Garrett Lo, Darshil Sheth, Diego Perez-Carlos, Cathy Ko

Professor Cariaga

CS 2640.04

December 10, 2023

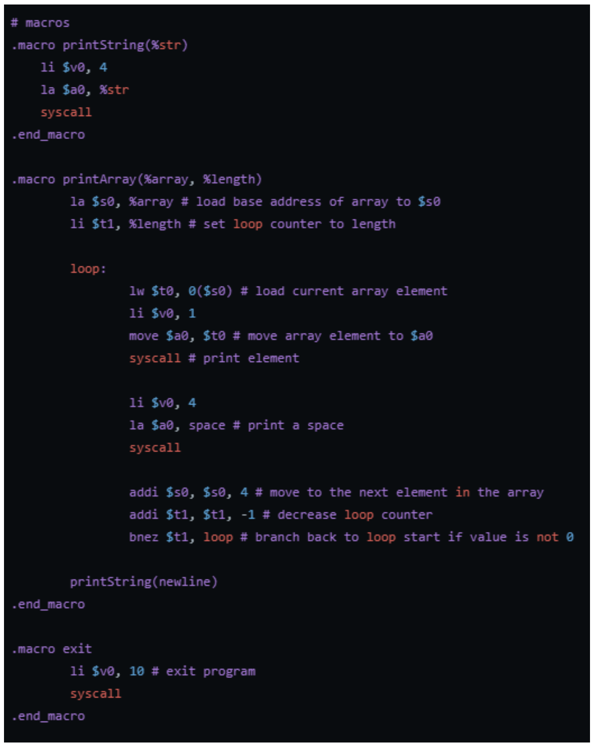
Final Project Report Paper

The project that our group chose was quick-sort. The reason that we chose quick sort was because we wanted to see how different it was to program an algorithm like it compared to how we would implement it in higher-level languages that we are more accustomed to. The quick sort algorithm stood out to us the most because of how it sorts in place rather than needing to create multiple sub-arrays while also being one of the most used sorting arrays in practice. The goal for this quick sort array was to make sure it could correctly sort the array while making use of macros as much as we could.

The Quick sort algorithm is an extremely effective algorithm that is able to correctly sort large amounts of data quickly. The benefit of using quick sorting is that we will not have to make a lot of sub-arrays; rather we just need to be able to move the elements into partitions and sort until we no longer need to. We perform this operation by starting at a pivot point and moving the elements of the array larger than our pivot to the back end while the smaller values get moved to the front. Once we finish separating the array into these two partitions within the same array, we recall the function to then sort each partition until there is one element as we cannot make a new partition at which point the array should be sorted. This method does mean that the quick sort algorithm requires recursion which makes it a bit more complex and also means small mistakes will lead to much less effective results without stopping the algorithm from working properly. The challenge of making this algorithm work in a language like assembly compared to a more advanced language like C++, Java or even Python will be interesting as we will need to be more deliberate with where we store and how we call on the data that we use.

The implementation of a quick sort algorithm is fairly similar in each case as most of the time you create different utility functions that then all assist you in being able to complete the algorithm. The first is having the partition function in which we figure out what our pivot is and where it relates to the rest of the elements of the array. We do this by checking if each element is larger or smaller, moving it to the right or left respectively until the pivot is all sorted. As so, in concept, recursion could be used to apply the division that happens into subarrays, but in addition to that, a stack can be used to keep track of subarrays as well to accomplish a similar goal. From here we have the main quicksort recursive function in which we pass the argument of the subarrays then have them made into partitions and sorted to which it will continue until the array is completely sorted.

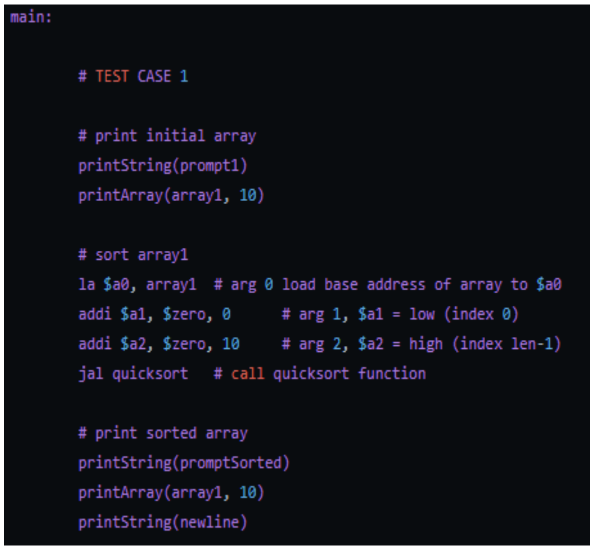
The layout of our group's quicksort algorithm program is a little bit different as we start with establishing a couple of macros in order to save time and for our program. The macros that were created for the program were *printString*, *printArray,* and the *exit* macro to exit the program. These macros are self-explanatory as the *printString* macro is used to print either the message about the array first before sorting or how the array looks after it is put through the quick sort algorithm. The next macro that is created and used is the *printArray* macro which again simplifies to print the array and is done once for the initial array and again for the array



```
# macros
.macro printString(%str)
    li $v0, 4
    la $a0, %str
    syscall
.end_macro

.macro printArray(%array, %length)
        la $s0, %array # load base address of array to $s0
        li $t1, %length # set loop counter to length

        loop:
                lw $t0, 0($s0) # load current array element
                li $v0, 1
                move $a0, $t0 # move array element to $a0
                syscall # print element

                li $v0, 4
                la $a0, space # print a space
                syscall

                addi $s0, $s0, 4 # move to the next element in the array
                addi $t1, $t1, -1 # decrease loop counter
                bnez $t1, loop # branch back to loop start if value is not 0

        printString(newline)
.end_macro

.macro exit
        li $v0, 10 # exit program
        syscall
.end_macro
```

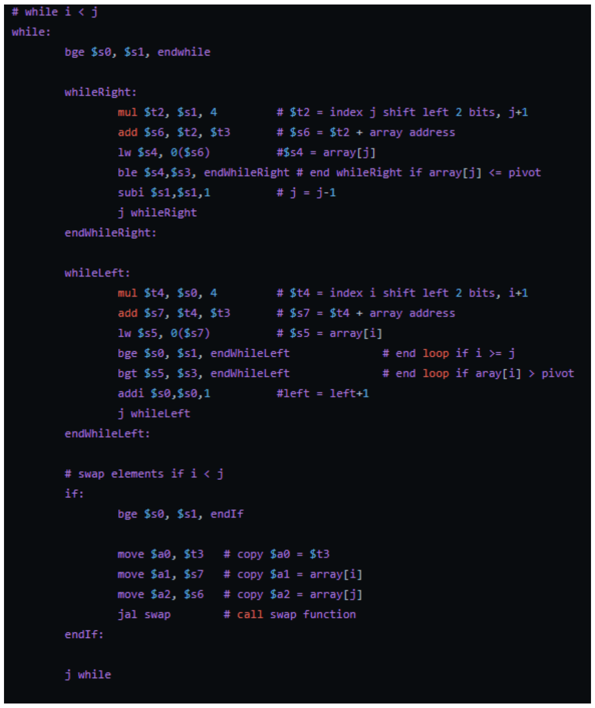
after it is gone through the quicksort. And the final is just the *exit* macro to end the program. These macros allow the code to be much cleaner as we use it multiple times and rather than having to write the code each time we would want to complete that action. The next portion is the .data portion of the code in which we declare all of the strings we will use as the prompt which will be called on with the *printString* macro. The most important item in the data portion is that we store our arrays here for all test cases. In our case, we have 3 different arrays that all get sorted through for 3 different test cases. That is the beginning of the program as it is just all setup that will be used later in order to assist a more fluid and structured approach.

The next portion of the program is the main function which we can separate into smaller chunks that deal with different aspects of the quick sort algorithm. The first portion is simply calling the quicksort function on the different test cases. In the first case, you can see that we call on the *printString* function in order to print the first prompt. This is to help show that the first array is printed in the array before the quick sort algorithm takes place. We then print the array by calling on the *printArray* macro using array1, and 10 being its length as an argument. This is again just to show



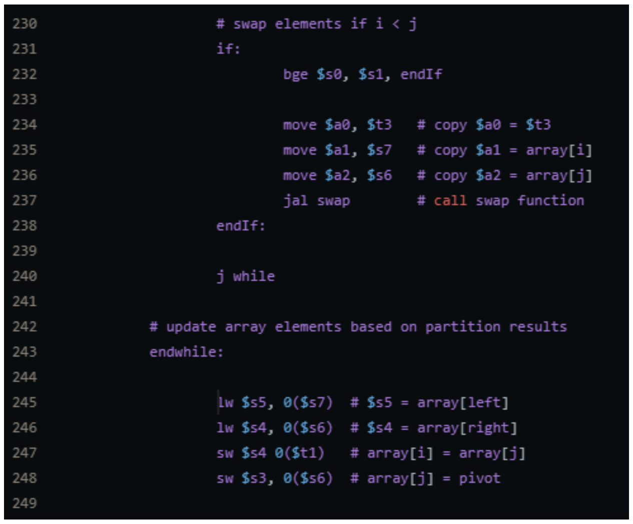
```
main:

        # TEST CASE 1

        # print initial array
        printString(prompt1)
        printArray(array1, 10)

        # sort array1
        la $a0, array1  # arg 0 load base address of array to $a0
        addi $a1, $zero, 0      # arg 1, $a1 = low (index 0)
        addi $a2, $zero, 10     # arg 2, $a2 = high (index len-1)
        jal quicksort   # call quicksort function

        # print sorted array
        printString(promptSorted)
        printArray(array1, 10)
        printString(newline)
```

how the unsorted array looked before the quicksort. Now we start the process of sorting the array by loading the address of the array and storing it for later. We also figured out what our highs and lows are and stored them in a1 and a2 respectively. Finally, we call on the quicksort function to continue to the next step of the program. With multiple functions and arguments that use the array in order to perform the quick sort algorithm, we have implemented a stack ADT. In the first part of the quick sort algorithm, we have allocated spaces to assign 4 variables into the stack, storing the variables of the lower and higher index of the array. When the sorting process is finished, we deallocated the stack memory to avoid memory access violations or any other unexpected behavior. Here come some of the challenges we have faced; using the stack in our program required a solid understanding of data structures and how they can be used to manage numbers in an array with efficiency. Using the stack in MIPS requires a structured program with a limited set of instructions with a smaller range of registers to use, which poses a great challenge in tracking the current location of stack pointers. If the current stack pointer malfunctions, it becomes impossible to correctly store numbers on the stack, and even harder to find them. However, to avoid these events beforehand, we have managed to initialize the stack and free up its space after the sorting process to correctly use the stack in our program. From here we move to the next portion of the program, which is where we figure out what the partition is. We accomplish this by calling on our partition function that stores our



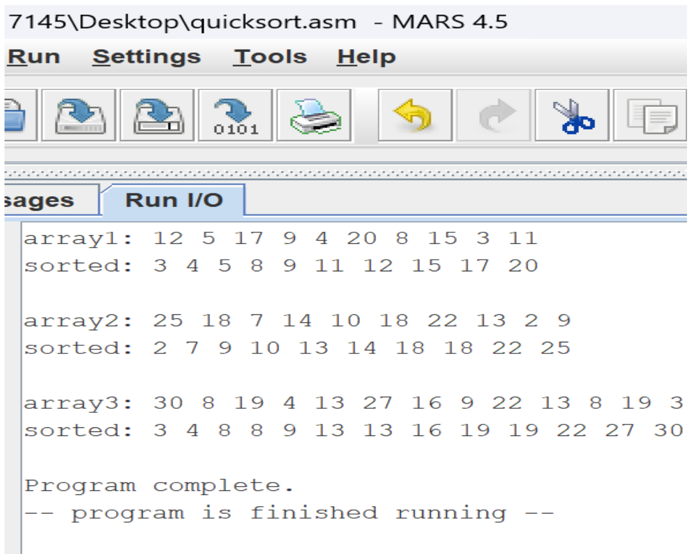
```
# while i < j
while:
	bge $s0, $s1, endwhile

	whileRight:
		mul $t2, $s1, 4		# $t2 = index j shift left 2 bits, j+1
		add $s6, $t2, $t3	# $s6 = $t2 + array address
		lw $s4, 0($s6)		#$s4 = array[j]
		ble $s4,$s3, endWhileRight # end whileRight if array[j] <= pivot
		subi $s1,$s1,1		# j = j-1
		j whileRight
	endWhileRight:

	whileLeft:
		mul $t4, $s0, 4		# $t4 = index i shift left 2 bits, i+1
		add $s7, $t4, $t3	# $s7 = $t4 + array address
		lw $s5, 0($s7)		# $s5 = array[i]
		bge $s0, $s1, endWhileLeft		# end loop if i >= j
		bgt $s5, $s3, endWhileLeft		# end loop if aray[i] > pivot
		addi $s0,$s0,1		#left = left+1
		j whileLeft
	endWhileLeft:

	# swap elements if i < j
	if:
		bge $s0, $s1, endIf

		move $a0, $t3	# copy $a0 = $t3
		move $a1, $s7	# copy $a1 = array[i]
		move $a2, $s6	# copy $a2 = array[j]
		jal swap	# call swap function
	endIf:

	j while
```

high and low array, returns its address in another stack and then goes into a while loop. This while loop is where the partition is found based on the pivot that is chosen in our case; it is our low or first element of the array and the while loop then checks the other elements of the array and if the current value is smaller than or equal to the pivot point in which case it would then taken out of this inner loop and moved to the next inner loop checking if the values are smaller than or equal to the pivot. Once both of these inner loops have been gone through the next part of the partition function which will swap the low and high with each other if the low is smaller than the high. It accomplishes this by calling on the swap function that is declared outside of the partition function which again uses the stack to save different positions and uses them to switch the value of the array at position of j and i. Once the partition is made we then move to the next function which is saving the position of the left sub-array, right subarray, and the pivot. After that, we then free up the space from the stack and return the array back with the pivot to which we will then start sorting each sub-partition starting with the left//lesser than side of the array. In order to accomplish this we just pass the array through the quicksort function with a change of parameters of lower to the pivot -1, the process is then repeated until the amount of elements in the left partition gets to a single element, and in that case, the elements on the left are all sorted. From here, the program will slowly start to complete all the right subarrays that are created until we reach the original pivot and will then work its way down the original subarray on the right with the larger elements. This method relies completely on recursion so that the partitions can be



```
		# swap elements if i < j
		if:
			bge $s0, $s1, endIf

			move $a0, $t3   # copy $a0 = $t3
			move $a1, $s7   # copy $a1 = array[i]
			move $a2, $s6   # copy $a2 = array[j]
			jal swap        # call swap function
		endIf:

		j while

	# update array elements based on partition results
	endwhile:

		lw $s5, 0($s7)  # $s5 = array[left]
		lw $s4, 0($s6)  # $s4 = array[right]
		sw $s4 0($t1)   # array[i] = array[j]
		sw $s3, 0($s6)  # array[j] = pivot

```

made to sort the array while being able to cover the most ground with the number of elements of the array. Once the quicksort has been completely completed we then go back to the #test case 1 segment of the program where the message for the new array is printed with the *printString* macro followed by the printArray macro that prints the same array as before except now being correctly sorted from least to greatest value. The program will then run for the next two cases in which it will follow the same steps as before running through the loops and recursion until the next 2 sets of array elements are also sorted and printed out. Once complete the program will then move into its final phase where it will call on the *printString* macro one last time to tell the user the program is being completed before calling the exit macro to end and exit the program. The desired result is seen down below with all 3 arrays being printed in their unsorted form followed by their sorted form using again the same array not using any new arrays and just moving the elements within the same array with recursion and the stack.



```
array1: 12 5 17 9 4 20 8 15 3 11
sorted: 3 4 5 8 9 11 12 15 17 20

array2: 25 18 7 14 10 18 22 13 2 9
sorted: 2 7 9 10 13 14 18 18 22 25

array3: 30 8 19 4 13 27 16 9 22 13 8 19 3
sorted: 3 4 8 8 9 13 13 16 19 19 22 27 30

Program complete.
-- program is finished running --
```

Throughout the process of creating quicksort, there were several challenges of edge cases that led to significant errors. To create a more robust and maintainable program, we have conducted three different test cases: test case one involves an array of numbers with length 10, and contains unique numbers; test case two involves an array of numbers with length 10 but with duplicate numbers; test case three involves an array of numbers with length 13, also with duplicate numbers. The purpose of different lengths and duplicate numbers for every test case was to observe whether the program can handle even and odd lengths of an array and check if duplicate numbers are sorted correctly. To test each case, we have used the initially implemented macros to print the string prompt, and the array itself. We did this to solve a different challenge which was to make sure we did not over-clutter or even have too many repeat functions in our code allowing our structure and overall efficiency to be more reliant on the sorting algorithm itself. Conducting multiple test cases under different circumstances has helped to enhance the program with varying input sizes and memory management of the implementation. Since error handling is crucial in programming, we have successfully tested different scenarios to ensure that the program performs quick sorting correctly. Most of the challenges about how to start or how to structure the program were solved by looking at the GeeksforGeeks website and their page about the quicksort algorithm. This was extremely helpful during the process of trying to deconstruct the process and translating it into the MIPs assembly.

During this project, a lot was learned, understood, and appreciated. The first and most important thing is the understanding of the quicksort algorithm. In order to complete the program there needed to be a much better understanding of what exactly we needed each part of the program to accomplish and not only what data was needed, utilized, and stored. Unlike higher-level languages, like Java, we were not able to simply call on the array we needed to store the register for the array so that when we needed it, it would be ready for us to call on. This did not just apply to the array itself but the data inside of it as well. Through this understanding, the relationship between all elements that were being used was also crucial including using the stack to help temporarily store data that would be needed later rather than assigning it a permanent place until it was written over. The appreciation of languages like Java and Python as assembly again is a very unmanaged program like C which made the process of making the program and deconstructing the logic and syntax to then create a functioning algorithm. These languages are a lot smaller than a program in MIPS assembly because we do not need to tell the computer explicitly what needs to be done and where values should be stored in its memory. These are some of the bigger takeaways that came out of this project about creating a quick sort algorithm in MIPS assembly that used recursion.

This project overall taught us a lot about how computers take in and read instructions while giving us an insight into the closest things we will have to machine code. Using this we also learned and understood the quick sort algorithm more as a whole and its inner workings as we had to be more specific with what we told the program to complete.

Works Cited

“Quicksort - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, GeeksforGeeks, 16 Oct. 2023, www.geeksforgeeks.org/quick-sort/.