CS2102 Analysis and Design of Algorithms **26 de abril de 2019**

Quicksort

Bryan Gonzales Vega University of Engineering and Technology

I. Instructions

Complete the following questions and attach your source code, LaTeX, and other tools (Dockerfile, images, etc) you required to complete the problems using your GitHub Classroom repository.

II. Analysis

We already prove that Quicksort algorithm runs at $\Theta(nlogn)$ time for the average and best case. Nevertheless for the worst case it runs at $\Theta(n^2)$. Although we have a solid understanding of the common implementation there are still some improvements we can do to it. The following exercises give you some hints about how you can improve the Quicksort algorithm. For all the exercises run some simulations to test out your answers.

1. Can you find the minimum number of elements n for which the Quicksort algorithm always runs faster? If so implement it and simulate some scenarios where you can see the improvement.

   Here I am running test for a array of integer randomly generated (non repeating number)
   I run approximately 20 times the simulations to get the average.
   Comparing Quicksort with Insertionsort we can detect that the minimun number of elements n for which the Quicksort algorithm will run faster is approximately 76, so $n = 76$.
   Mergesort is an extra algorithm to compare.
   (Images are down there, find them)
   (If you are not happy about the graphics, you can try the simulations)

2. Besides time a recursive algorithm consumes memory. Explain the memory consumption of a recursive algorithm .

   This memory usage will depend on a lot of aspects that affect your code and application. When a recursion occurs, you cause a function to call itself. During that calling, a stack keeps on filling with the local variables that recurred each and every time. This may lead to the stack being full reaching a stack overflow error.

3. What is tail recursion? Is it faster than the regular recursion ? Try it out with the Quicksort algorithm. Does it run faster?

   Tail recursion is a way to optimize a recursive algorithm, it attacks the part of the recursion when it creates a new environment in the stack (local variables, parameters, etc.). Tail recursion reuses the same environment every time a recursion call is made and so the space occupied in the stack is constant, preventing the stack overflow issue.

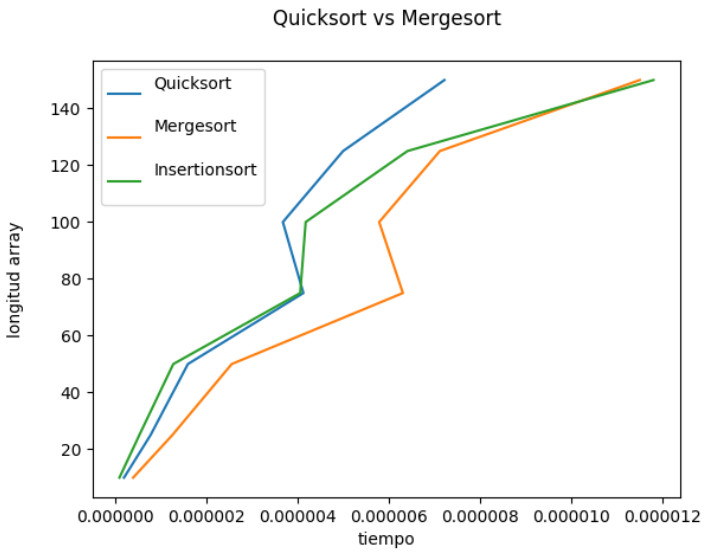

Figura 1: comparing 3 algorithms

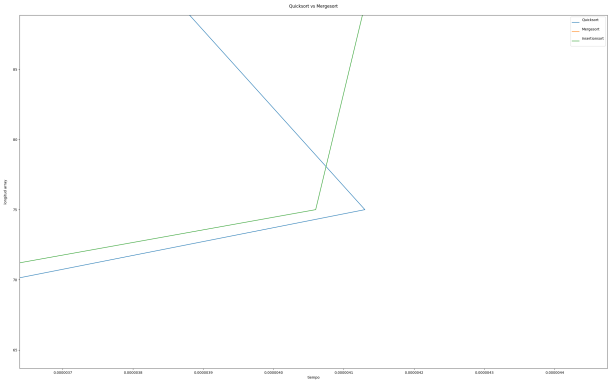

Figura 2: closer look

4. How can the Quicksort algorithm be implemented in a parallel way ? Propose your pseudo-code for that task and sketch the execution time.

   I'll do something like this:

Algorithm 1 Paralel Quicksort

1: **procedure** MyProcedure
2: $part = ArraySize/NUMHILOS$
3: $core = 0$
4: **for** $i = 0, i < part$ **do**$QuickSort(A, i * part, i * part * core)$, $core = core + 1$
5:

5. Propose one improvement to the Quicksort algorithm in terms of space or execution time.

   An improvement will be a "Tail Recursion Randomized Quicksort", the Random part will improve the execution time and the Tail Recursion optimization will improve the space of usage (avoiding possible stack overflow errors)

6. Quicksort requires a pivot element that could be chosen in different ways like: i) always the first element, ii) the last element, iii) the middle one or iv) a random element. Is there any difference in terms of performance ? If so specify for which cases.

   The an swear is It depends. it depends on the array, its ordered probability will tell us which position to use. For example, it will be times when the right half of the array will be a little sorted, so choosing a pivot to the left can help to order the array easier . Like that, any cases can appear and it will be a faster way to sort the array (depending on the pivot position). However , as we don't know how sorted will be the array the best option is to choose a random position.

III. Inversions

In the resources folder you will find a text file with 100,000 integers between 1 and 100,000 (inclusive) in some order, with no integer repeated. Your task now is to compute the number of inversions of the given file. Compare the execution times of the naive and divide and conquer implementations. As an extra you can compare both implementations with some other strategy. Write the number of inversions and plot your results.

Here the solution.