Selection Sort Project

Name: Harris Spahic
Pledge: "I pledge my honor I have abided by the Stevens Honor System".

Idea:
Call the selection sort function, sort the array. Once that function has finished, return to the call function and then print elements in the array.

What is the Selection Sort function?

```
void selectionSort(int arr[], int size)
{
   // One by one move boundary of unsorted subarray
   if(size <= 1)
         break;
   int s = 0;
   while(s < size)
   {
      min = arr[s];
      min_indx = s;
      for (int i = s; i < size; i++){
      if (arr[i] < arr[min_indx]){
         min = arr[i]
         min_indx = i ;}}
      s++;
      // Swap the found minimum element with the first element
      swap(&arr[min_indx], &arr[s]);
   }
}
```

Step -1: Write swap helper function.

Add used registers to stack to be restored. When called, x4 will hold the address of i & x5 will hold the address of some other variable j, which we will switch. For our code's purposes that should always be s & our current min.

```
// Swap will return nothing, X4 = address of i, X5 = address of j
swap:
   sub SP, SP, #16
   STUR x9, [SP, #8]
```

```
   STUR x10, [SP, #0]
```

Get the values of s & current min from x5 & x4. Store the loaded values in the opposite order to those same registers. This switches the values stored in each array.

```
   ldr x9, [x4]         // a = arr[s]
   ldr x10, [x5]        // b = arr[current min]
```

```
   STUR x9, [x5]        // arr[current min] = a
   STUR x10, [x4]       // arr[s] = b
```

Restore the old values of x9 and x10, then pop the stack. Then go back to the previous position in the function.

```
   LDUR x9, [SP, #8]
   LDUR x10, [SP, #0]
   add SP, SP, #16
   BR LR
```

Step 0: Branch to selection_sort_start & after print function.

```
.global _start

_start:
   BL selection_sort_start
   BL print_result
```

Step 1: Initialize local variables. (+ Add potentially changed variables to stack. (LR))

```
selection_sort_start:
   ldr x2, =array
   sub SP, SP, #8
   STUR LR, [SP, #0]
   ldr x8, =size              // x8 = Size of array
   ldr x8, [x8]
```

Step 2: For convenience we make x21 represent the ADDRESS of the last element in the array. We will be working with addresses for the rest of our functions. (X8 will continue to hold the total size)

```
   mov x21, x8                // x21 = size of array
   lsl x21, x21, #3
   add x21, x2, x21           // x21 = address after last element of array
(addr + 8 * size)
   subs xzr, x8, #1           // if x8 <= 1 --> end function
   B.LE end_selection_sort
   add x9, x2, xzr            // x9 = s = &a[1]
```

If the size of the array is 1 or less, end the function. Else let s = the base address of the array and start the while loop.

Step 3: While the address of s is less than the size, load in the first index of the partition of the array we are looking at (0, 1, 2, … , size - 1) to x10 and call it min. Then copy s into x11 representing the index i we want to increment. Then save the address (&a[s]) and the first address (&a[i]) into x4 and x5 respectively for use in swap.

```
while_loop:
   subs xzr, x9, x21          // while &arr[s] < &arr[size]
   B.GE end_selection_sort
   ldr x10, [x9]              // x10 = min = a[s]
   mov x11, x9                // &a[i] = &a[s]
   mov x4, x9                 // local variable x4 = &a[s]
   mov x5, x11                // local variable x5 = first address of i
```

Step 4: Now that all the variables needed for our for loop are ready (mainly i = s), we keep checking if i's address is greater than size's. If it is, we exit. Else we get value at index of i & compare it with the current minimum value. If the value at the address of i's index in the array is less than the current min, we switch the minimum x10 with the value at i's address a[i]. We switch the saved address of the minimum as well. Then return to the top of the loop incrementing the address of i by 8 (size of byte).

```
for_loop:
   subs xzr, x11, x21         // if &a[i] >= &a[size] --> exit
   B.GE exit_for_loop
   ldr x12, [x11]             // x12 = a[i]
   subs xzr, x12, x10         // if a[i] < min, switch
```

```
   B.LT switch
return_to_for_loop:
   add x11, x11, #8          // &a[i] = &a[i+1]
   B for_loop
switch:
   mov x10, x12
   mov x5, x11               // switch min address to new min
   B return_to_for_loop
```

Step 5: When we exit the loop, we branch to swap swapping the min value address with the address of s. We then increment to the next element by adding #8 to the address value of s. The portion of the array before s is now sorted. We return to the while loop and keep doing this for-loop until s is at the end of the array.

```
exit_for_loop:
   BL swap                    // Swap minimum with current s
   add x9, x9, #8             // &a[s] = &a[s+1]
   B while_loop
```

Step 6: Once s is at the end of the array we branch to the "end_selection_sort" which loads back our original link address to the line following the selection_sort_start branch under the _start branch. It then pops our stack and goes back to that address.

```
end_selection_sort:
   ldur LR, [SP, #0]
   add SP, SP, #8
   BR LR
```

Step 7: We go to the next line after our selection_sort call, branching to print_result which prints our final results.

```
_start:
   BL selection_sort_start
   BL print_result
```

Step 8: Print result is really simple. It first loads in the base address of our sorted array (x2) into a temporary address x11. And gets an index offset 0 and loads it into x12.

```
print_result:
   mov x11, x2          // make temp = &a[0]
   mov x12, #0
print_loop:
   ldr x0, =solution
   mov x1, x12
   ldr x2, [x11]
   sub SP, SP, #16
   STUR X11, [SP, #0]
   STUR X12, [SP, #8]
   bl printf
   LDUR X11, [SP, #0]
   LDUR X12, [SP, #8]
   add SP, SP, #16
   add x12, x12, #1
   add x11, x11, #8
   subs xzr, x11, x21  // if x11 >= &a[size] end
   B.LT print_loop
   mov x0, #0
   mov w8, #93
   svc #0

solution:
   .ascii "array[%d] = %d\n"
   .end
```

Then we print the element of the array at index x12 with its index, x11 and x12 respectively. We then make a stack reference for our two temporary variables x11 & x12, (printf will overwrite them otherwise). Pop the stack, increment the index and temporary base, check if the new address (base + offset) is less than &a[size]. If it is repeat print_loop for the next element, else end.

And there we have it. That is my selection sort!