. We save our counter, our offset, and the array. We then print the current element. After we do so, we need to load the things we stored previously. We add to the offset and decrement the counter. We are going to do this until we have printed the entire array. Once we print the entire array, we add back to the stack, and we terminate the program.