Family Name:_____

Given Names:_____

ID#:_____

CMPT 295 Lab 5 (0.5%)

The purpose of this lab is to examine some of the tools for benchmarking your code. You will use them again on Assignment 6.

Part 1: Benchmarking a Loop

- Login to the CSIL machines in the Linux environment, download and unpack the care package into `sfuhome`, and change directories into `lab5`. There is 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 $[0, \texttt{N}/100]$, 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.

- Try `man getrusage` for a complete description of what data is supplied by the system call. The program uses the fields within `struct timeval ru_utime`. To see what is contained within a `struct timeval`, search for it in a web browser.

- 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, to the nearest millisecond (ms). There should be a fairly close agreement between this number and the number of microseconds that the program reports for the main loop. Why do you think they might differ? Which number is guaranteed to be higher?

- 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 data table 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 and 16000000. Again run the program 5 times, and record your data.

- Does the data make sense? It shouldn't. The running time roughly doubles for each doubling of `N` except for the last run. It's because there is a bug in `main.c`, in how the times are computed. Fix the bug and adjust your data accordingly.

- Calculate the average of each set of 5 numbers and report each number in the column marked μ. Congratulations! You have just benchmarked the `for` loop of `main.c`. 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 Subroutine

Next, you will benchmark two different versions of quicksort.

- Adjust your `main.c` so that it will report the time, in μs, of each of the quicksort functions.

- Just like in Part 1, you will run each 5 different times on each of 5 different values of `N`. Average your values and place each in the column marked μ.

- Plot all three datasets on the graph paper at the end of this lab. Connect each data set with a sequence of 4 line segments and assign a label to each. Use a ruler.

- This is the last step of Lab 5. Bring your work to your TA at the front of the lab. After verifying your work, the TA will then collect this sheet.

Thinking Ahead

There are a few things worth noting about this lab.

First, the behaviour of quicksort — ostensibly $O(N \log N)$ in the average case — can be viewed as a graphical relationship just like any other function. Though it was only over the 5 data points sampled in the lab, you should have observed a slight curve, concave-up. The effect of the $\log N$ term is slight, but certainly present.

Second, the behaviour of the main loop — ostensibly $O(N)$ — had a similar slight curve, concave-up. This is probably due to the non-locality of the memory accesses in the permutation routine: the number of *cache misses* increases non-linearly as N grows larger.

Last, 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 the faster variation of the two, Lomuto's algorithm appears in more CS textbooks because it is easier to understand.

Data Sheet

Main Loop

N	t_1	t_2	t_3	t_4	t_5	μ
1000000						
2000000						
4000000						
8000000						
16000000						

Quicksort 1

N	t_1	t_2	t_3	t_4	t_5	μ
1000000						
2000000						
4000000						
8000000						
16000000						

Quicksort 2

N	t_1	t_2	t_3	t_4	t_5	μ
1000000						
2000000						
4000000						
8000000						
16000000