Lab 6 – Participation Activity 8

Objective:

- The purpose of this lab is to introduce us to the idea of *benchmarking* and to familiarize ourselves with some of the tools used to benchmark code.

Lab 6 Submission – Participation Activity 8

- As part of Lab 6, you will need to submit your Participation Activity 8 (the last two pages of Lab 6):
    - Complete as much of the Lab 6 as you can during your lab session.
    - Write your name, student number and circle your section (D100 or D200).
    - Save your Participation Activity 8 (the last two pages of Lab 6) as a pdf and upload it onto Crowdmark (the assessment is called *Lab 6 – Participation Activity 8*)

  Participation Activity 8 Deadline: 12:30pm on Tuesday March 28.

- You also need to submit your **completed** Lab 6 - Participation Activity 8 (the last two pages of Lab 6) as part of our Assignment 8. When marking Assignment 8, the TAs will be looking at the correctness of your data sets and your graph. Therefore, make sure you follow all the steps of Lab 6.

What is *Code Benchmarking*

Wikipedia says that "in computing, a **benchmark** is the act of running a computer program in order to assess its relative **performance**, normally by running a number of standard tests and trials against it."

In this lab, we shall benchmark our program to assess its time performance as it is executing a few tasks on an array of various sizes. More specifically, we shall look at how long our program takes to …

- randomize an array of various sizes, and
- quicksort this array, using two different partitioning algorithms.

Part 1: Benchmarking a Loop

- Login to a CSIL machine in the Linux environment, download `Lab6-Files.zip` and unpack it into `sfuhome/CMPT295`. Change directory into `Lab6` where you will find the usual `makefile` and a collection of source files.

- Start by opening `main.c` in your favourite editor. The program's main purpose is to initialize an array of N integers, in the range [0, (N-1)/100], in a randomly permuted order.
- When you build the executable and run it, it will display the time it took (in microseconds (µs)) for the main loop to complete. This is done via the `getrusage()` system call, which asks the Linux operating system how much processor time has elapsed since the beginning of the program. Both calls to `getrusage()` record timing information in their parameters `start` and `end`.
- Important step: To know exactly how this timing information is processed, have a look at the `printf` statement in `main.c` where you will find both parameters `start` and `end` being used. These parameters are both of type `struct rusage`. Check out the definition of this `struct` at this web page: http://man7.org/linux/man-pages/man2/getrusage.2.html). As you can see, the `struct rusage` makes use of the `struct timeval` (named `ru_utime)` in order to keep track of timing information.

  You now need to see what `struct timeval` contains in order to have the complete picture. To do so, have a look at this web site:
  https://manpages.ubuntu.com/manpages/impish/man3/timeval.3bsd.html
  where you discover (under the Description section of this web site) that this structure has two fields. Read all about these two fields as this knowledge will help you later on in this lab when you are debugging the code.
- An alternative to using `getrusage()` is to use the Linux `time` command. To try it, run `time ./x`. It should report the amount of ***user*** time for the entire program in seconds, to the nearest millisecond (ms). There should be a fairly close agreement between the number you just obtained when you executed the `time` command above and the number of microseconds used to execute the main loop of `main.c`, which was reported by your executable `x` when you ran the program at the start of this Part 1. Why do you think these two numbers are not exactly the same?
- In this part of Lab 6, you are going to time the main loop using different values of N. Initially, N is 1000000 (10^6). Run the program 5 times, and record the time in µs for each in the *Main Loop* table of the data sheet found at the end of this lab.
- Next, change the value of N to 2000000 (2×10^6). Run the program 5 times, and record the times in row 2 of the same table.
- Repeat for N equal to 4000000, 8000000, 16000000 and 32000000. Again run the program 5 times and record your data.
- Does the data make sense? To answer this question, we need to have an idea of what kind of results we are expecting. We are expecting the running time to roughly double for each doubling of N. Why? Because this main loop has a time complexity of O(N). This seems to occur except for the last value of N or two. This is because there is a bug in `main.c`, more specifically in how the times are computed.

Hint: have a look at the `printf` statement and the two fields of `struct timeval`. Fix the bug and run your program 5 times for the values of N that produced buggy times. Replace the buggy times in the data sheet with these new times.

- Calculate the average of each set of 5 numbers and report each number in the column marked μ.

Congratulations! You have just benchmarked the main loop of `main.c`, where the elements of the array `A` are randomized. You should observe a roughly linear progression. It isn't exactly linear, however. Do you have any suspicions as to why?

Part 2: Benchmarking a Function

Next, you will benchmark two different versions of quicksort, each quicksort uses a different partitioning algorithm.

- Adjust your `main.c` so that it will report the time, in µs, of each of the quicksort functions. In doing so, do not delete or comment out the main loop since you still need a randomized array to benchmark each quicksort function.

  What you need to do is move both calls to `getrusage()` so they wrap around the appropriate call to quicksort function and uncomment that call. You also need to move the print statement you just fixed in the previous Part of this lab.

- Just like in Part 1, you will run each quicksort function 5 different times using each of the 6 different values of N. Record your times in the appropriate table of the data sheet. Average your values and place each average in the column marked μ.
- Plot all three data sets on the graph paper found at the end of this lab. Use N (the sizes of your array) as the x axis and time (your recorded times - in seconds) as the y axis (since time is a function of N). Connect each data set with a sequence of 5 line segments. Finally, label each of the three resulting "curves" using the name of its corresponding data set.
- You can now move on to Part 3 below.

Part 3: Challenge

Knowing that the behaviour of the main loop is O(N), is the shape of the curve on your graph representing the main loop as expected? What does the expected curve look like?

Answer: A straight line since the relationship between N and the execution time of the main loop is linear. If this curve is not linear, does it have a concave upward curve to it? If so, this is probably due to the non-locality of the memory accesses performed during the execution of this loop (i.e., the number of cache misses increases non-linearly as N grows larger). We shall discuss this in more details when we cover memory caches in our last unit (Chapter 6 in our textbook).

Knowing that the average case behaviour of quicksort is O(N log N), have a look at your graph. Can you detect a slight concave upward tendency to the two curves on your graph representing

Quicksort 1 and *Quicksort 2*? This would match the O(N log N) curve (see: http://cooervo.github.io/Algorithms-DataStructures-BigONotation/big-O-notation.html). It may be very slight.

Lastly, looking at the two curves on your graph representing *Quicksort 1* and *Quicksort 2*, which one executes more quickly? It is said that Tony Hoare, the creator of quicksort back in 1960, was trying to write a simple sorting routine that took advantage of locality of memory references. He coded the original version in assembly, which might not be a surprise considering how tight the corresponding C code is! Although Hoare's algorithm is faster, Lomuto's algorithm appears in more CS textbooks because it is the easier of the two to understand.

Thank you to Brad Bart for having inspired this lab.

CMPT 295 – Spring 2023

Last (Family) name:

First name:

Student number:

Circle your section: D100 D200

Participation Activity 8

Data Sheet

Data set: *Main Loop*

N	t1	t2	t3	t4	t5	μ
1000000						
2000000						
4000000						
8000000						
16000000						
32000000						

Data set: *Quicksort 1*

N	t1	t2	t3	t4	t5	μ
1000000						
2000000						
4000000						
8000000						
16000000						
32000000						

Data set: *Quicksort 2*

N	t1	t2	t3	t4	t5	μ
1000000						
2000000						
4000000						
8000000						
16000000						
32000000						