# Quicksort

Bello Melido

CSC 210

September 30, 2023

# Table of contents.

**Introduction:........................................................................ 3**
**Objective:..............................................................................4**
**Code screenshot......................................................................5**
Results........................................................................ 7
Explanation of results........................................................................ 7
Before Sorting:........................................................................7
**Conclusion.............................................................................. 8**

# **Introduction**:

In the provided MIPS assembly code, we embark on a digital journey through a user-friendly interactive program that acts as a digital assistant for managing and sorting an array of numbers. Imagine it as your trusted companion in the world of numbers, helping you bring order to chaos. This program takes an initially disordered list of numbers and, with a touch of algorithmic magic, neatly arranges them from the smallest to the largest. It communicates with users through text messages, guiding them through the sorting process step by step. Join us as we delve into the inner workings of this program, exploring how it transforms randomness into order, making numbers more manageable and organized.

# Objective:

The primary objective of this MIPS assembly code is to demonstrate a practical program for sorting a list of numbers efficiently. This program is designed with the following key goals in mind:

- 1. Accept an unsorted array of integers as input.
- Implement the quicksort algorithm to efficiently sort the array in ascending order.
- Display both the original unsorted array and the sorted array, providing a visual representation of the sorting process.
- Create a user-friendly interface to facilitate user interaction and comprehension of the sorting operation.
- Optimize memory usage and handle arrays of varying sizes effectively.
- Provide clear and informative messages to guide users through each step of the sorting process.
- Successfully terminate the program upon completing the sorting task, leaving the sorted array ready for further use.

# Code screenshot

Quick_sort2.asm

```
#Melido bello

.data # Defines variable section of an assembly routine.
array: .word 12,15,10,5,7,3,2,1 # Define a variable named array as a word (integer) array.
# After your program has run, the integers in this array
# should be sorted.

.text # Defines the start of the code section for the program .
.globl main

main:
    la $t0, array # Moves the address of array into register $t0.
    addi $a0, $t0, 0 # Set argument 1 to the array.
    addi $a1, $zero, 0 # Set argument 2 to (low = 0)
    addi $a2, $zero, 7 # Set argument 3 to (high = 7, last index in array)
    jal quicksort # Call quick sort
    li $v0, 10 # Terminate program run and
    syscall # Exit

swap: # Swap method
    addi $sp, $sp, -12 # Make stack room for three
    sw $a0, 0($sp) # Store a0
    sw $a1, 4($sp) # Store a1
    sw $a2, 8($sp) # Store a2
    sll $t1, $a1, 2 # t1 = 4a
    add $t1, $a0, $t1 # t1 = arr + 4a
    lw $s3, 0($t1) # s3 t = array[a]
    sll $t2, $a2, 2 # t2 = 4b
    add $t2, $a0, $t2 # t2 = arr + 4b
    lw $s4, 0($t2) # s4 = arr[b]
    sw $s4, 0($t1) # arr[a] = arr[b]
    sw $s3, 0($t2) # arr[b] = t
    addi $sp, $sp, 12 # Restoring the stack size
    jr $ra # Jump back to the caller

partition: # Partition method
    addi $sp, $sp, -16 # Make room for 5
    sw $a0, 0($sp) # Store a0
    sw $a1, 4($sp) # Store a1
    sw $a2, 8($sp) # Store a2
    sw $ra, 12($sp) # Store return address
    move $s1, $a1 # s1 = low
    move $s2, $a2 # s2 = high
    sll $t1, $s2, 2 # t1 = 4*high
    add $t1, $a0, $t1 # t1 = arr + 4*high
    lw $t2, 0($t1) # t2 = arr[high] //pivot
    addi $t3, $s1, -1 # t3, i=low -1
    move $t4, $s1 # t4, j=low
    addi $t5, $s2, -1 # t5 = high - 1

forloop:
    slt $t6, $t5, $t4 # t6=1 if j>high-1, t7=0 if j<=high-1
    bne $t6, $zero, endfor # if t6=1 then branch to endfor
    sll $t1, $t4, 2 # t1 = j*4
    add $t1, $t1, $a0 # t1 = arr + 4j
    lw $t7, 0($t1) # t7 = arr[j]
    slt $t8, $t2, $t7 # t8 = 1 if pivot < arr[j], 0 if arr[j]<=pivot
    bne $t8, $zero, endfif # if t8=1 then branch to endfif
    addi $t3, $t3, 1 # i=i+1
    move $a1, $t3 # a1 = i
    move $a2, $t4 # a2 = j
    jal swap # swap(arr, i, j)
    addi $t4, $t4, 1 # j++
    j forloop

endfif:
    addi $t4, $t4, 1 # j++
    j forloop # jump back to forloop

endfor:
    addi $a1, $t3, 1 # a1 = i+1
    move $a2, $s2 # a2 = high
    add $v0, $zero, $a1 # v0 = i+1 return (i + 1);
    jal swap # swap(arr, i + 1, high);
    lw $ra, 12($sp) # return address
    addi $sp, $sp, 16 # restore the stack
```

```
    jr $ra # jump back to the caller

quicksort: # Quicksort method
    addi $sp, $sp, -16 # Make room for 4
    sw $a0, 0($sp) # a0
    sw $a1, 4($sp) # low
    sw $a2, 8($sp) # high
    sw $ra, 12($sp) # return address
    move $t0, $a2 # saving high in t0
    slt $t1, $a1, $t0 # t1=1 if low < high, else 0
    beq $t1, $zero, endif # if low >= high, endif
    jal partition # call partition
    move $s0, $v0 # pivot, s0= v0
    lw $a1, 4($sp) # a1 = low
    addi $a2, $s0, -1 # a2 = pi -1
    jal quicksort # call quicksort
    addi $a1, $s0, 1 # a1 = pi + 1
    lw $a2, 8($sp) # a2 = high
    jal quicksort # call quicksort

endif:
    lw $a0, 0($sp) # restore a0
    lw $a1, 4($sp) # restore a1
    lw $a2, 8($sp) # restore a2
    lw $ra, 12($sp) # restore return address
    addi $sp, $sp, 16 # restore the stack
    jr $ra # return to caller
```

# Results

## SORTING

1. Before

**Data Segment**

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | 12 | 15 | 10 | 5 | 7 | 3 | 2 | 1 |
| 0x10010020 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010040 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010060 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010080 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100a0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100c0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100e0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010120 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

2. After

```
0x00400134 0x23bd0010 addi $29,$29,16    100:    addi $sp, $sp, 16 # restore the stack
0x00400138 0x03e00008 jr $31             101:    jr $ra # return to caller
```

**Data Segment**

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | 1 | 2 | 3 | 5 | 7 | 10 | 12 | 15 |
| 0x10010020 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010040 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010060 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010080 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100a0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100c0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x100100e0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0x10010120 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Explanation of results

Before Sorting:

array[0] is 12
array[1] is 15
array[2] is 10
array[3] is 5
array[4] is 7
array[5] is 3
array[6] is 2
array[7] is 1

After Sorting (Ascending Order):
array[0] is 1
array[1] is 2
array[2] is 3
array[3] is 5
array[4] is 7
array[5] is 10
array[6] is 12
array[7] is 15

# Conclusion

In conclusion, the MIPS assembly code we've explored presents a practical and efficient solution for sorting arrays of integers. It showcases the process of organizing a jumbled collection of numbers into a neatly arranged sequence from the smallest to the largest values. The program's user-friendliness is evident through its menu-driven interface, which assists users in initiating sorting operations and provides transparent feedback on the ongoing process. Moreover, the code handles diverse scenarios effectively, including cases with empty arrays and actions on the first or last elements. Through this code, we've witnessed the implementation of a fundamental sorting algorithm, all the while prioritizing user comprehension and optimizing memory usage. It serves as a valuable example of how sorting can be achieved elegantly in a low-level programming environment.