Project 2: QuickSort & Product Rankings

Submission deadline: 9pm on 13th Oct, 2023
(6% of the total grade + 1% of extra credit)

This assignment consists of two parts – Part A and Part B

Part A (3% of the total grade): Implement the QuickSort algorithm in C and experiment with the performance of different ways of choosing the pivot element. In this assignment you should do so by keeping track of the number of comparisons between input array elements made by QuickSort.[1] For several different input arrays, determine the number of comparisons made with the following implementations of the ChoosePivot subroutine:

1. Always use the first element as the pivot. ("First")

2. Always use the last element as the pivot. ("Last")

3. Use a random element as the pivot. (In this case you should run the algorithm 10 times on a given input array and average the results.) ("Random")

4. Use the *median-of-three* as the pivot element. The goal of this rule is to do a little extra work to get much better performance on input arrays that are nearly sorted or reverse sorted. ("Median-of-three")

   In more detail, this implementation of ChoosePivot considers the first, middle, and final elements of the given array. (For an array with even length 2k, use the kth element for the "middle" one.) It then identifies which of these three elements is the median (i.e., the one whose value is in between the other two), and returns this as the pivot.[2]

   For example, with the input array

8	3	2	5	1	4	7	6

   the subroutine would consider the first (8), middle (5), and last (6) elements. It would return 6, the median of the set {5, 6, 8}, as the pivot element. To choose the median in (a, b, c) in the order of their index, (1) compare a and b first, and find the smaller one (say b), (2) compare b (smaller one in (1)) and c, and if b>c, b is the pivot (2 comparisons), otherwise (b < c), then (3) compare a and c and pick the smaller one as the median (3 comparisons). If you have only two elements in the input array, pick whatever element as the pivot without comparison, then partitioning with the pivot would require only one comparison.

When a pivot is chosen, swap it with the first element before carrying out partitioning with the pivot. After partitioning, the pivot needs to be put into the right position. This is the exact same implementation that we learned in class. Here is an example of running the program – ignore actual values below as they may be incorrect. For "Random", take the floor of the average if it's a floating point number (e.g., 3.7 -> 3).

---

[1] There's no need to count the comparisons one by one. When there is a recursive call on a subarray of length m, you can simply add m-1 to your running total of comparisons. (Recall that the pivot element is compared to each of the other m-1 elements in the subarray in this recursive call.)

[2] A careful analysis would keep track of the comparisons made in identifying the median of the three candidate elements, in addition to the comparisons made in calls to Partition.

```
$ ./quick < inputFile
First: 25
Last: 31
Random: 23
Median-of-three: 21
```

Logistics for Part A:

- Write the code in C – your code should compile with gcc installed on eelab5 or eelab6
- Your program should accept integers from stdin. Each input line has one integer and the number of the integers is at most 10 million. Your program should keep on reading the input until it meets EOF. You can assume the input integer can be represented as a signed 32-bit integer.
- Once your program reads all input values, your program should compute the number of comparisons for different ChoosePivot strategies: "First" (1), "Last" (2), "Random" (3), "Median-of-three" (4). Note that for each case, you need to make a copy of the input values to feed them as input to QuickSort with the different ChoosePivot strategy.
- Your program should print out the number of comparisons as above.
- We will test with only distinct numbers as input for grading.
- It is desirable to do proper error handling, but error handling isn't our focus on grading.
- Name your source code file name as quick.c.

Part B (3% of the total grade + 1% of extra credit):

Imagine you are a data analyst at a niche e-commerce company. Your company wants to enhance its product recommendation system by providing customers with a list of products sorted by both sales volume and user ratings. The goal is to help customers discover products that are not only popular but also highly rated. Each product (denoted by $product_i$, 1 <= i <= N, and i called *productID*) has been sold a specific number of times (T_i, in the range 1-1000) and has accumulated a specific mean rating (R_i in the range 1-10). Write a program that can return the products that are both popular and frequently purchased. Specifically: products with higher average user ratings should appear first (in descending order). Within products with the same rating, products with higher sales volumes should appear first.

The input is given as follows from stdin and ends by EOF.

```
4

75 7

100 6

25 7

20 10
```

The first line (4) indicates the total number of products (4). After that, each line shows T_i and R_i for job_i. For example, "75 7" in the second line says T_1=75, R_1=7 for $product_1$. "100 6" in the third line says says T_2=100, R_2=6 for $product_2$. Write a program that finds and outputs the appropriate sequence of highly rated and popular products.

The input is given as follows from stdin and ends by EOF.

If "inputFile" is a file that contains the sample input above, then it should print out the output as follows. Your program should be named as "productRanking".

```
$ ./productRanking < inputFile
4 1 3 2
```

The output says the highest-ranking productID is four. The tie at the 7 ranked products (productIDs 1 and 3) is ordered by sales volume. Of course, you can manually type in the input to stdin followed by Ctrl+D (EOF).

Logistics for Part B:

- Write the code in C – your code should compile with gcc installed on eelab5 or eelab6
- For generality of this sort of problem, you should assume that N is a positive integer that can be as large as 1,000. T_i can be as large as 1,000 as well, while R_i is in the range 1-10
- Your program should print out the result to stdout.
- Your code should finish in a reasonable amount of time – you'll get penalty if your algorithm is brute-force
- Use qsort() if you need sorting.
- Name your source code file as "productRanking.c".

Collaboration policy:

- You can discuss algorithms and implementation strategies with other students, but you should write the code on your own. That is, you should NOT look at the code of anyone else.
- You can use the code in the textbook or in the slides, but you are NOT allowed to look up the Internet for the quick sort algorithm. You can study the usage of qsort() or command line processing, etc. in the Internet, though.

Submission:

- You need to submit 4 files – your "quick.c" (part A), "productRanking.c" (part B), "Makefile", and "readme" in a tarred gzipped file with the name as YourID.tar.gz. If your student ID is 20211234, then 20211234.tar.gz should be the file name. You need to put these files into a folder whose name is YourID, and then run the tar command (Don't submit a tarbomb!).
- 'make' should build both binaries, "test" and "productRanking".
- You will get **ZERO** points for each part if "quick.c" or "productRanking.c" doesn't compile (due to compiling errors). Make sure that your code compiles well before submission.
- Suppress all warnings at compiling by turning on the "-W -Wall" options in the CFLAGS in Makefile. You will get **some penalty** if gcc produces *any* warnings.
- "readme" (a text file with "readme" as the file name) should contain the followings. (1) Explain how your implementation of the quick sort algorithm in part A works briefly (2) Explain your algorithm of part B. (3) (Optional) prove that your algorithm in part B is always correct (**extra credit of 1%**).

Grading:

- We will evaluate the accuracy for both Part A and Part B.
- We will evaluate the performance of Part B.
- We will evaluate your readme file
- We will evaluate the clarity of your code – coding style, comments, etc.