SACO 2006 Day 1 solutions

Bruce Merry

Four Yorkshiremen

Suppose a number $n > 1$ is not prime. Then it has a factor a, and so n can be written as $n = ab$. If $a > \sqrt{n}$ and $b > \sqrt{n}$ then $ab > n$, a contradiction. Thus any non-prime has a factor which is at most its square root. Checking whether a number is prime is thus as easy as checking all possible factors up to the square root.

Checking if a number is a palindrome is most easily done by converting the number to a string (using the features of your language of choice) and checking if it is symmetrical. To count the prime palindromes less than N, it suffices to test each number in the range.

Joke

The first simplification that can be made is to solve a slightly easier problem. Let $f(K, c)$ be the number of letter c's in the first K characters of the ciphertext. Then the frequencies that should be output are $f(B, c) - f(A - 1, c)$.

Consider the case when $N = 1$ (one encryption). Applying one level of encryption to "badbeef" gives "aa|dc|a|aa|ea|ea|f", where the bars separate the groups generated from each letter. If we want to count the frequencies in the first 6 characters, we will have the first three groups, plus part of the fourth group. Since the strings in the first three groups are known in advance (they are from the substitution table), we can compute frequency tables for them in advance. The frequency counting then just involves adding together these tables, and adding on counts for any partial group.

To solve the problem completely, it is sufficient to apply this principle recursively. In advance, we compute the length and frequency table that results from encrypting each letter each possible number of times (up to N). To count the frequencies in a range, we start with the biggest blocks, namely those that come from encrypting the letters of the plain-text N times. For one of these we may have a partial block, in which case we need, say, K_1 letters from that block. If that block was generated from a letter 'a', we then apply one level of encryption to expand the 'a' to a once-encrypted string, and apply the algorithm recursively to compute counts for the first K_1 characters of the $N - 1$-times encryption of this string.

Mouse

Finding a way to link a bunch of things into a single ring using only prescribed connections is known as the Hamiltonian cycle problem, and in the general case there are no known efficient solutions (technically, the problem is *NP-complete*). However, the constraint that every mouse is a stranger to more than half the other mice makes the problem easier to solve.

One can start by putting one mouse into a line, then finding a stranger to it, then a stranger to that one and so on, just building up the line one mouse at a time. However, at some point one might find that the mouse just added knows *all* the mice that are not yet in the line. In this case, it is necessary to go back and modify the line. One way to do this is to take a tail-section of the line and reverse it (e.g., replace 1 2 3 4 5 6 with 1 2 3 6 5 4).

Let the mouse just added to the line be Mr. A, one of the mice not yet in the line be Mr. B, and the number of mice currently in the line be k. There are k possible reversals (including just reversing Mr. A, which has no effect). Mr. A knows all $N-k$ mice not in the line, so he knows at most $k-1-\frac{N}{2}$ of the mice in the line. This rules out $k-1-\frac{N}{2}$ of the possible reversals, since he cannot end up next to a mouse he knows. We also want to be able to append Mr. B to the line, which rules out up to $\frac{N}{2}-1$ of the reversals, leaving at least 2. We can thus always add other mouse to the line.

Once all the mice are in a line, we may not be able to connect it into a circle. If this is the case, we do another reversal. We want the first mouse to stay in place, so we rule out reversing the entire range and only consider $N-1$ reversals. Of these, up to $\frac{N}{2}-1$ may be illegal because they would put the last mouse next to a friend, and another $\frac{N}{2}-1$ because they would move a friend of the first mouse to the end. This leaves one viable reversal.

Wooden

You are given some information about the shape of the roof, and any efficient solution must exploit this information. There are two key optimisations that can be made with the information:

1. To determine whether an animal fits in a particular place, it is only necessary to check whether the left or right end is too tall. If both sides fit, then the roof cannot dip in the centre.

2. Shorter animals should be packed towards the ends, with taller animals towards the middle.

A basic algorithm (that can score up to 60%) starts by sorting the animals by height. It then goes through the animals from shortest to tallest, and for each animal decides to either discard it, to pack it as far left as possible, or to pack it as far right as possible. Since there is a choice for each animal, this is most easily implemented by recursion, with three-way branching at each level. The number of operations grows exponentially, so this algorithm is inappropriate for the larger test cases.

Suppose the recursive function above took three parameters: L, the furthest left that the next animal can go (because further left would collide with another animal), R, the furthest right that the next animal can go, and A, the number of animals that have already been considered. This is mathematically a function, in the sense that for a given (L, R, A) it will always return the same value. We can thus store these values in an array, indexed by L, R and A. A simple way to modify the recursion is to treat this array as a complete cache: if we have seen the same parameters before, just return the value from the array; if not, compute the value as normal and save it in the cache before returning it. This is known as *memoisation*. A variation of this technique called *dynamic programming* is similar but works iteratively (looping over L, R and A to fill in the values in the array using previously computed values), and can be optimised to reduce the memory overhead.