Solutions

BEADS

We represent each swapper *i* over conveyor p_i and p_i +1 as a permutation π_i, such that π_i (p_i) = p_i +1, π_i (p_i +1) = p_i, and π_i (j) = j for $j \notin \{ p_i, p_i +1\}$. Thus, for this task, the answer to question (*K*; *J*) is

$\pi_J (\pi_{J-1} (\ldots(\pi_1 (K))))$

A naive solution that evaluates that permutation for each question runs in time *O*(*MQ*), where *M* is the number of swappers and *Q* is the number of questions asked.

We can speed that computation up by some preprocessing. For simplicity, we assume that *M* is some power of 2. Let permutation $\pi_{i,j}$ be the composition of π_j, π_{j-1}, …, π_i, i.e.,

$\pi_{i,j} = \pi_j \bullet \pi_{j-1} \bullet \ldots \bullet \pi_i$

For each level *l* = 0, 1 … log*M*, we precompute partial composition $\pi_{i2^{l},(i+1)2^{l}}$ for *i* = 0, 1,…$M/2^l$. There are M + M/2 + M/4 + … ≤ 2M such partial compositions. Note that with these partial compositions at hand, one can answer the query by evaluating *O*(log*M*) partial compositions. For example, to compute $\pi_{1,13}$, we need only to compute $\pi_{1,8} \bullet \pi_{9,12} \bullet \pi_{13,13}$.

Another key issue is how to represent each precomputed $\pi_{i,j}$. Note that if we store each partial composition directly as a complete function, we need Ω(N) space for one of them and this leads to Ω (NM) memory requirement, which is too much. We can reduce the memory requirement by only store the value $\pi_{i,j}$ (a) for only a such that $\pi_{i,j}$ (a) 6= a. We can store them as a sorted list, and it takes *O*(log*N*) time to evaluate one partial composition. As such, Q questions can be answered in time *O*(*Q*log*M* log*N*). Since there are *M* swappers and at most log*M* levels, the memory requirement reduces to *O*(*M* log*M*). Any solution that meets this time and space bounds receives full score.

DNA

First, we ignore the templates and only count the number of sequences of form-*k*.

Let $A[i][k][\alpha]$ denote the number of sequences of form-*k* of length *i* beginning with α. Note that if we require that the second character β comes before α, possible sequences of form-*k* are those constructed by concatenating α with form-(*k*-1) sequences beginning with β. On the other hand, we require that β is α or comes after α, we can concatenating α with any form-*k* sequence.

Thus, we have the following recurrence

$$A[i][k][\alpha] = \sum_{\beta \geq \alpha} A[i-1][k][\beta] + \sum_{\beta < \alpha} A[i-1][k-1][\beta]$$

This idea can be extended to count the number of sequences that also match the templates. Table *A* can be computed in time *O*(*MK*).

Given *A*, we can trace back the computation of *A* to find the required *R*-th gene sequence.

Roads

We first note one structural property that allows us to use greedy approach to solve this task. Given a graph *G* = (*V*, *E*), where *V* is the set of villages, and *E* is the set of roads. We color each edge *e blue* if it is a cobblestone road and *red* otherwise. This task wants to find a spanning tree of *G* that contains exactly *k* blue edges.

Consider any spanning tree *T*. Note that if we consider subgraph of *T* that contains only blue edges, we get a forest. It is known that for any two forest *F1* and F2, such that *F1* has *k1* edges, *F2* has *k2* edges, and *k2* > *k1*, there exists some edge $e \in F2$ such that $F1 \cup \{e\}$ remains a forest. (See, e.g., Cormen et al., Chapter 16 on matroids.) This implies that to find spanning tree with *k* blue edges, we can never make a mistake by taking in wrong blue edges.

With that fact, there are many ways to find a required spanning tree. We present here one solution.

The algorithm has two rounds. First, we try to find a set of “required” blue edges, i.e., these edges must be in the tree to ensure connectivity. We start by finding all connected components of a subgraph containing only red edges. We then greedily add blue edges joining different components so that the graph is connected. These blue edges added in this round form the starting blue edge set B. Clearly if we need more than *k* edges to connect the graph, we can be sure that there is no spanning tree with the required property.

On the second round, we start with a forest *F* with only edges in *B*. We then greedily add blue edges to *F* while maintaining that the resulting graph *F* remains a forest. We keep adding until we get *k* blue edges in *F*. After that, we complete the algorithm by using red edges to join different connected components in *F*.

If we fail in any steps, it means that no solution exists.

The main data structure that we use in both rounds is the union-find data structure for maintaining disjoint sets.

In the first round, to find connected components of red edges, we just union every pair of components containing end points of red edges. Before adding blue edges *e*, we call two find subroutines to check if *e* joins two different components. If that's the case, we union both components. In the second round we may proceed like the second step of the first round.

To analysis the running time, for each round, we call at most n unions and *O(m)* finds. Using a set of tree with union-by-rank heuristics ensures that each operation (union or find) runs in time *O(log n)*. Thus, the algorithm runs in time *O((n + m) log n)*.