Code Jam to I/O for Women 2016 - Code Jam to I/O for Women

Analysis: Password Security

To solve this problem, let's start with some general observations about the passwords:

- If any password is only one letter, then the answer is `IMPOSSIBLE` because that password is guaranteed to appear in any permutation of the alphabet. (Incidentally, Google Code Jam does not endorse the use of single-character passwords! An attacker with access to a list of alphabet letters can quickly try them all.)
- If any password has a repeated letter, it can be ignored, since it cannot possibly appear in any permutation of the alphabet.

In Small dataset 1, there's only one password. You might have considered picking a single permutation and hoping that none of the passwords in the test cases happened to be in it. We designed our test cases to thwart that approach: for some letter (say, A), they include a set of fifty test cases BA, CA, ..., ZA, AB, AC, ..., AZ. No matter where the A is in your chosen string, it will be before or after at least one other letter, and is guaranteed to include one of those two-letter passwords, so it will fail at least one case.

One possible approach for Small dataset 1 is to build an alphabet permutation that will definitely not contain each given password:

- If the password is only one letter, return `IMPOSSIBLE`.
- If the password has a repeated letter, it does not affect the answer, and you may return any permutation.
- Otherwise, reverse the password and then tack on whatever remains of the alphabet, and return that.

Another approach is to produce a set of alphabet permutations of which at least one will not contain the password, and then check all of those until you find one that works. For example, you can generate every possible rotation of the alphabet: ABC...XYZ, BC...XYZA, C...XYZAB, and so on. No string of two or more letters can possibly appear in all 26 of those rotations, so one of them must work. Or, you can check the alphabet and the reversed alphabet — again, no string of two or more letters can appear in both of those.

In Small dataset 2, since there are multiple passwords to contend with, the strategies above won't work. Moreover, there are now sets of passwords for which there is no solution, even if none of the passwords is a single letter. One such set is the tricky set of fifty words mentioned above: BA, CA, ..., ZA, AB, AC, ... , AZ. Just as there was no one password that could pass when those fifty words were used as individual test cases, there is no solution when all of them appear in a single test case. Even when there is a solution, you have to find one of the four hundred septillion permutations of the alphabet that works.

One strategy is to try enumerate all of the impossible scenarios. With some thought, it is possible to prove that they always fall into one of three categories:

1. The same letter is required to be at both the beginning and the end. Our AB, AC, ..., AZ, BA, CA, ..., ZA example from before falls into this category. The first 25 bigrams would force A to be at the end (by prohibiting any other letter from appearing after it), and the last 25 bigrams would force A to be at the beginning (by prohibiting any other letter from appearing before it).
2. Two different letters are both required to be at the beginning. For example, consider the set BA, CA, ..., ZA, AB, CB, ..., ZB. The first 25 bigrams would force A to be at the

beginning, and the last 25 bigrams would force B to be at the beginning. But they can't both be there!
3. Two different letters are both required to be at the end. For example: AB, AC, ..., AZ, BA, BC, ..., BZ.

We have a truly marvelous demonstration of this proposition which the margins of this webpage are too narrow to contain[1].

Even if you can't prove to yourself that these are the only three impossible cases, you can always run a simulation on a smaller scenario (with, say, all sets of 10 bigrams from a six-letter alphabet) to increase your confidence.

So, you can explicitly check for these situations, and if the test case doesn't fall into one of those categories, it's guaranteed that at least one answer exists. Now the problem is to find one! You can use various heuristics to try to increase the probability of finding an answer, but you have a perfectly good computer. Why not try a bunch of random permutations until one works?

It is worth noticing that, if the random procedure does find an answer, it will certainly be correct. So, even if you are unconvinced about your impossible case enumeration, you can always try it. If the program finishes, then the output is guaranteed to be correct. If it does not, it could be that the random procedure is too slow in finding a solution, or that you've missed some impossible case.

Most Code Jam problems are written such that random approaches take a prohibitively long time, if they work at all. However, sometimes you can prove to yourself that a random approach is suitable. Consider a case that isn't impossible, but is very close to one of the impossible ones. For example, this case forces ZA (but not BZA) at the end.

AB AC AD AE AF AG AH AI AJ AK AL AM AN AO AP AQ AR AS AT AU AV AW AX AY AZ BA CA DA EA FA GA HA IA JA KA LA MA NA OA PA QA RA SA TA UA VA WA XA YA BZA

As you can confirm with a simulation, the probability of a random permutation meeting this particular set of conditions turns out to be more than 0.147%. Those might not be great odds for a personal wager, but they're great odds for a computer that can generate and check tens of thousands of alphabet permutations (or more) each second. The odds of finding an answer in 1000 tries are $1 - ((1 - 0.00148)^{1000})$, or a little over 77%. If we swap out the 1000 for 10000 to represent 10000 tries, the odds of finding an answer go up to over 99.9999%. But we also need to consider that there are 100 test cases; if we pretend that none of them are IMPOSSIBLE, that drops the overall chance of succeeding to a mere 99.996%.

So, if you can convince yourself that there can't be cases much worse than the one above, you can just run 1000 times — or, if you want even greater certainty and your program is fast enough, 10000 times — until you find an answer.

Once you're convinced that your random procedure will either find a correct answer (with very high probability) or fail, you can even use a set of 10000 runs with no found answer as "proof" (beyond a reasonable doubt, at least) of impossibility — you don't need to explicitly check for the types of impossible cases mentioned above at all.

Randomized approaches like this are useful outside of programming contest problems. Check out the Fermat primality test, which provides one way of checking whether a number is likely to be prime without actually factoring it. Primality tests like that one are commonly used as part of encryption systems that rely on a fresh supply of large prime numbers.

[1] We do have a full proof, but we leave it as an exercise for mathematically inclined readers to do themselves.