Computation Structures
Project 1
Sorting algorithms in β-assembly

November 22, 2021

General information

- **Deadline: December 17, 2021, 23:59** (There will be a penalty for late submission).
- Project must be done by **teams of two students**.
- Questions will no longer be answered 24 hours before the deadline.
- English is strongly encouraged.
- Contact: `gaulthier.gain@uliege.be`, Office 1.8 (B37).

1 Sorting Algorithms

The goal of this project is to make you more familiar with the β-assembly language by getting your hands dirty and writing code. In this project, you will implement two sorting algorithms: Bubble sort and Quicksort.

1.1 Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the previous element with the current element if the previous element is bigger. If no swaps occur, the array is sorted in ascending order. If a swap does occur, another pass is performed. Figure 1 represents an example of the Bubble sort algorithm applied on an unsorted array.

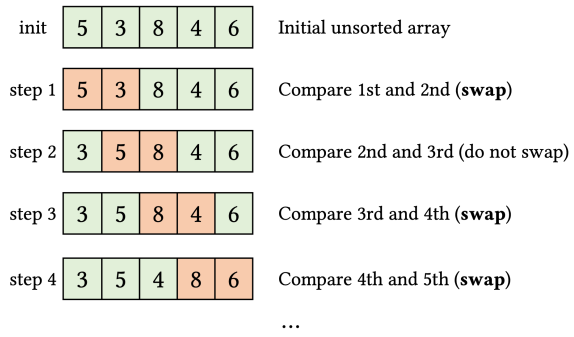

Figure 1: Example of the Bubble sort algorithm (incomplete)

1.2 Quicksort

Quicksort is an in-place sorting algorithm based on the following recursive procedure:

1. **Base case:** if the array size is less than or equal to 1, the array is already sorted and nothing has to be done.
2. **Recursive case:**
   (a) Select a pivot value in the array (see below for the selection strategy);
   (b) Partition the array around this pivot value (see an example of partitioning in Figure 2);
   (c) Sort recursively the sub-arrays to the left and to the right of the pivot value using Quicksort.

The pivot can be selected using various strategies:

1. Picking a specific element (e.g., the first of the array);
2. Picking an element at random;
3. Picking the median element among three (e.g. the first, the element at the middle of the array and the last element)

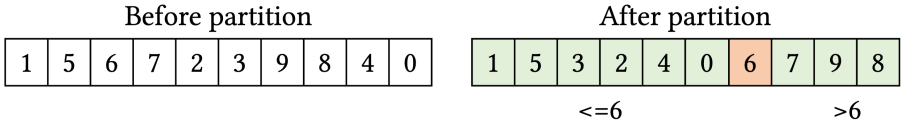

Figure 2: Example of array partitioning. Chosen pivot is value 6 at index 2 in the initial array.

For more information about sorting algorithms, you can check online resources:

1. Your course on data structures and algorithms (e.g., INFO0902-1);
2. Explanation and interactive illustration: `https://visualgo.net/en/sorting`;
3. Wikipedia: `https://en.wikipedia.org/wiki/Bubblesort`, `https://en.wikipedia.org/wiki/Quicksort`;

Bonus

You can choose any pivot selection strategy for your implementation, the simpler being picking the first element of the array. However, implementing the median of three strategy will earn you a bonus. Using this strategy induces the modifications in the C implementation presented in Listing 1.

```c
/*
 * Return the address of the median of the values located at the given
 * addresses.
 * @param first int* Address of the first element
 * @param second int* Address of the second element
 * @param third int* Address of the third element
 * @return int* Address of the median element
 */
int* median_of_three(int* first, int* second, int* third) {

    if (*first < *second) {
        if (*third <= *first) {
            return first;
        } else if (*second < *third) {
            return second;
        } else {
            return third;
    } else {
        if (*first <= *third) {
            return first;
        } else if (*second < *third) {
            return third;
        } else {
            return second;
        }
    }
}

void quicksort(int* array , size_t size) {
    // ...
    int* pivot_addr = median_of_three(
        array ,
        array + (size / 2) - 1, array + size - 1
    );
    size_t pivot = pivot_addr - array;
    // ...
}
```

Listing 1: Modifications to the C implementation of Quicksort for using the median of three pivot selection strategy.

2 Objectives

Your task is to implement two procedures:

1. a `quicksort` procedure in a file called `quicksort.asm`. This procedure should be callable using a label `quicksort`;
2. a `bubblesort` procedure in a file called `bubblesort.asm`. This procedure should be callable using a label `bubblesort`.

Each procedure takes the two following parameters:

1. `array`: the memory address of the first element of the array of signed (32 bits) integers to sort
2. `size` : the size of the array (size ≥ 0)

There will be a penalty for not conforming to this interface. Whatever you write in your `quicksort.asm` and `bubblesort.asm` files, your procedures should be callable using the example code shown in Listing 2.

```
|; Reg[R1]:  address of array[0]
|; Reg[R2]:  size of the array
PUSH(R1)
PUSH(R2)
CALL(bubblesort) |; or CALL(quicksort)
|; CALL(bubblesort) stands for BEQ(R31, bubblesort, LP)
DEALLOCATE(2)
```

Listing 2: Example of call to the quicksort procedure

3 Structure

- `beta.uasm`: it is the definition of the β-assembly. You can check this file to see which macros you can use in your own code.
- `main.asm`: this file contains the main program that will be used to test your procedure. It does the following:
  1. It reserves some memory for an array of size n;
  2. It initializes it with increasing values ranging from 1 to n;
  3. It shuffles the array;
  4. It calls the Quicksort and Bubble sort procedures that you must implement.
- `util.asm`: this file contains some code needed by the `main.asm` file (for filling the array and shuffling it).

You are provided with two C files: `bubblesort.c` and `quicksort.c`. These two files provide a C implementation of the two sorting algorithms (Bubble sort and Quicksort). You can use them as a basis for your implementation, but it is not mandatory to strictly stick to the details in those algorithms for your assembly implementation. However, it is expected that you implement the *recursive* Quicksort procedure presented in Section 1.2.

4 Additional guidelines

4.1 Practical organization

In order to learn β-assembly effectively, **this project will be done by teams of two students**. A report of **maximum one page** must be provided. In this report, you must briefly explain your implementation.

You will include your completed `bubblesort.asm`, `quicksort.asm` and your report (PDF only) in a ZIP archive named `sXXXXXX_NAME1_sYYYYYY_NAME2.zip` where (`sXXXXXX`, `NAME1`) and (`sYYYYYY`, `NAME2`) are the student `ID` and family name in uppercase of the two team members. Naming your files differently or submitting other files **will result in a penalty**.

Submit your archive to the **Montefiore Submission Platform**[1], after having created an account if necessary. If you encounter any problem with the platform, please contact the teaching assistant. However problems that unexpectedly and mysteriously appear five minutes before the deadline will not be considered. **Do not send your work by e-mail; it will not be read**.

Plagiarism is not allowed and will be severely punished. Any detected attempt will result in the grade of 0/20 for the full course for all who participated to the fraud. Asking questions on the Forum (on eCampus), and helping your fellow students on the Forum is allowed (to a reasonable extent).

4.2 Code guidelines

Choose a coding style and stick to it. You are advised to use the coding style used in `main.asm`. The goal here is not to write code which is as compact and efficient as possible, but to learn the concepts of β-assembly. However, your code should not be unreasonably long and inefficient: minimize the number of registers you use in your recursive procedures, as it impacts the growth rate of your stack.

4.3 Documentation

One of the challenges when writing assembly code is to write a program which is relatively easy to understand. Thus, the second most important element taken into account for your grade (after correctness) will be your code's readability. Use comments extensively (your comments can be larger than your code), but do not be verbose: explain the non obvious, not the immediately apparent.

In addition to the comments written alongside your code, all your procedures should be documented using pre-/post-conditions:

- Arguments have to be properly defined
- Any return value must be documented
- Any side effect (e.g. modification of the dynamic memory) must be documented

You are free (and advised) to use macros to reduce the redundancy in your code. Those macros should be documented like your procedures (arguments and operations performed).

Good luck and have fun !

[1] `http://submit.montefiore.ulg.ac.be/`