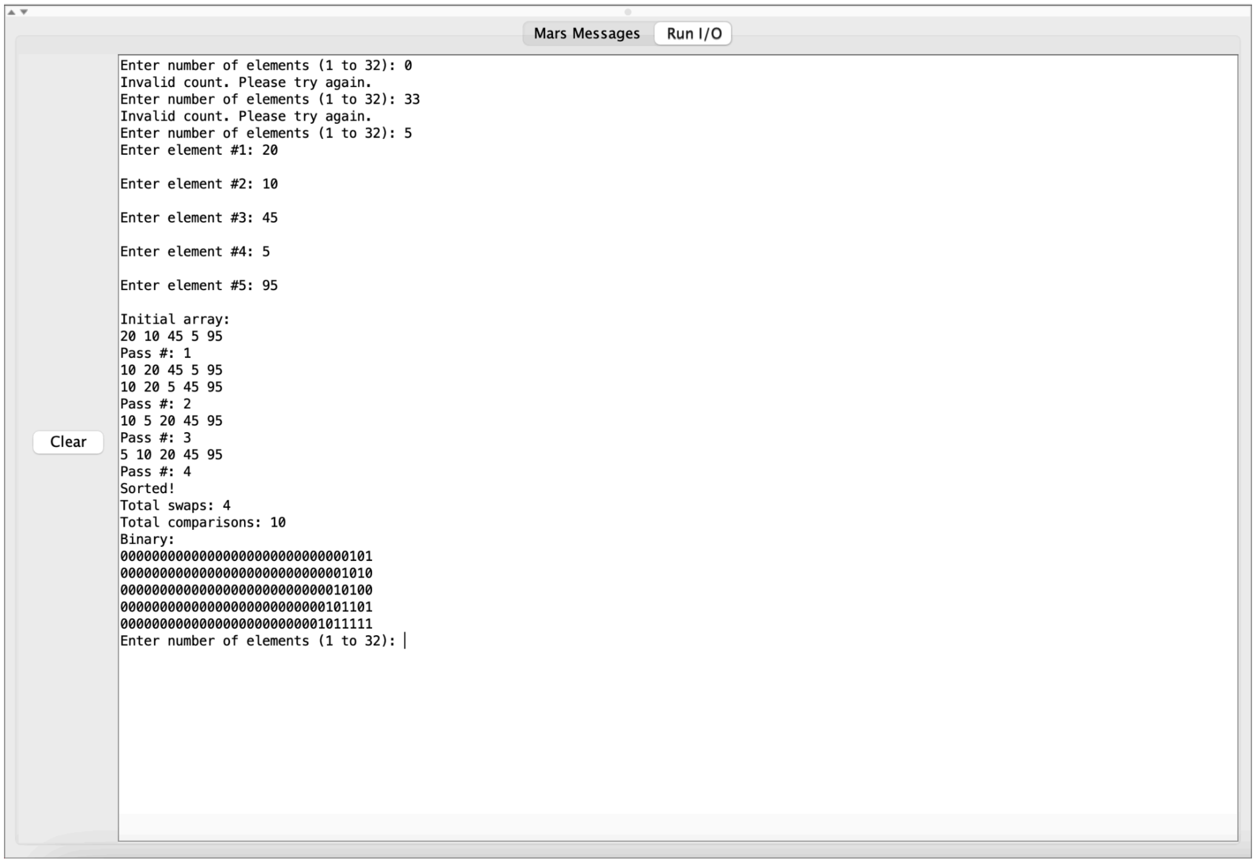
# Bubble Sort Visualizer — asm6.s

## ProjectOverview

My MIPS assembly program implements a bubble sort visualizer that reads a user specified number of elements (1–32), accepts that many integers as input, and then performs a bubble sort on the array. Rather than sorting silently, it prints the array state after each swap, tracks and reports swap and comparison counts, and finally displays each sorted element in binary. The program uses several MARS-specific syscalls, including sleep and binary print to satisfy the extra syscall/tool requirement and provides a clear, step-by-step animation of how bubble sort works.



```
Enter number of elements (1 to 32): 0
Invalid count. Please try again.
Enter number of elements (1 to 32): 33
Invalid count. Please try again.
Enter number of elements (1 to 32): 5
Enter element #1: 20

Enter element #2: 10

Enter element #3: 45

Enter element #4: 5

Enter element #5: 95

Initial array:
20 10 45 5 95
Pass #: 1
10 20 45 5 95
10 20 5 45 95
Pass #: 2
10 5 20 45 95
Pass #: 3
5 10 20 45 95
Pass #: 4
Sorted!
Total swaps: 4
Total comparisons: 10
Binary:
00000000000000000000000000000101
00000000000000000000000000001010
00000000000000000000000000010100
00000000000000000000000000101101
00000000000000000000000001011111
Enter number of elements (1 to 32): 
```

## Features and UserInteraction

### Input Validation

Prompts “Enter number of elements (1–32):”

Rejects values below 1 or above 32, printing “Invalid count. Please try again.” until the user enters a valid count.

### Element Entry

For each index i, it prints “Enter element #i:” and reads an integer. After each entry, a new line is emitted so the console remains organized.

### Initial Array Display

Before sorting begins, the program prints “Initial array:” followed by the list of entered elements. This gives a clear baseline for the sorting animation.

### Bubble-Sort Animation

Implements the classic two-nested-loop bubble sort:

```
Outer loop over passes i = 0 to N–2
Inner loop over j = 0 to N–i–2,
```

comparing adjacent elements. On each out-of-order pair, the elements are swapped, and the entire array is printed on one line. Between prints, the program invokes the MARS sleep syscall (syscall 32) for a short 100 ms pause, giving the user time to observe each step.

### Pass, Swap, and Comparison Counters

Prints “Pass #” and the pass number at the start of each outer iteration. Maintains a swap counter, which is incremented after each swap, and a comparison counter, which is incremented before each comparison, both stored as words in memory. After sorting completes, it prints “Total swaps: X” and “Total comparisons: Y.”

### Binary Output

To demonstrate bit-level representation, the program prints each sorted element in 32-bit binary form using the MARS “print integer in binary” syscall (syscall 35), one line per element.

### Repeat Sorting

After displaying binary output, the program loops back to the initial prompt, allowing the user to sort a fresh array without restarting the simulator.

## ProgramImplementation Details

**Data Section**

String literals (.asciiz) for prompts, labels, and messages.

A fixed-size array buffer (.space 128) to hold up to 32 integers.

Word-sized counters for swaps and comparisons.

**Control Flow**

**readN:** Prompt, read, and validate the element count.

**readElements:** Loop to read and store each element in the array.

**sortArray:** Initialize swap and comparison counters to 0. Print the initial array. Two nested loops (outerLoop, innerLoop) perform sorting.

**Comparison:** Each inner iteration increments the comparison counter, loads two adjacent words, and uses slt/bne to decide on swapping.

**Swap:** On swap, values are stored back into memory, the swap counter is incremented, and the entire array is printed. After each pass, the program prints "Pass #" to indicate progress. When no further passes are needed, control jumps to doneSorting:

**doneSorting:** Prints "Sorted!", swap count, and comparison count. Prints binary representations. Jumps back to readN: for a new session.

**Memory Addressing:** Array indexing uses scaled offsets: compute offset = index × 4 via sll, then add to the base address of array using la and addu, followed by lw/sw.

**MARS-Specific Syscalls Used:** **Print string** (4) and **print integer** (1) for all user prompts and values. **Read integer** (5) to obtain user inputs. **Sleep** (32) to insert a visible pause between each printed array state. **Print integer in binary** (35) to show final array elements in binary form. **Exit** (10) to terminate the program cleanly.

### Challenges and Debugging

**Register Management:** With many counters and indices, choosing a consistent register allocation, for example, mixing $s* for persistent values and $t* for temporaries) prevented accidental overwrites.

**Label Conflicts:** Renaming data-section and code-section labels, for example invalidStr vs. invalidInput resolved assembler symbol conflicts.

**Loop Boundaries:** Carefully calculating the inner-loop limit N – i – 1 and clamping array indices ensured no out-of-bounds memory access.

**Visual Clarity:** Inserting newlines after each integer input and scaling the sleep duration made the console output easy to follow.

## Conclusion

This bubble sort visualizer not only demonstrates the sorting algorithm's mechanics but also showcases advanced MIPS techniques including memory arithmetic, nested loops, and MARS-specific syscalls for sleep and binary output. The program is robust (input-validated, counter-tracked, repeatable). It serves as both a learning tool for sorting and a demonstration of practical MIPS assembly programming under the MARS simulator.