ECE 30
Introduction to Computer Engineering
Programming Project: Spring 2023
Quick Sort with Lomuto Partitioning

May 2, 2023

Project TA: And Ka @ucsd.edu)

1 Project D

Write a program to *cksort* algorithm on a list of integers using Lomuto partitioning scheme. read an unsorted array and sort (or partially sort, possibly) it using a Quicksort algorithm. Quicksort uses a recursive partition function.

2 Quicksort Algorithm

Quicksort is a divide-and-conquer algorithm for sorting a list by recursively sorting sub-lists. The steps in quicksort are:

1. Pick an element, called pivot, from the list
2. Partition the list such that after partitioning
   - All the elements that are less than the pivot come before the pivot (left partition)
   - All the elements that are greater than the pivot come after the pivot element (right partition)
   - The pivot element is in its correct position *after* sorting

   There are various methods for partitioning. In this project we will use Lomuto partitioning scheme as described below.

3. Recursively, apply steps 1– 2 separately on the left and right partitions

In this project, we will read a list, which is denoted as 'a' in the template, consisting of long integers. The memory address of the first element of the list will be in X0 register. The addresses of the following values will be X0+8, X0+16,X0+24,... Our program will also take *low* and *high* index values, which specify the boundaries for our algorithm to implement sorting.

Example: Let

$$arr = [4, 3, 5, 2, 100, 10, 8, 7]$$

,

If $low = 0$ and $high = 7$, then the sorted array would be

$$arr = [2, 3, 4, 5, 7, 8, 10, 100]$$

As *low* and *high* values coincide with the first and the the last indices, respectively, our program will sort the whole array.

However, in our project, our algorithm is also expected sort only a part of an array, in which the left-most and the right-most boundaries are given by *low* and *high* values, and keep the other parts unsorted. As another example,

$$[4, 3, 5, 2, 100, 10, 8, 7]$$

This time, suppose $low = 2$ and $high = 5$, which indicates that we only need to sort the values coming from indices {2,3,4,5}. Note that the values attained by *low* and *high* are also included in the sorting. The (partially) sorted would be

$$[4, 3, 2, 5, 10, 100, 8, 7]$$

Base case: If the values of *low* and *high* are equal to each other, then the algorithm is not supposed to change anything in the given array.

Pivot Selection: There are various methods for choosing the pivot element, such as: (1) choosing the first element, (2) choosing a random element, and (3) choosing the middle element. For this project, we will choose the pivot as the last element of our array (or sub-array).

Lomuto Partitioning: As the pivot is chosen as the last element, we can apply the partition algorithm. For this project, we will implement one of the first partitioning algorithms invented for Quicksort, namely Lomuto partitioning. We will initialize a temporary_pivot_index (TPI), say i, which would count the number of elements less than or equal to the pivot. As you can easily infer, the correct index for the pivot value in the sorted array is (TPI + 1) after we apply the partition function. Secondly, we will have another index,say j, to trace the values in the actual array. Let the name be current_index which is abbreviated as CI. Different from other partitioning schemes, in Lomuto partitioning, the pivot element is kept in the last index until the end of each iteration and it is placed in its correct index at the end.

To understand Lomuto partitioning, or the essence of partitioning in general, one could check the useful Wikipedia page on Quicksort (in the pseudocode for Lomuto partitioning on that page, the index i and the index j coincide with our assignment in the previous paragraph) and watch this video illustration of Lomuto partitioning on Youtube.

As opposed to the Lomuto partitoning pseudocode presented on the Wiki page, we will implement our partition function recursively instead of using a for loop.

3 Implementation

In this project, you have to write the following functions, namely ***swap***, ***partition*** and ***quicksort*** to implement the quicksort algorithm in LEGv8 assembly language. Some details to note are:

- Use the procedures prototype as mentioned below and given to you in the template. Don't change the registers or the arguments passed to the procedures or the values returned.
- Follow the "Procedure Call Convention" for calling procedures, passing registers and managing the stack. The procedures should not make any assumptions about the implementation of other procedures.

- We expect your code to be well-commented. Each instruction should be commented with a meaningful description of the operation. For example, this comment is bad as it tells you nothing:

```
// x1 ge
sub x1,
```

A good comm e meaning behind the instruction. A better example would be:

```
// Initialize loop counter x1 to n-1
sub x1, x2, #1
```


3.1 Function 1: swap(a, b)

Swaps two values pointed by a and b. It is a very generic swapping operation that you may have encountered in C language.

3.1.1 Parameters

- X0: the address of the first value
- X1: the address of the second value

3.1.2 Return Value

- This function does not return anything.

3.1.3 Pseudo-code

Swapping two values in the addresses X0 and X1. (note that you will need a temporary register).

3.1.4 Examples

Inputs:

- X0=100 (and suppose the value stored in address 100 is a)
- X1=108 (and suppose the value stored in address 108 is b)

When exiting:

- The value stored in address X0=100 is b
- The value stored in address X1=108 is a

3.2 Function 2: partition(a, low, high, TPI, CI)

This function must separate the given (sub)list into two parts based on the pivot value, such that all elements that are less than the pivot lie in the left partition and all the elements that are greater than the pivot lie in the right partition. (Elements equal to the pivot are assigned to either parts based on your implementation's indices.) You must return the final index of the pivot.

You must implement the partition function as follows: (1) start with the last element as pivot, (2) collecting each element greater than (or equal, possibly) the pivot to the right-side and each element less than (or equal, possibly) the pivot to the left-side of the array, this step is literally how Lomuto partition works, (3) place the pivot between the two partitions.

3.2.1 Parameters

- X0: Starting address of a (sub)list (corresponding to `a`.)
- X1: *low* index
- X2: *high* index
- X3: TPI
- X4: CI

3.2.2 Return Value

- X0: The index of the pivot element

3.2.3 Pseudo-code

```
function PARTITION(a, low, high, TPI, CI)
    pivot ← a[high]
    i ← TPI
    j ← CI
    if j==high then
        SWAP(a[i+1],a[high])
        return i+1
    end if
    if a[j]<= pivot then
        i ← i + 1
        SWAP(a[i],a[j])
    end if
    return PARTITION(a, low, high, i, j+1)
end function
```

3.2.4 Examples

1.
Inputs:

- a: 4, 2, 7, 3, 1, 6, 9, 0, 8
- low: 0

- high: 8
- TPI: $low - 1 = -1$
- CI: $low = 0$

Returns:

- 7

When exiting:

- a: 4, 2, 7, 3, 1, 2.

Inputs:

- a: 100, -1, 5, 3, 7, 2, 6, 1, 4, 737
- low: 2
- high: 8
- TPI: $low - 1 = 1$
- CI: $low = 2$

Returns:

- 5

When exiting:

- a: 100, -1, 3, 2, 1, 4, 6, 7, 5, 737

3.3 quicksort(a, low, high)

This is the main function to recursively sort the given list. Unless there is at most one element in the sub-list, this function will call the partition function to partition the list, which will return the final position for the pivot element; then it will recursively call quicksort on the left and right partitions separately.

3.3.1 Parameters

- X0: Starting address of a (sub)list
- X1: *low* index
- X2: *high* index

3.3.2 Return Value

- This function does not return anything.

3.3.3 Pseudo-code

```
function QUICKSORT(a, low, high)
    if low < high then
        pivot_position ← PARTITION(a, low, high, low- 1, low)
        QUICKSORT(a, low, pivot_position-1)
        QUICKSORT(a, pivot_position + 1],pivot_position+1, high)
    end if
end function
```

3.3.4 Examples

Inputs:

- a: 100, -1, 5, 3, 7, 2, 6, 1, 4, 737
- low: 2
- high: 8

Result:

- a: 100, -1, 1, 2, 3, 4, 5, 6, 7, 737

4 Instructions

- You are encouraged to test the example arrays given above with your program. Another test array is given as [8, -5, 8, 4, 6, 100, 746, -16]. You can have the related data file named as 'Test_data.txt' on Canvas > Files > Project S23. Make sure that your program works properly with any suitable values of *low* and *high*, i.e. $low \leq high$.
- You must submit your solution by emailing (to ecehjty@gamil.com) a completed file ABC_XYZ_2023_project.s, where ABC and XYZ are the @ucsd.edu of each student. For example, if Sherlock Holmes and John Watson were working together on a submission, the file name would be: sholmes_jwatson_2023_project.s
- Fill the names and PID of each student in the file.
- Each project team contributes their own unique code – no copying, cheating, or hiring help. We will check the programs against each other using automated tools. These tools are VERY effective, and you WILL get caught.
- Similarly, using ChatGPT to come up with the code is also prohibited and will be considered as cheating.
- Please start this project early.
- Try to test the behavior of each function independently, rather than trying to code all of them at once. It will make debugging far easier.