University of Waterloo
CS240E, Winter 2022
Programming Assignment 2

Due Date: Wednesday, February 9, 2022 at 5pm

Be sure to read the assignment guidelines (`http://www.student.cs.uwaterloo.ca/~cs240e/w22/guidelines/guidelines.pdf`).

Question 0 Academic Integrity Declaration

Read, sign and submit P02-AID.txt now or as soon as possible.

Question 1 [20 marks]

Suppose that you have n bolts and n nuts. Each bolt has exactly one matching nut, but unfortunately they got separated, and so now you don't know which bolt fits with which nut. If you insert the bolt into the nut, then you can determine whether nut is too large, too small, or whether it is the (unique) nut that fits this bolt. But because the size-difference are so small, it is impossible to compare two nuts directly, or two bolts directly.

For this programming assignment, implement a randomized algorithm for which the expected number of comparisons is $O(n \log n)$ and that rearranges a given set of nuts and bolts into matching pairs. Figuring out how to do this rearrangement is part of your assignment. (Hint: the idea should resemble *randomized-quick-sort*, but you cannot do *partition* and therefore must find a different way to re-arrange your arrays into sub-arrays suitably).

The interface is as follows. Nuts and bolts are identified via an ID-number, so your input is only the number n. Given n, you create two arrays N and B of size n (representing n nuts and n bolts) with ID-numbers $0, \ldots, n-1$. You need to rearrange N and B such that in the result nut $N[i]$ matches bolt $B[i]$ for all i, and the arrays are ordered from smallest to largest (in terms of nut-size and bolt-size, not ID number). Your output should write ID-pairs (x, y) (one per line) as well as the total number of comparisons that were used. The `main` in the provided stub-file handles this input and output; do not change this part of the code but add the code that does the actual rearrangement. You are welcome (in fact, invited) to break up your code using subroutines or even entire new classes.

To compare a nut with a bolt, use `compareNutAndBolt(int nutID, int boltID)` (within class `NutBoltComparer`), which returns a number that is equal to zero if the nut fits the bolt, negative if the nut is too small, and positive if the nut is too big. You *must* use this function for doing comparisons, and you *must not* change the class. We recommend that you never even look at `NutBoltComparer.cpp`; your program must work no matter how `nutBoltCompare` is implemented. The only promise that we make is that the outcome of comparisons will be consistent with some (unknown) total order among the nuts and among the bolts.

Submit: A file `nutsAndBolts.cpp` with the details of the `main` routine filled in. The provided file gives a stub. Submit your solution to Marmoset. Marmoset will be set up to translate your program with `g++ -std=c++17`.

You must ensure that your program has $O(n \log n)$ expected run-time; in particular the number of comparisons may be bad sometimes, but repeated runs (even on the same instance) must use $O(n \log n)$ comparisons in average.