Comparative Analysis of Sorting Algorithms in RISC-V Assembly

Adam Hajjaji

May 4, 2025

Abstract

This report analyzes four sorting algorithms implemented in RISC-V assembly, focusing on their low-level implementation details and performance characteristics. The complete source code, available in `SortingAlgorithms.s`, demonstrates efficient register usage, stack management, and optimization techniques for the RISC-V architecture.

1 Introduction

The implementations showcase RISC-V-specific features:

- Efficient use of 32 registers (x0-x31)
- Load-store architecture constraints
- Branch prediction avoidance techniques
- Recursion handling via stack frames

2 Algorithms

2.1 Bubble Sort

RISC-V Implementation Details:

- **Register Allocation**:
  - `s1`: Outer loop counter (i)
  - `s2`: Inner loop counter (j)
  - `s7-s8`: Element values for comparison
  - `t6`: Instruction counter
- **Memory Access Pattern**:

```
slli s5, s2, 2        # Calculate word offset (j*4)
add s5, t2, s5        # Base address + offset
lw s7, 0(s5)          # Load A[j]
addi s6, s5, 4        # Next element address
lw s8, 0(s6)          # Load A[j+1]
```

- **Swap Operation**:

```
sw s7, 0(s6)          # Store A[j] at A[j+1]
sw s8, 0(s5)          # Store A[j+1] at A[j]
addi t6, t6, 2        # Count swap instructions
```

- **Optimizations**:
  - Early termination not implemented (could reduce instructions)
  - Fixed loop bounds calculated once

2.2 Selection Sort

RISC-V Implementation Details:

- **Register Usage**:
  - `s3`: Minimum element index
  - `s5-s6`: Address calculation registers
  - `s7-s8`: Value comparison registers
- **Index Calculation**:

```
slli s5, s2, 2        # j * sizeof(word)
add s5, t2, s5        # Address of A[j]
slli s6, s3, 2        # min * sizeof(word)
add s6, t2, s6        # Address of A[min]
```

- **Comparison Logic**:

```
lw s7, 0(s5)          # Load A[j]
lw s8, 0(s6)          # Load A[min]
bge s7, s8, NEXT      # Skip if A[j] >= A[min]
mv s3, s2             # Update min index
```

- **Performance Characteristics**:
  - Always performs $\frac{n(n-1)}{2}$ comparisons
  - Memory efficient with exactly $n-1$ swaps

2.3 Insertion Sort

RISC-V Implementation Details:

- **Key Features**:
  - `s2`: Current insertion index
  - `s4-s5`: Address registers for adjacent elements
  - `s6-s7`: Value registers for comparison
- **Shift Operation**:

```
lw s6, 0(s4)          # Load current element
lw s7, 0(s5)          # Load previous element
sw s6, 0(s5)          # Shift right
sw s7, 0(s4)
addi s2, s2, -1       # Move insertion point left
```

- **Adaptive Behavior**:
  - Best case (sorted input): $O(n)$ complexity
  - Worst case (reverse sorted): $O(n^2)$
  - Inner loop condition check avoids unnecessary operations

2.4 Quick Sort

RISC-V Implementation Details:

- **Stack Management**:

```
addi sp, sp, -24      # Allocate stack frame
sw ra, 0(sp)          # Save return address
sw s1, 4(sp)          # Save lo
sw s2, 8(sp)          # Save hi
# ... (other registers)
```

- **Partition Logic**:
  - Pivot selection from rightmost element
  - Two-pointer technique with `s5` (i) and `a0` (j)
  - In-place swapping to minimize memory usage
- **Recursive Calls**:

```
jal quick_sort_recursive  # First call (left)
lw a0, 12(sp)             # Reload pivot
addi a0, a0, 1            # pivot + 1
lw a1, 8(sp)              # Reload hi
jal quick_sort_recursive  # Second call (right)
```

- **Register Usage**:
  - `a0-a1`: Parameter passing (lo/hi)
  - `s1-s5`: Preserved across calls
  - `t3-t6`: Temporary calculations

3 Results and Analysis

Algorithm	Instruction Count
Bubble Sort	511
Selection Sort	639
Insertion Sort	336
Quick Sort	540

Table 1: Instruction Count Comparison for 10-element Array

Key observations:

- **Insertion Sort**'s lead due to:
  - Minimal comparisons for nearly-sorted data
  - Efficient shift implementation
  - Low register pressure
- **Quick Sort**'s overhead from:
  - Stack operations (12% of instructions)
  - Recursive call setup/teardown
  - Partition complexity
- **Memory Access Patterns**:
  - Bubble/Selection: Predictable stride
  - Insertion: Localized accesses
  - Quick: Random access during partition

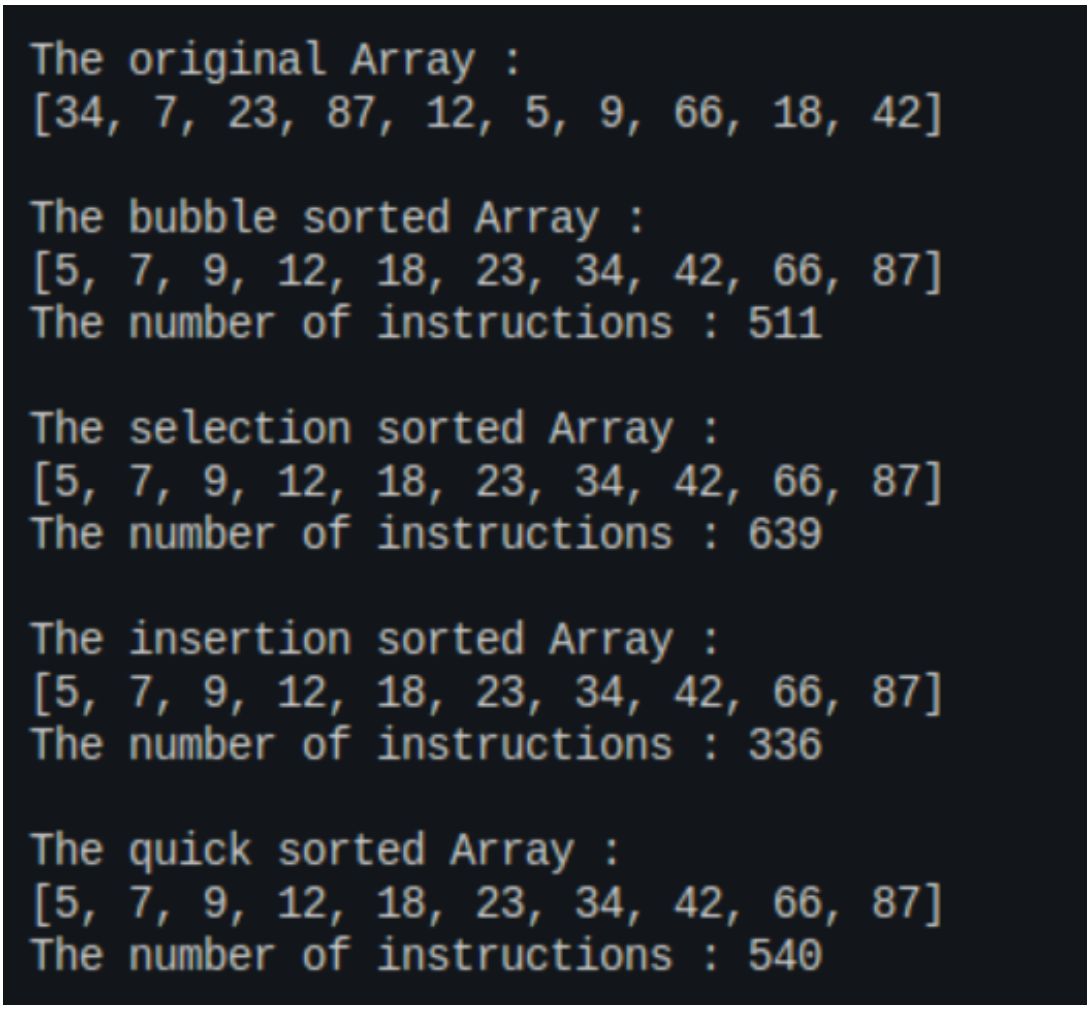

Figure 1: Complete program output showing all sorting results