Santa's Programming Challenge 2016 – editorial

A Anagrams

One solution is to count how many times each letter appears in both strings, and answer `YES` if the counts are the same. Another is to sort both strings and check if they are equal.

B Slim Chances

Denote the event of getting k coin flips out of n by K, and the event of the first coin flip being heads by F. We want to find out whether $\mathbb{P}[K|F]$ is greater than $\mathbb{P}[K]$.

First let us notice a special case: if $k > n$, then $\mathbb{P}[K] = 0 = \mathbb{P}[K|F]$, and we answer `SAME`. Otherwise, there are two methods again:

- One is to write down a formula for $\mathbb{P}[K]$, which is $\binom{n}{k}(1/3)^k(2/3)^{n-k}$, and for $\mathbb{P}[K|F]$, which is the probability of getting $k-1$ heads out of $n-1$, equal to $\binom{n-1}{k-1}(1/3)^{k-1}(2/3)^{n-k}$. Trying to compute and compare these values outright can lead only to a headache (it would be slow and/or not precise enough), but we can simplify them a lot:

$$1 \leqslant \frac{\mathbb{P}[K|F]}{\mathbb{P}[K]} = \frac{\binom{n-1}{k-1}(1/3)^{k-1}(2/3)^{n-k}}{\binom{n}{k}(1/3)^k(2/3)^{n-k}} = \frac{\frac{(n-1)!}{(k-1)!(n-k)!}}{\frac{n!}{k!(n-k)!} \cdot \frac{1}{3}} = \frac{\frac{(n-1)!}{(k-1)!(n-k)!} \cdot 3}{\frac{(n-1)!n}{(k-1)!k(n-k)!}} = \frac{3k}{n}.$$

- The other method is to rewrite the conditional probability:

$$1 \leqslant \frac{\mathbb{P}[K|F]}{\mathbb{P}[K]} = \frac{\mathbb{P}[K \cap F]}{\mathbb{P}[K]\mathbb{P}[F]} = \frac{\mathbb{P}[F|K]}{\mathbb{P}[F]} = \frac{\frac{k}{n}}{\frac{1}{3}} = \frac{3k}{n}$$

  (this is an implicit application of Bayes' theorem).

Both methods lead to the same solution: it is enough to compare n to $3k$.

C Operators

Let OPT_i be the optimal solution for the first i elements of a (i.e., for the sequence $a_1, \ldots, a_i$). We start with the following simple observation. Consider an optimal placement of operators $+$ and $\times$ between $a_1, \ldots, a_n$, achieving OPT_n. We can either have $+$ or $\times$ in front of a_n. In case we have $+$ in front of a_n, then $OPT_n = OPT_{n-1} + a_n$. In the other case, when $\times$ is in front of a_n, then as $\times$ can appear only between two consecutive elements of a, the sign in front of a_{n-1} must be $+$, and we have $OPT_n = OPT_{n-2} + a_{n-1} \cdot a_n$.

We do not know whether, in order to obtain OPT_n, we should put $+$ or $\times$ in front of a_n. However, our observations give rise to a dynamic programming approach, and we can obtain OPT_n using the following recurrence:

$$OPT_i = \begin{cases} 0 & \text{if } i < 1 \\ a_i & \text{if } i = 1 \\ \max\{OPT_{i-1} + a_i, OPT_{i-2} + a_{i-1} \cdot a_i\} & \text{if } 1 < i \leq n \end{cases}$$

Using either a bottom-up or a top-down approach, OPT_n can be computed in $O(n)$ time.

D In The Dark

We build a graph where cells of the grid are the nodes, and each node has four edges to the adjacent cells (except the border nodes, which have fewer). Then the task becomes a standard application of BFS, which we run from `A`. Whenever visiting a node for the first time, we should store in it a pointer to its parent in the BFS-tree; then we can efficiently recreate the shortest path by walking back from `B`.

E Grapes

We are given a rooted tree. For every edge of the tree, when this edge is removed, the tree falls apart into two pieces; we will find the number of [non-root] leaves in each of the two pieces. Once we have that, for every edge we take the minimum of the two numbers, and find the maximum of these minima over all edges.

Suppose the entire tree has ℓ_1 non-root leaves. Every edge in the tree goes from a node v to its parent; the numbers of leaves in the two pieces are the number of leaves in v's subtree (call it ℓ_v) and $\ell_1 - \ell_v$, respectively.

We will compute all values ℓ_v recursively. If v is a leaf, then $\ell_v = 1$; otherwise, $\ell_v = \sum_{w\text{: son of } v} \ell_w$. We can just use this formula in a post-order traversal of the tree, which is implemented as a DFS. The complexity of this solution is $O(n)$.

F Beautiful Sleigh

We begin by computing, for every pair of letters α and β, how much it costs to change α to β (this value can be ∞). This is just the shortest path between α and β in the graph formed from the input by treating Santa's painting methods as weighted edges. Note that even though there can be 10^5 methods, only at most $26 \cdot 25$ of them are relevant (we take the cheapest edge between any pair of vertices). Since the graph is small, any method of obtaining all-pairs shortest paths on a weighted graph will work here. The easiest such method by far is the Floyd-Warshall algorithm.

Once we have that, we can just check every divisor d of n and try to make the string have a period of length d. That is, we want to ensure that: $s_1 = s_{d+1} = s_{2d+1} = ... = s_{n-d+1}$, $s_2 = s_{d+2} = s_{2d+2} = ... = s_{n-d+2}$, ..., $s_d = s_{2d} = ... = s_n$. So we have d independent subproblems, each of which asks us to make n/d letters equal, at the cheapest cost possible. To solve one such subproblem, we just loop over all possible letters β and see how costly it is to change all the n/d letters to β; we pick the cheapest β. We sum up the d costs to get a cost for the period length d; having done this for every period, we select the cheapest one.

The complexity of the solution is $O(n \cdot D(n) \cdot |\Sigma|)$, where $D(n)$ is the number of divisors of n (if $n \leq 10^5$, then $D(n) \leq 128$) and $|\Sigma| = 26$. This was efficient enough to get accepted. However, there are two ways to make this solution faster:

- A nice observation is that it is enough to check only prime divisors p of n (and take $d = n/p$); this is because if $s = t^{kl}$, for some integers $k, l > 1$, then also $s = (t^k)^l$ (enforcing a period of length $n/(kl)$ is no easier than enforcing a period of length n/l). While a number under 10^5 may have up to 128 divisors, it will only have at most 6 prime divisors.
- To solve a single subproblem, instead of iterating over all n/d letters for each possible β (which takes time $O(|\Sigma| \cdot n/d)$), one can instead first compute the counts of each letter and then iterate over β's, resulting in a running time of $O(n/d + |\Sigma|^2)$.

Finally, it is important to use 64-bit integers (`long long` in C++, `long` in Java) for the results. Not doing so would result in overflows.

G Elvish Game

The solution hinges on the following crucial observation, which says that replacing two equal letters by two other equal letters does not change anything:

Lemma 1. *Let s be a string (state of game). If s' is a string obtained by replacing two occurences of a letter α in s by two occurences of another letter β, then s and s' are equivalent (in the sense that the same elf wins the game).*

Proof. Consider any sequence of moves played on s; we will argue that almost the same sequence can be played on s'. Because there is more than one α in s, at some point one of the elves must make a move $\alpha\alpha \to \gamma$, for some γ. The same game can be played on s', with the exception that instead of this move, a move $\beta\beta \to \gamma$ is made. At that point the states of both games become equal. (And vice versa, every sequence which can played on s' can also be played on s.)

Therefore the state of the game can be adequately described with two numbers: p, the number of pairs of equal letters, and s, the number of single letters left over. (For example, *aaabbbbccc* is described by $(p, s) = (4, 2)$ because $aaabbbbccc = aa + bb + bb + cc + a + c$.) We

have $p \leq 50000$ and $s \leq 26$. At any point, there are two moves possible. We take a pair and insert a new letter, which can be one of two types: a letter which creates a pair with a single letter (only possible if $s > 0$), or a letter which becomes single (only possible if $s < 26$). Thus the game (p, s) is in a winning position if and only if one of the games $(p, s-1)$, $(p-1, s+1)$ is losing. This leads to a simple dynamic programming solution with complexity $O(|\Sigma| \cdot n)$ (where $|\Sigma| = 26$).

H Infinite Incantation

Let us first interpret the problem in the language of graph theory. We are given an undirected graph $G = (V, E)$ with $n \cdot m$ vertices corresponding to the board given as input and edges between adjacent fields. Each vertex has a letter written on it. Now we would like to find a cycle in G (not necessarily a simple cycle) such that when we traverse the cycle an infinite number of times, then the resulting sequence of letters is equal to $SSSSSS\ldots$, where S is the word (spell) given as input. For instance if $S = abcabcabc$, then it is fine to output a cycle which produces $abcabc$, since the infinite repetitions of these two words are equal.

The insight required to solve this problem is that the graph G is not the one we want to work with. It is better to instead construct another graph G', in which the problem just reduces to finding cycles. Let us now define $G' = (V', E')$:

- The vertices of V' are triples (x, y, i) corresponding to the position on the board (x, y) ($1 \leq x \leq n$ and $1 \leq y \leq m$) and the position in the spell i ($1 \leq i \leq s$). We keep only those triples (x, y, i) which satisfy the condition that the letter at position i in the spell S is the same as the one at position (x, y) in the board.

- Two vertices (x_1, y_1, i_1) and (x_2, y_2, i_2) are adjacent in G' if and only if (x_1, y_1) and (x_2, y_2) are adjacent and i_1, i_2 differ by one (modulo s).

It is easy to see that now the problem reduces to finding a cycle which starts in a vertex of the form $(x, y, 1)$ for some (x, y). This can be done (e.g. using DFS) in linear time which respect to the size of G', which is $O(nms)$.