Diversity – solution

Adrian Beker

September 2, 2021

1 The solution in the case of a single query

To begin with, let's consider the case $Q = 1$, i.e. the case of a single query. We will first determine how to efficiently compute the total diversity of a given sequence, which will be denoted by S. If we denote by K the diversity of the sequence, we may without loss of generality assume that the sequence consists of the numbers $1, 2, \dots, K$ (by performing coordinate compression). A *gap* of a value $i \in \{1, 2, \dots, K\}$ is a maximal (with respect to inclusion) contiguous subsequence not containing i. Let i have r_i gaps and let $\ell_{i,1}, \dots, \ell_{i,r_i}$ be their lengths, in the order from left to right. Then it is easy to see that i occurs on a total of

$$\frac{N(N+1)}{2} - \sum_{j=1}^{r_i} \frac{\ell_{i,j}(\ell_{i,j}+1)}{2}$$

contiguous subsequences. Indeed, each contiguous subsequence not containing i is contained in a unique gap. Hence, we have (using linearity of expectation)

$$S = \sum_{i=1}^{K} \left(\frac{N(N+1)}{2} - \sum_{j=1}^{r_i} \frac{\ell_{i,j}(\ell_{i,j}+1)}{2} \right).$$

Now note that $\sum_{j=1}^{r_i} \ell_{i,j} = N - c_i$, where c_i is the number of occurrences of the value i in the sequence, so we have

$$\sum_{i=1}^{K} \sum_{j=1}^{r_i} \ell_{i,j} = \sum_{i=1}^{K} (N - c_i) = KN - \sum_{i=1}^{K} c_i = KN - N = N(K-1).$$

Hence, the above expression for S can be simplified as follows:

$$S = \frac{1}{2} \left(KN(N+1) - N(K-1) - \sum_{i=1}^{K} \sum_{j=1}^{r_i} \ell_{i,j}^2 \right).$$

We conclude that the problem reduces to the following: permute a given sequence so that the sum of the squares of the lengths of all gaps, call it T, is as large as possible. In order to solve this problem, we make the following observation:

Claim 1. In every optimal solution, the occurrences of each value $i \in \{1, 2, \dots, K\}$ form a contiguous subsequence.

Proof. We say that a value $i \in \{1, 2, \dots, K\}$ is *good* if its occurrences form a contiguous subsequence. We will show that if there exists $i \in \{1, 2, \dots, K\}$ which is not good, then we can permute the sequence in such a way that T and the number of good values both increase. To this end, pick some value i which is not good and consider two adjacent blocks formed by its occurrences in the sequence. For each value $j \neq i$, let $u_j, v_j \in \{1, 2, \dots, r_j\}$ be the indices of the gaps of the value j

containing the left and the right block respectively. Let the lengths of the left and the right block be a and b respectively. On moving the left block immediately next to the right block, T changes by

$$\Delta_1 = \sum_{j\neq i,\ a_j\neq b_j} [(\ell_{j,u_j} - a)^2 + (\ell_{j,v_j} + a)^2 - \ell_{j,u_j}^2 - \ell_{j,v_j}^2] > 2a \sum_{j\neq i,\ a_j\neq b_j} (\ell_{j,v_j} - \ell_{j,u_j}).$$

Similarly, on moving the right block immediately next to the left block, T changes by

$$\Delta_2 = \sum_{j\neq i,\ a_j\neq b_j} [(\ell_{j,u_j} + b)^2 + (\ell_{j,v_j} - b)^2 - \ell_{j,u_j}^2 - \ell_{j,v_j}^2] > 2b \sum_{j\neq i,\ a_j\neq b_j} (\ell_{j,u_j} - \ell_{j,v_j}).$$

Note that the above strict inequalities hold because there exists at least one value $j \neq i$ such that $a_j \neq b_j$, e.g. a value that occurs between the two blocks in consideration. Hence, at least one of the differences Δ_1, Δ_2 is strictly positive, so we can certainly perform one of the described transformations so that T increases. Moreover, it is clear that the number of good values increases because the value i becomes good, but no other value ceases to be good. This concludes the proof. □

By Claim 1, the problem reduces to the following: find a permutation π of the set $\{1, 2, \dots, K\}$ which maximizes the function

$$f(\pi) = \sum_{i=1}^{K} \left(\left(\sum_{j=1}^{i-1} c_{\pi(j)} \right)^2 + \left(\sum_{j=i+1}^{K} c_{\pi(j)} \right)^2 \right).$$

In other words, we have to permute the sequence $c_1, \dots, c_K$ so that the sum of the squares of the sums of all proper prefixes and suffixes is as large as possible. We can solve this problem naively in time $\mathcal{O}(K \cdot K!)$, or using dynamic programming with bitmasks in time $\mathcal{O}(K \cdot 2^K)$. However, for a solution with polynomial complexity, it is necessary to make the following observation:

Claim 2. In every optimal solution, the values in the sequence first increase and then decrease. More precisely, if π is an optimal permutation, then there exists $i \in \{1, 2, \dots, K\}$ such that $c_{\pi(j)} \leq c_{\pi(j+1)}$ for all $1 \leq j < i$ and $c_{\pi(j)} \geq c_{\pi(j+1)}$ for all $i \leq j < K$.

Proof. Fix and index $j \in \{1, 2, \dots, K-1\}$ and let π' be the permutation obtained by swapping the elements at positions j, $j+1$ in π, i.e. formally $\pi' = \pi \circ \tau$, where τ is the transposition which swaps j, $j+1$. Then we have

$$\begin{aligned} f(\pi') - f(\pi) &= (L_j + c_{\pi(j+1)})^2 + (R_{j+1} + c_{\pi(j)})^2 - (L_j + c_{\pi(j)})^2 - (R_{j+1} + c_{\pi(j+1)})^2 \\ &= 2(L_j - R_{j+1})(c_{\pi(j+1)} - c_{\pi(j)}), \end{aligned}$$

where we define $L_k = \sum_{l=1}^{k-1} c_{\pi(l)}$, $R_k = \sum_{l=k+1}^{K} c_{\pi(l)}$ for $1 \leq k \leq K$. Thus, if π is optimal, it follows that $L_j - R_{j+1}$ and $c_{\pi(j+1)} - c_{\pi(j)}$ have opposite sign, from which the desired conclusion follows. □

It follows from Claim 2 that an optimal solution can be built as follows: we iterate over the values $c_1, \dots, c_K$ in increasing order and we maintain two sequences, a left one and a right one. In each step, we put the current value either at the end of the left sequence or at the beginning of the right sequence. In the end, we obtain the sought permutation by concatenating the left and the right sequence. It is now not hard to devise a solution via dynamic programming which runs in time $\mathcal{O}(K \cdot N)$, where $N = \sum_{i=1}^{K} c_i$. The state is the current position in the sorted sequence $c_1, \dots, c_K$ and the sum of the left sequence, while the transition consists of choosing where to put the current value.

It turns out that an even stronger assertion holds (strictly speaking, it is stronger than the assertion obtained from Claim 2 on replacing the universal quantifier by an existential one):

Claim 3. If we (without loss of generality) assume that $c_1 \leq \ldots \leq c_K$, then the permutation π_0 given by

$$\pi_0(i) = \begin{cases} 2i - 1 & \text{if } 1 \leq i \leq \lceil \frac{K}{2} \rceil \\ 2(K - i + 1) & \text{if } \lceil \frac{K}{2} \rceil < i \leq K \end{cases}$$

is optimal. In other words, it is optimal to first arrange all elements with odd indices in increasing order and then all elements with even indices in decreasing order.

Proof. Let π be any permutation; we will show that $f(\pi_0) \geq f(\pi)$. To begin with, we introduce the map

$$s : S_K \to \mathbb{N}^{2(K-1)}$$

with the property that for all $\sigma \in S_K$ we have $s(\sigma)_1 \geq \ldots \geq s(\sigma)_{2(K-1)}$ and the multiset $\{s(\sigma)_i \mid 1 \leq i \leq 2(K-1)\}$ is equal to the multiset consisting of the sums of all proper prefixes and suffixes of the sequence $c_{\sigma(1)}, \ldots, c_{\sigma(K)}$ (here S_K denotes the set of all permutations of the set $\{1, 2, \ldots, K\}$). We have to show that

$$\sum_{i=1}^{2(K-1)} \varphi(s(\pi_0)_i) \geq \sum_{i=1}^{2(K-1)} \varphi(s(\pi)_i),$$

where

$$\varphi : \mathbb{R} \to \mathbb{R}, \quad x \mapsto x^2$$

is a convex function. To this end we will use Karamata's inequality. It suffices to show that for all $1 \leq i \leq 2(K-1)$ we have

$$\sum_{j=1}^{i} s(\pi_0)_j \geq \sum_{j=1}^{i} s(\pi)_j \tag{$\dagger$}$$

and that equality holds for $i = 2(K-1)$. The latter claim is obvious because for all $\sigma \in S_K$ we have

$$\sum_{j=1}^{2(K-1)} s(\sigma)_j = \sum_{i=1}^{K-1} \left(\sum_{j=1}^{i} c_{\sigma(j)} + \sum_{j=i+1}^{K} c_{\sigma(j)} \right) = N(K-1).$$

To prove ($\dagger$), observe that it is enough to show that for all $K \leq i \leq 2(K-1)$ we have

$$\sum_{j=i}^{2(K-1)} s(\pi_0)_j \leq \sum_{j=i}^{2(K-1)} s(\pi)_j. \tag{$\star$}$$

Indeed, then the inequality ($\dagger$) readily follows in the case $K-1 \leq i < 2(K-1)$ by subtracting the inequality ($\star$) with $i+1$ in place of i from the inequality ($\dagger$) with $2(K-1)$ in place of i. On the other hand, in the case $1 \leq i \leq K-1$, the inequality ($\dagger$) follows from the inequality ($\star$) with $2K-i-1$ in place of i because the i prefixes/suffixes with smallest sums are complementary to the i prefixes/suffixes with the largest sums.

Hence, it remains to prove the inequality ($\star$). It is enough to show that for all $1 \leq i \leq K-1$ and $0 \leq j \leq i$, the following inequality holds

$$\sum_{k=1}^{\lceil \frac{i}{2} \rceil} \sum_{l=1}^{k} c_{\pi_0(l)} + \sum_{k=1}^{\lfloor \frac{i}{2} \rfloor} \sum_{l=1}^{k} c_{\pi_0(K-l+1)} \leq \sum_{k=1}^{j} \sum_{l=1}^{k} c_{\pi(l)} + \sum_{k=1}^{i-j} \sum_{l=1}^{k} c_{\pi(K-l+1)}.$$

After grouping the same terms together, we see that this inequality is equivalent to

$$\textstyle\sum_{k=1}^{\lceil \frac{i}{2} \rceil} \left(\lceil \frac{i}{2} \rceil - k + 1 \right) c_{\pi_0(k)} + \sum_{k=1}^{\lfloor \frac{i}{2} \rfloor} \left(\lfloor \frac{i}{2} \rfloor - k + 1 \right) c_{\pi_0(K-k+1)} \leq \sum_{k=1}^{j} (j-k+1) c_{\pi(k)} + \sum_{k=1}^{i-j} (i-j-k+1) c_{\pi(K-k+1)}.$$

This inequality we can prove as follows. First, we replace the coefficients $j-k+1$ for $1 \leq k \leq j$ and $i-j-k+1$ for $1 \leq k \leq i-j$ in the expression on the right-hand side with the coefficients $\lceil \frac{i}{2} \rceil - k + 1$ for $1 \leq k \leq \lceil \frac{i}{2} \rceil$ and $\lfloor \frac{i}{2} \rfloor - k + 1$ for $1 \leq k \leq \lfloor \frac{i}{2} \rfloor$, in such a way that their relative ordering remains unchanged. Second, we permute the elements $c_{\pi(k)}$ for $1 \leq k \leq j$ and $c_{\pi(K-k+1)}$ for $1 \leq k \leq i-j$ so that their respective coefficients are oppositely sorted. Third, for all $1 \leq j \leq i$ we replace the j-th smallest element among the mentioned elements with c_j. This transforms the expression on the left-hand side into the one on the right-hand side, in such a way that its value never increases during the process. Indeed, in the first step, the coefficient of each element doesn't increase (easy check – this boils down to the fact that the sorted version of the multiset $\{j-k+1 \mid 1 \leq k \leq j\} \cup \{i-j-k+1 \mid 1 \leq k \leq i-j\}$ for $j = \lceil \frac{i}{2} \rceil$ pointwise dominates the sorted version of the same multiset for any other value of j). In the second step, the claim is immediate from the rearrangement inequality, whereas in the third step, the claim is obvious. This concludes the proof. $\square$

Finally, the problem can be solved by sorting the sequence $c_1, \ldots, c_K$ and evaluating the function f on the permutation π_0 from Claim 3, in time $\mathcal{O}(K \log K)$ (after the initial step of determining the sequence $c_1, \ldots, c_K$, which takes $\mathcal{O}(N \log N)$ time).

2 The solution in the case of many queries

Once we have established the solution of the problem in the case $Q = 1$, we may move on to the version with more than one query. The main idea is to use the so-called Mo's algorithm, which is nothing else but a way of ordering the queries which yields a small sum of sizes of the symmetric differences of intervals in neighbouring queries. More concretely, if we denote the intervals by (l_i, r_i) (0-based) for $1 \leq i \leq Q$, then we sort the queries according to the lexicographic ordering of the respective pairs $(\lfloor \frac{l_i}{B} \rfloor, r_i)$, where we set $B = \lfloor \sqrt{N} \rfloor$. In other words, we split the sequence into $\lceil \frac{N}{B} \rceil$ blocks of size B, and we split the queries into groups depending on the block to which the left endpoint of the interval belongs. In each group, we sort the queries in increasing order according to the right endpoint of the interval. It is easy to see that the aforementioned sum of sizes of the symmetric differences of intervals doesn't exceed $\mathcal{O}((Q+N)\sqrt{N})$.

How does this observation help us to solve the problem? If we process the queries in the described order, then for all $0 \leq i \leq N-1$ we can maintain a counter $freq[i]$ which denotes the number of occurrences of the value i in the current interval (under the assumption that we have previously performed coordinate compression on the whole sequence). When moving on to the next query, we can refresh the counters by simply iterating over the symmetric difference of the current interval and the next one.

It is clear that the sequence $c_1, \ldots, c_K$ from Section 1 consists precisely of the non-zero elements of the array $freq$. However, there can be many such elements, so we have to keep track of the array $freq$ by storing it in a compressed form. This can be done by storing for each $1 \leq i \leq N$ a counter *comp*$[i]$ which denotes the number of occurrences of the value i in the array $freq$. We are now interested in the non-zero elements of the array *comp*. How many such elements can there be? Observe that the sum of *comp*$[i] \cdot i$ over all $1 \leq i \leq N$ equals the sum of $freq[i]$ over all $0 \leq i \leq N-1$, which is at most N. Thus, if the array *comp* has L non-zero elements, then the sum of their indices is at the same time at least $1+2+\ldots+L = \frac{L(L+1)}{2}$ and at most N. Therefore, we have $L \leq \sqrt{2N}$, so *comp* has at most $\mathcal{O}(\sqrt{N})$ non-zero elements.

It remains only to efficiently evaluate the function f on the permutation π_0, for the sequence $c_1, \ldots, c_K$ given in the described compressed form. This can be done in time $\mathcal{O}(L)$ by performing a summation which involves the sum of squares of the elements of an arithmetic progression, on the condition that we can efficiently obtain the non-zero elements of the array *comp* in the order in which they appear in the array. This can be achieved by either storing the array *comp* in a map, or by storing the indices of the non-zero elements in a separate vector, which is then sorted as needed. This approach has time complexity $\mathcal{O}((Q+N)\sqrt{N}\log N)$. For the purposes of sorting, we can also use radix sort in order to recover the complexity $\mathcal{O}((Q+N)\sqrt{N})$. Even though the latter approach is faster in theory, it is the former approach that gives better results in practice. For more details concerning implementation, please consult the attached source code of the official solution.

3 Appendix: Inequalities used in the proofs

In the proof of Claim 3, we made use of the following (more or less standard) results:

Theorem 1. (Karamata's inequality) Let $f : \mathbb{R} \to \mathbb{R}$ be a convex function, that is to say, such that for all $x, y \in \mathbb{R}$, $t \in [0,1]$ the following inequality holds

$$f((1-t)x+ty) \leq (1-t)f(x)+tf(y).$$

Let $x_1, \dots, x_n$ and $y_1, \dots, y_n$ be real numbers such that $x_1 \geq \dots \geq x_n$, $y_1 \geq \dots \geq y_n$ and for all $1 \leq i \leq n$ we have

$$\sum_{j=1}^{i} x_j \geq \sum_{j=1}^{i} y_j,$$

with equality in the case $i = n$. Then the following inequality holds

$$\sum_{i=1}^{n} f(x_i) \geq \sum_{i=1}^{n} f(y_i).$$

Proof in the case $f(x) = x^2$. Let $p_i = \sum_{j=1}^{i} x_j$, $q_i = \sum_{j=1}^{i} y_j$ for $0 \leq i \leq n$. Then we have $p_0 = q_0 = 0$, $p_n = q_n$ and $p_i \geq q_i$ for all $0 \leq i \leq n$. Using summation by parts, we obtain that

$$\begin{aligned}\sum_{i=1}^{n} f(x_i) - \sum_{i=1}^{n} f(y_i) &= \sum_{i=1}^{n}(x_i - y_i)(x_i + y_i)\\ &= \sum_{i=1}^{n}[(p_i - p_{i-1}) - (q_i - q_{i-1})](x_i + y_i)\\ &= \sum_{i=1}^{n}[(p_i - q_i) - (p_{i-1} - q_{i-1})](x_i + y_i)\\ &= \sum_{i=1}^{n}(p_i - q_i)(x_i + y_i) - \sum_{i=0}^{n-1}(p_i - q_i)(x_{i+1} + y_{i+1})\\ &= \sum_{i=1}^{n-1}(p_i - q_i)(x_i + y_i) - \sum_{i=1}^{n-1}(p_i - q_i)(x_{i+1} + y_{i+1})\\ &= \sum_{i=1}^{n-1}(p_i - q_i)[(x_i - x_{i+1}) + (y_i - y_{i+1})] \geq 0,\end{aligned}$$

where the last inequality holds because $p_i - q_i, x_i - x_{i+1}, y_i - y_{i+1} \geq 0$ for $1 \leq i \leq n-1$. □

Theorem 2. (Rearrangement inequality) Let $x_1, \dots, x_n$ te $y_1, \dots, y_n$ be real numbers such that $x_1 \leq \dots \leq x_n$, $y_1 \leq \dots \leq y_n$. Then for all permutations σ of the set $\{1, 2, \dots, n\}$ the following inequalities hold

$$\sum_{i=1}^{n} x_i y_{n-i+1} \leq \sum_{i=1}^{n} x_i y_{\sigma(i)} \leq \sum_{i=1}^{n} x_i y_i.$$

Proof. Note that the first inequality follows by applying the second inequality to the sequence $-y_n, \dots, -y_1$ in place of $y_1, \dots, y_n$. Hence, it suffices to prove the second inequality. Choose a permutation σ which maximizes the value of the sum $\sum_{i=1}^{n} x_i y_{\sigma(i)}$. If there is more than one such permutation, choose the lexicographically smallest one. We claim that σ is the identity permutation, i.e. that we have $\sigma(i) = i$ for all $1 \leq i \leq n$. Indeed, if this is not the case, then there exists $i \in \{1, 2, \dots, n-1\}$ such that $\sigma(i) > \sigma(i+1)$. Then let $\sigma' = \sigma \circ \tau$, where τ is the transposition that swaps i, $i+1$. Then we have

$$\sum_{i=1}^{n} x_i y_{\sigma'(i)} - \sum_{i=1}^{n} x_i y_{\sigma(i)} = x_i y_{\sigma(i+1)} + x_{i+1} y_{\sigma(i)} - x_i y_{\sigma(i)} - x_{i+1} y_{\sigma(i+1)} = (x_{i+1} - x_i)(y_{\sigma(i)} - y_{\sigma(i+1)}) \geq 0.$$

Furthermore, σ' is lexicographically smaller than σ, which is a contradiction. □

L-Triominoes – solution

Domagoj Bradač

September 2nd, 2021

1 An $\mathcal{O}(2^W \cdot HW)$ solution using dynamic programming

We describe a dynamic programming solution which is standard for problems of this type. We build the tiling from the bottom up and from left to right as follows. Suppose $1 \leq r \leq H, 1 \leq c \leq W$ and we have so far tiled every square (r', c') with $(r', c') \leq (r, c)$ (and potentially some other squares in the process). In the next step, we put a triomino covering the current square (r, c). Note that there are at most four possible ways to do this (see Figure 1)

What do we need to keep track of? By assumption, we have already tiled all squares $(r', c') < (r, c)$. It is easy to check that the only other squares we might have additionally tiled are the following:

- (r, c') with $c' \geq c$;
- $(r + 1, c')$ with $c' \leq c$.

(See Figure 2.)

In total, there are $W + 1$ squares which might additionally be tiled. Hence, we can describe the state of our dynamic program as (r, c, m), where (r, c) is the position of the current square and m is a bitmask of $W + 1$ bits describing which additional squares are tiled. As mentioned, the DP transition can be done in time $O(1)$, yielding the total running time $\mathcal{O}(2^W \cdot HW)$.

2 The directed graph on bitmasks

To discuss the remaining solutions, we introduce a directed graph D. Its vertex set consists of all 2^W bit-vectors of length W. Such a vector represents a partially tiled row: 1's correspond to occupied squares and 0's to squares which need to be tiled. For two bit-vectors (u, v), the edge $u \to v$ appears in D if we can complete the tiling of the row partially tiled according to bitmasks u such that the next row has partial tiling

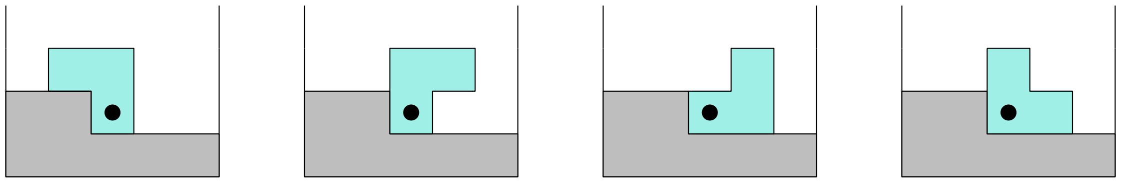

Figure 1: Four possible ways to continue tiling. Gray squares mark the previously tiled squares and the dot marks the current square (r, c).

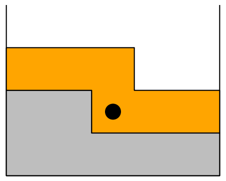
Figure 2: Squares we need to keep track of in a DP solution

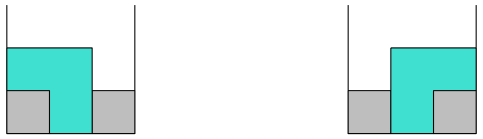
Figure 3: There are two ways to continue tiling from the bitmask 101.

corresponding to bitmask v. E.g., suppose $W = 3$ and $u = 101$. There are two ways to finish the tiling of this row (Figure 3). Hence, there are two edges going from 101: $(101, 110)$ and $(101, 011)$.

Note that we can construct the entire digraph D in time $\mathcal{O}(2^{2W} \cdot W)$ using the abovementioned dynamic programming approach.

How can we use this digraph to obtain a faster solution? Let $1 \leq r \leq H$ be an arbitrary row. Suppose we can tile, bottom-up as before, the first $r - 1$ rows, possibly tiling some additional squares in row r. Let S be the set of squares in row r which are either tiled or initially missing and let v_S denote the vertex in D corresponding to S. We say this tiling produces the set S in row r. Define

$$U_r = \left\{ v_S \in V(D) \mid \text{ there is a tiling which produces } S \text{ in row } r \right\}.$$

Now let $v_S \in U_r$ and consider a tiling of the first $r-1$ rows which produces S in row r. Let T be an arbitrary subset of squares in row $r+1$ and let v_T its corresponding vertex in D. Let us for the moment ignore any squares missing in row $r+1$. Then, by definition of D, we can extend our tiling such that the set of tiled squares in row $r+1$ is T if and only if (v_S, v_T) is an edge in D. Consider a set T we can achieve this way. If it contains any of the squares missing in row $r+1$ then this tiling is invalid. Otherwise, we add the missing squares to T, and add T to the set U_{r+1} of possible configurations in the next row.

A naïve implementation of the above yields an algorithm with running time $\mathcal{O}(2^{2W} \cdot H + K)$, worse than the dynamic programming approach.

However, this approach can be vastly improved. Let us first identify the simpler problem we want to attack. Suppose $1 \leq r < s \leq H$ and there are no tiles between rows r and s, inclusively. Given U_r, we need to be able to quickly determine U_s. In the setting of our directed graph D, it is enough to solve the following subproblem $O(k)$ times.

Problem 2.1. Given a set $U \subseteq V(D)$ and a positive integer ℓ, determine all vertices $v \in V(D)$ for which there exists a vertex $u \in U$ and a walk of length ℓ from v to u.

3 Matrix multiplication – $\mathcal{O}(2^{3W} \cdot K \log H)$

The first way to solve this problem uses a standard technique of matrix multiplication. It can be implemented to run in time $\mathcal{O}(2^{3W} \cdot K \log H)$. Using bitwise operations one can achieve a very small constant factor (at most 1/64 on modern architectures) compared to the usual implementation. Additionally, this approach can be sped up to achieve running time $\mathcal{O}(2^{3W} \cdot \log H + 2^{2W} \cdot K \log H)$, which is left as an exercise to the reader. However, we decided not to award these speedups any additional points as they represent technical improvements to the solution which do not introduces any of the ideas required to solve the problem fully.

4 The full solution – $\mathcal{O}(2^W \cdot WK \cdot C(W))$.

The function $C(W)$ is a parameter which we will introduce formally later. To motivate our solution, we start with a definition.

Definition 4.1. The *period* of a directed graph is the greatest common divisor of the lengths of all (directed) cycles in the graph.

We immediately state a useful fact which in some form can be found in most textbooks about Markov chains (e.g. see [**?**]).

Fact 4.2. *Let $F = (V, E)$ be a strongly connected directed graph with period k. Then, V can be partitioned as $V = V_1 \dot\cup V_1 \dot\cup \ldots \dot\cup V_k$ such that for any $1 \le i \le k$, the vertices in V_i only have outgoing edges towards V_{i+1}, where we denote $V_{k+1} = V_1$. Additionally, there exists a constant $C = C(F)$, such that the following holds. Let $1 \le i, j \le k$, and let $u \in V_i, v \in V_j$ be arbitrary vertices. Then for any integer $\ell \ge C$, there is a walk from u to v of length ℓ if and only if $j - i \equiv \ell (\operatorname{mod} k)$.*

From now on suppose that D is strongly connected. This is actually not true and we will later describe the structure of D. However, solving the problem under the assumption that D is strongly connected captures the relevant ideas. Suppose we are given the period k of D as well as a partition $V(D) = V_1 \dot\cup V_1 \dot\cup \ldots \dot\cup V_k$ and the integer C from Fact 4.2. Then for given U and ℓ we can answer Problem 2.1 as follows. If $\ell \ge C$, find the set I of indices $1 \le i \le k$, such that $U \cap V_i \ne \emptyset$. For each $V_i, i \in I$, add the set $V_{(i+\ell) \operatorname{mod} k}$ to the answer. This can be done in time $\mathcal{O}(|D|) = \mathcal{O}(2^W)$. If $\ell < C$, we can answer Problem 2.1 using dynammic programming in time $\mathcal{O}(2^W \cdot \ell W)$ as described in Section 1. Combining the two and recalling that we need to solve $\mathcal{O}(K)$ instances of Problem 2.1, we derive an algorithm with time complexity $\mathcal{O}\big(2^W \cdot WK \cdot C(W)\big)$, where $C = C(W)$ is the mentioned constant.

Finally, we describe the describe the structure of D. The cases $W = 2, 3$ have a different structure than larger values and these cases need to be handled separately. We leave this as a simple exercise for the reader. (Or see the attached source codes.) For $W \ge 4$:

a) If W is odd, there are two special vertices, those corresponding to masks $1010 \ldots 101$ and $0101 \ldots 010$. The former has no incoming edges (it is impossible to achieve this configuration by tiling the previous row) while the latter has no outgoing edges (it is impossible to continue the corresponding tiling). From now on we ignore these vertices and all of the following statements come with the exception of special vertices.

b) If W is not divisible by 3, then there is a single weakly connected component, it is also strongly connected, and has period 3. If W is divisible by 3, then there are three connected components V_0, V_1, V_2, all are strongly connected and have period 1. The partition from Fact 4.2 in the former case, and the connected components in the latter, correspond to residues modulo 3 of the number of 1's in the corresponding bitmasks.

c) For every non-trivial strongly connected component D' of D, we have $C(D') \le 11$. :-)

All of these statements can be verified with a piece of code which runs for a few minutes or less. In addition, the first point can easily be found with pen and paper. The second point can be guessed by considering residues modulo 3. The third point is difficult to guess precisely and is the most difficult to verify. That being said, obtaining (or guessing) a reasonably good upper bound on C is enough to obtain full points. Having determined the structure of D, we can hard-code it into our program and we do not need to explicitly generate the digraph D.

5 Subtasks

The following presents the subtasks for this problem and their intended solutions.

Subtask	Score	Constraint	Intended solution
1	10	$H \leq 1000$	Dynamic programming
2	7	$K = 0$	Pen and paper analysis
3	11	$W \leq 3$	This subtask can be solved using matrix multiplication. However, it can also be solved by doing a pen and paper analysis of the directed graph D. Such a solution of the subtask can lead to a full solution.
4	17	$4 \leq W \leq 6$	Matrix multiplication
5	35	$7 \leq W \leq 13$	Full solution
6	20	No special constraints	Full solution, correct handling of the special cases $W = 2, 3$.

Newspapers – solution

Ivan Paljak

September 5, 2021

1 Branko survives if there is a cycle

The easiest observation in this problem is that the input graph at least has to be a tree in order for Ankica to have a successful strategy.

To prove this, we can generalize the explanation of the second example. More precisely, if we assume the graph contains a cycle, we can imagine Branko starting at any node of the cycle that is different from a_1. Since each node in the cycle has two neighbours that are also part of the cycle, Branko can choose a node in the i-th turn that's different from a_i.

2 A brute force approach

Notice that Ankica only needs to keep track of the set of nodes Branko can occupy before each turn. Of course, before the game, she assumes that Branko can occupy any node.

This leads us to represent the game as a graph where each node corresponds to a set of nodes that Branko can occupy at some turn. The graph will be directed, and the edge from node u to node v will mean that Ankica can make a guess such that if Branko could be located at a set of nodes represented by u before a turn, he could be located at a set of nodes represented by v after the turn.

Obviously, we want to find the shortest path from a node representing the set of all nodes to a node representing an empty set.

This approach was fast enough to solve the first subtask.

3 Branko doesn't survive on a chain

Now let's show that Ankica can always catch Branko on a chain.

Suppose node labeling is consistent with the second subtask, and let's color the nodes with even labels white, and nodes with odd labels black.

The cases when $N = 1$ and $N = 2$ can be trivially solved, we assume $N \geq 3$ in the rest of the section.

Ankica will make a total of $2N - 4$ guesses: $(2, 3, \dots, N-1, N-1, N-2, \dots, 2)$. It is easy to show that she will catch Branko during the first $N - 2$ turns if he starts on a white node, and that she will catch him during the last $N - 2$ turns if he starts on a black node.

Suppose Branko starts on a white node. In that case, during the first $N - 2$ turns, Ankica and Branko will occupy a node of the same color. It's easy to see that in order for Branko to survive the i-th turn, it must be true that $b_i > a_i$. This is because $b_1 > a_1$ (otherwise he is caught in the first turn), and in order for b_i to be less than a_i, we would have to have some $b_j = a_j + 1$ ($j < i$), which is not possible because b_j and a_j must be of the same parity. Since $a_{N-2} = N - 1$, there is no node of the same color with greater label, and we can conclude that Branko will be caught during the first $N - 2$ turns if he starts on a white node.

If Branko starts on a black node, he will occupy a node of different color than Ankica during the first $N - 2$ turns. Since the $(N - 1)$-st turn repeats the node, Branko will occupy the same colored node during the last $N - 2$ turns. Thus, the last $N - 2$ turns when Branko starts on a black node are analogous to the first $N - 2$ turns when he starts on a white node, just in the opposite direction.

We will later show that this construction is indeed optimal.

A commonly submitted suboptimal solution for a chain with similar reasoning was $(1, 2, \ldots, N, N, \ldots, 1)$.

4 Branko survives on a star with three arms of length three (S')

Here we will show that Ankica cannot catch Branko on a specific star-shaped tree with 10 nodes. We will label the central node with label 10, the three arms with labels $(1, 2, 3)$, $(4, 5, 6)$, and $(7, 8, 9)$ as depicted below.

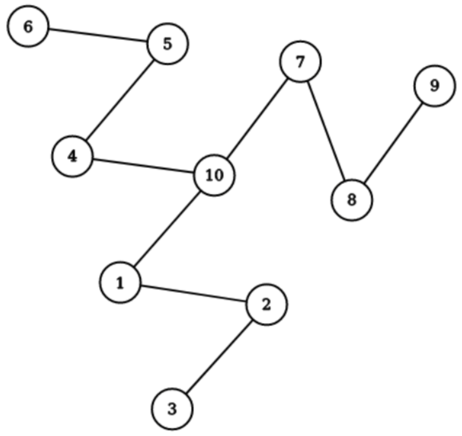

Figure 1: Star with three arms of length three

We will show that Branko can survive by spending his even turns either at the central node (10), or at the middle arm nodes $(2, 4, 8)$. Loosely speaking, Branko will attempt to stay as close to the central node as possible during the entire game.

Therefore, if Branko finds himself next to the central node, i.e. in one of nodes $(1, 4, 7)$, he will move to a central node in the next turn unless Ankica makes that guess. In that case, he will move to a middle arm node, i.e. one of $(2, 4, 8)$.

If Branko finds himself on a middle arm node during a turn, he will move towards node 10, i.e. to one of the nodes $(1, 4, 7)$, unless that is Ankica's next guess. In that case, he is forced on of the leafs $(3, 6, 9)$.

Obviously, if Branko finds himself at nodes $(3, 6, 9)$, he will be forced to move to nodes $(2, 4, 8)$ respectively.

The key idea is that if Branko finds himself in the central node, he will move to a branch which Ankica will not guess in the next turn, or the next branch in which Ankica wlil guess two closest nodes to the central in succession, i.e. $(1, 2)$, $(4, 5)$, or $(7, 8)$.

Notice that we only need to show that Branko will never be at a leaf node when Ankica guesses its neighbour on the next turn. In order for Branko to visit a leaf node, that means that Ankica must guess the node of the same arm next to the central one in that turn. If Ankica catches Branko on the next turn, that would imply her guessing two successive nodes from the previous section. For that to be possible, she would have to have guessed such consecutive nodes of some other arm after his last visit to node 10. This is in contradiction with the rules because during her guesses, he would have returned to the central node.

5 Branko doesn't survive on a tree that doesn't contain S'

In general, it should be easy to conclude that if Branko can't survive on a graph G, he also can't survive on a graph H with an induced subgraph isomorphic to G. In other words, Branko will escape in any tree containing S'.

Interestingly enough, it turns out that Ankica can always catch Branko on a tree that doesn't contain S'. We can show that by employing a similar strategy to the one explained in a section that deals with a chain.

Let's observe the longest path (diameter) of a tree, and assume it's nodes are labeled 1, 2, ..., l, where l is the length of the diameter. We will also generalize the coloring from the chain analysis. A node will be colored white if his distance from node 1 is even, otherwise it will be colored black. Note that this coloring induces a bipartition of the tree.

Note that there must not be any node adjacent to nodes 1 and l, that are not 2 and $l-1$ respectively, as this would contradict with the diameter definition.

Note that there must not be any chain of length 2 with nodes not part of the diameter, that is connected to node 2 and $l-1$, as this would also contradict with the diameter definition.

Finally, note that there must not be any chain of length 3 with nodes not part of the diameter, that is connected to any node of the diameter because that would either contradict with the diameter definition or imply the existence of a subtree isomorphic with S'.

This means that all non-leaf nodes are either in the diameter or adjacent to a node in the diameter. This allows us to construct a strategy for Ankica that is very similar to the strategy employed for a chain. Here, Ankica will "move" from node 3 to node $l-2$ and back, but will also take into account all neighbouring non-leaf nodes.

More precisely, when after entering a node i on her route to node l, she will visit all non-leaf neighbours of i that are not $i+1$, before continuing to $i+1$. She will also visit the node i between each two successive visits to the neighbouring nodes. After visiting node l, she will reverse the walk back to node 2.

By similar reasoning to the one from section 3, we can inductively show that Ankica will catch Branko during the first half of the walk if he starts on a white node, and that she will catch him during the second half of the walk if he starts on a black node.

6 The optimal strategy

In the strategy from the previous section, we can differentiate between three types of nodes: leaves, inner nodes (i.e. nodes $3 \dots l-2$), and nodes adjacent to inner nodes that are not leaves.

Alice never guesses the leaf nodes, 2 times guesses each node adjacent to the inner nodes, and $2f(v)-2$ times guesses a path node v with $f(v)$ neighbours that are not leaves. We will show that Branko can survive if any of these nodes is guessed any less than the described number of times.

Case 1. Node v adjacent to the inner nodes.
Since $a_i = v$ for at most one value of i, $a_i \neq v$ must hold for either all even or all odd values of i. Therefore, there is a way for Branko to move between v and a node adjacent to v the whole time without him being caught.

Case 2. Node v that is an inner node.
Branko will again try to stay close to v. Let u_1, u_2, $\ldots$, u_k be the neighbours of v, and u'_i be the neighbour of u_i that is not equal to v.

The number of times Ankica guesses v is at most $2k - 3$, therefore she guesses v on at most $k - 2$ even and at most $k - 2$ odd turns. We can assume she guesses v on at most $k - 2$ even turns. We will put Branko at v on every odd turn where $a_i \neq v$.

Now, we can divide Ankica's guesses into chains of longest sequences where she guessed v at every odd turn. In other words, we are looking at some a_i, a_{i+2}, $\ldots a_{i+2l}$ all equal to v, but $a_{i-2} \neq v$ and $a_{i+2} \neq v$. We can choose some u_j different from all $a_{i-1}, a_{i+1} \ldots, a_{i+2l+1}$. This is possible because $l < k - 2$. At all odd turns in this sequence, Branko will move to u'_j.

We now know all of Branko's whereabouts for all odd turns.

For even turns we consider where Branko is located at neighbouring odd turns. If on at least one of those turns he is at u'_j, he will be at u_j on the corresponding even turn. If at both neighbouring turns he is at v, he can simply choose any u_j that Ankica will not guess on a said even turn.

This shows that the strategy from the previous section is indeed optimal.

Disclaimer: The explanations in this editorial are meant to be illustrative, rather than rigorous. Rigorously proving each claim entails some cumbersome handling of special cases (e.g. small trees with $l \leq 2$), certain boundary checks (e.g. out of bounds indices), etc. We believe these shouldn't pose a serious problem to the reader that has grasped the main ideas.

Stones – solution

Ivan Žufić, Daniel Paleka

September 4th, 2021

This problem is a twist on the standard Nim game, with the catch that the pile to take stones from is adversarially chosen by the opponent.

However, the solution has nothing to do with the Sprague-Grundy theory and bitwise operations. Instead, we will only use the standard approach of separating the states into winning and losing states.

Definition 0.1. *A state is the set $(a_1, \dots, a_N)$ of piles, together with the index i of the pile the opponent selected.*

Let's write down the inductive definition of losing and winning state in the context of this game:

1. The states with all piles empty are losing;
2. A state is winning if and only if the turn player can make a move that reaches a losing state;
3. A nonempty state is losing if and only if the turn player must make a move that reaches a winning state.

The above defines a valid partition of all states by induction on the number of stones.

0.1 The first two subtasks

For the first subtask, we can explicitly enumerate all $(M+1)^N \cdot N$ states (some are equivalent, but $8^7 \cdot 7$ is small enough) and brute force the described induction using dynamic programming.

A good strategy for solving this problem is: implement the first subtask, get the points on the evaluator, and then try to find some pattern.

The first thing that is easy to see from the printed moves is that the turn player should always take everything or leave exactly one stone in the chosen pile. Using this observation, we can implement a dynamic programming in time complexity $O(3^N)$.

0.2 The full solution

We can find a characterization of the losing states.

Let A be the number of piles containing exactly one stone.
Let B be the number of piles containing strictly more than one stone.
Let H be the number of stones in the currently selected pile.

If $B = 0$, we win if and only if A is odd. It's easy to prove this by induction.

Otherwise, let $K \geq 1$. Partition the post-pile-selection states into three types:

- Let *type-W* states be such that A is even;
- Let *type-L* states be such that A is odd and $H = 1$.
- Let *type-O* states be other states, i.e. A is even and $H > 1$.

Claim. Every *type-W* state is winning, and every *type-L* state is losing.

Proof. We have two cases.

- First case: we're in a *type-W* state.

  If $H \neq 1$ and $B = 1$, then we can remove all stones from the current pile, which is clearly a winning move, since A is even.

  If $H = 1$, we can remove the only stone in that pile and select any other pile with exactly one stone. This is possible, since A is even. This leads to a *type-L* state with smaller total number of stones.

  Otherwise, if $H \neq 1$ and $B > 1$, then remove $H - 1$ stones from the pile, and force the next player to select the last stone from current pile. This also leads to a *type-L* state with smaller total number of stones.

- Second case: we're in a *type-L* state.

  Note that $H = 1$ and A is even, so the next state will have $B > 0$, odd A, and it will have smaller total number of stones.

Every *type-W* state leads to a *type-L* state or some losing state with $B = 0$, and every *type-L* state leads to a *type-W* state, and in every transition total number of stones is smaller, so *type-W* are winning states, and *type-L* are losing positons.

$\square$

Remark. *Type-O* states are winning, but that is not important at all, since it is guaranteed that the starting state is winning regardless of the first selected pile.

To solve the problem, implement the moves used to prove the Claim. It is enough to take $O(N)$ time per move, that is, $O(NM)$ time in total.

Tortoise – solution

Josip Klepec

September 4th, 2021

1 Setup

Definition 1.1 (Area). *An* Area *is a maximal continuous subsequence of locations with at least one candy and no playgrounds.*

Lemma 1.1. *Tom buys candies in the order of increasing location.*

Proof. We can easily prove that it is never optimal to first buy a candy from a later Area and then return to buy a candy from a previous Area.

Then we can prove that in each Area we will buy candies in order. This is left as an exercise, as the proof is very similar to the proof of the the next Lemma. □

Lemma 1.2. *In each Area, Tom will first take some candies to the left playground (possibly zero) and then some candies to the right playground.*

Proof. Assume he takes a candy to the right playground, then takes the next one to the left.

Let $a < b \leq c < d$ be the locations of the playground, the first and the second cady, and the right playground.

- If $d - c \leq b - a$, we can take both candies to right playground and achieve better result (less time spent) and arrive at the same time to the second candy.
- Else, we can take the first candy to the left and second candy to the right because

  $$(d-b) + (d-a) + (c-a) > (b-a) + (d-a) + (d-c).$$

  (we spend less time in total); and

  $$(d-b) + (d-c) > (b-a) + (c-a)$$

  (we will arrive earlier to the second candy).

Doing this exchange a finite number of times we can achieve that we take some prefix of candies to the left playground and some suffix to the right. □

Lemma 1.3. *In each Area with a right playground, if Tom takes at least one candy, he will take at least one to the right playground.*

Proof. If he takes candies only to the left playground, he can just take the last one to the right playground. □

Corollary 1.4. *In each Area with a right playground, Tom will take at least one candy.*

Definition 1.2 (Left cycle)**.** *Tom takes a candy, brings it to the left playground on his left and comes back.*

Definition 1.3 (Right cycle)**.** *Tom is located at the right playground of some Area. He goes back to pick up the candy and brings it to the playground.*

Definition 1.4 (Candy walk)**.** *Tom buys a candy. Walks to the right playground and drops it there.*

Lemma 1.5. *If a candy is closer to the right playground, it will not be involved in a Left cycle and vice versa.*

Proof. Obviously only one of those things can happen. With exchanges we can keep the solution valid with less time spent in this Area. □

Lemma 1.6. *If a candy is at the same distance to both playgrounds, it can be shown that it doesn't matter which cycle we do with it. We can always do left for example.*

2 Greedy

Greedy algorithm works as follows.

For a candy i we use x_i to denote its position and w_i for twice the distance to its closest playground. We order the candies such that $w_1 \leq w_2 \leq \dots$

We consider candies and its cycle in order. If adding that cycle keeps solution valid, we add that cycle. Multiset of cycles is valid if we can manage to buy everything in time and in each Area with right playground we can do one Candy walk (we can carry some candy to the right playground after left cycles and before right cycles).

3 Proof

Lemma 3.1. *Tom makes exactly one candy walk in any area which has a playground to its right and has at least one candy.*

We say a set of candies S is valid if Tom can pick up this set of candies, each candy in one cycle and in addition one candy from each non-candiless area with a playground to its right.

Let S_G/S_O be the set of candies picked up with a cycle by the greedy/optimal algorithm, respectively. For a candy i we use x_i to denote its position and w_i for twice the distance to its closest playground. We order the candies such that $w_1 \leq w_2 \leq \dots$

Let $Y = S_G \setminus S_O$ and let $i = \min Y$. Then it is enough to find a valid set S'_O with $|S'_O| = |S_O|$ and $\min S_G \setminus S'_O > i$. Let $X = S_O \setminus S_G$. Observe that, by our greedy algorithm, we have

$$\min X > \min Y = i. \tag{1}$$

We consider several cases:

- There is no playground to the right of x_i.

  Let j be leftmost candy in X. We have $w_j \geq w_i$ by (1) and set $O' = O \setminus \{j\} \cup \{i\}$.

  This is trivial case.

- Cycle i is a Left cycle and Candy walk in optimal algorithm is to the left or at the same candy as i.

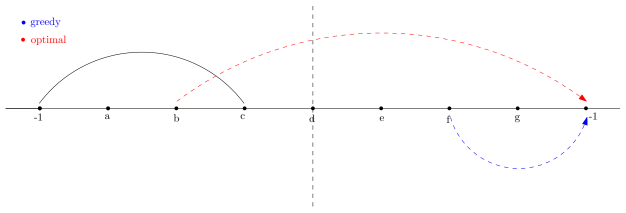

Figure 1

In this example $x_i = c$

If there exist a cycle right of i and left of Candy walk in greedy in X (locations c, d, e, f), then we remove leftmost such j cycle and it becomes a new candy walk.

Else let j be the leftmost candy in X. New Candy walk will be the same as in greedy (location f).

In both cases we have $w_j \geq w_i$ by (1) and set $O' = O \setminus \{j\} \cup \{i\}$.

- Cycle i is a Left cycle and Candy walk in optimal algorithm is to the right.

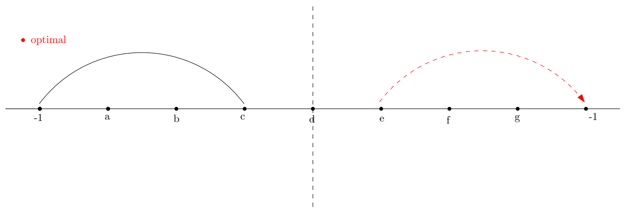

Figure 2

Let j be the leftmost candy in X.

If it is left of i then Candy walk will stay the same (we removed longer cycle that happened before).

Else, j is in some Area. If it is Left cycle, Candy walk in that Area can stay the same (because we added smaller cycle than we added and both of them happen before). But if it is the Right cycle, then that Area can use the same Candy walk as greedy. Because prefix is the same.

Again we set $O' = O \setminus \{j\} \cup \{i\}$.

- Cycle i is a Right cycle.

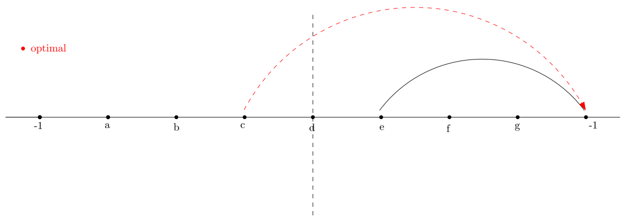

Figure 3

Candy walk in optimal solution cannot be to the right of i because that cycle would then also be in greedy (greedy takes some suffix of Right cycles).

Let j be the rightmost left of i u X if such exists. If the candy walk was at x_i and needs to be moved to the left then we can set candy walk to any available candy in this Area. Such will exist because we will remove either the last left cycle in this Area or some bigger cycle. And candy will be available for a candy walk because greedy algorithm is also valid.

Otherwise let j be the leftmost in cycle in X. This case is similar to the previous one.

4 The algorithmic parts

4.1 First subtask

This can be achieved with dynamic programming and bitmasks (with state $mask, time, position$). Or just using the assumption that Tom will take candies in order and then using solely bitmasks.

4.2 Second subtask

This can be achieved with dynamic programming keeping time, position, position of last taken candy and a boolean saying whether we have a candy in hand.

4.3 Third subtask

This can be achieved with dynamic programming. We are using result of Lemma 1.2. Having boolean did we start taking candy to the right playground in current Area.

4.4 The full solution

We simulate the greedy algorithm described in 2. To get full points one should use some kinda of data Structure to check validity when adding a cycle.

For the fourth subtask, it is enough to implement this validity check in linear time. And for full points we can keep how many seconds before Wilco we get to every shop.

Wells – solution

Domagoj Bradač

September 4th, 2021

1 Notation

We denote the given tree by $T = (V, E)$. We write $P(u, v)$ for the unique path from u to v in T and $d(u, v)$ for the number of edges in $P(u, v)$. The diameter of T is a longest (simple) path in T. We denote a fixed diameter of T by D and imagine the edges not on D to be directed away from D. Given a vertex $v \in T$, we write $r(v)$ for the closest vertex of v on D (if $v \in D$, then $r(v) = v$). We write T_v for the subtree rooted at v following the directed edges away from the diameter. We denote by $h(v)$ the height of T_v and by b_v some vertex on the "bottom" of T_v, that is, $b_v \in T_v$ and $d(v, b_v) = h(v)$.

We define a *partial colouring* of T as follows: we say that a vertex is coloured red if it is marked, blue if it is not marked and uncoloured if we have not yet decided whether or not it is marked. Given a partial colouring c, a path of length $k - 1$ is said to be invalid with respect to c if it contains more than one red vertex or all its vertices are coloured blue, otherwise it is *valid*. A (partial) colouring is invalid if it contains an invalid path of length $k - 1$, otherwise it is valid. A colouring is *complete* if all vertices are coloured (red or blue). A partial colouring c is an *extension* of a partial colouring c' if $c(v) = c'(v)$ for any vertex v which is coloured under c'.

2 Observations

Recall the following fact about a diameter of a tree.

Fact 2.1. *Suppose z_ℓ, z_r are endpoints of a diameter of T. Then for any vertex $v \in T$:*

$$\max_{u \in T} d(v, u) = \max\{d(v, z_\ell), d(v, z_r)\}.$$

Let $D = (d_1, d_2, \dots d_L)$ be a diameter of T with endpoints $z_\ell = d_1$ and $z_r = d_L$. If $L < k$, then there is no path of length $k - 1$ in T, so the answer is YES and the number of colourings 2^n. Now, assume $L \geq k$. Then, there are k ways to colour the diameter, for any $1 \leq i \leq k$, we colour the vertices $d_i, d_{i+k}, d_{i+2k}, \dots$ red and the other vertices of D blue.

We say a path P is *crossing* if both its endpoints are not in D and it contains a vertex in D.

Claim 2.2. *Let c be an arbitrary complete colouring of T. Then c is valid if and only all non-crossing paths are valid with respect to c.*

Proof. Suppose this is not the case, and consider a crossing path P of length $k - 1$ with endpoints u, v. Assume without loss of generality that $z_l, r(u), r(v), z_r$ appear in this order on D, with possibly some of these

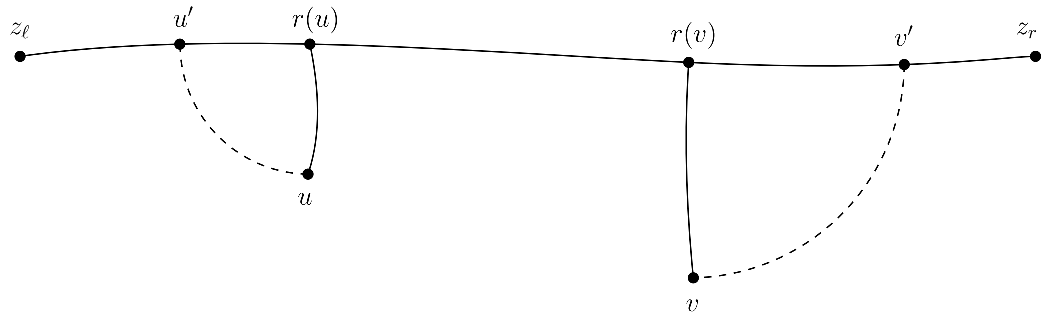

Figure 1

vertices being identical. Since D is a diameter, there exists a vertex $v' \in D$ such that $d(u, v') = d(u, v) = k$ and $r(v)$ lies between $r(u)$ and v'. Analogously, there exists a vertex $u' \in D$ such that $d(u', v) = d(u, v) = k$ and $r(u)$ lies between u' and $r(v)$ (see Figure 1). Note that

$$d(u', v') = d(u, v') + d(u', v) - d(u, v) = k + k - k = k.$$

For vertices x, y let $f_c(x, y)$ denote the number of red vertices in $P(x, y)$ under the colouring c and note that a path $P(x, y)$ of length $k - 1$ is valid if and only if $f_c(x, y) = 1$. Then:

$$f_c(u, v) = f_c(u, v') + f_c(u', v) - f_c(u', v').$$

Under our assumptions, the paths $P(u, v'), P(u', v)$ and $P(u', v')$ are valid, implying $f_c(u, v) = 1$. In other words, $P(u, v)$ is valid as well. □

Consider a vertex v not on the diameter such that

$$h(v) + \max\{d(v, z_l), d(v, z_r)\} < k - 1. \tag{1}$$

As D is a diameter, this implies there is no path of length $k - 1$ containing v. Hence, v can be coloured arbitrarily without changing the validity of the colouring. We delete all vertices satisfying (1) and multiply the number of colourings by 2 for each deleted vertex. From now on, we assume there are no vertices satisfying (1).

Claim 2.3. *Let c be a partial colouring and let v be a vertex such that there exists a path of length at least $k - 1$ containing v and a vertex coloured red under c. Then in any proper extension of c, the colour of v is the same. In other words, if such a path exists, the colour of v is uniquely determined.*

Proof. We need to show that the colour of v is uniquely determined by the partial colouring c. This follows immediately. Let P be a path of length at least $k - 1$ containing v and a red vertex under c. Observe that every k^{th} vertex on P must be red. Since there is a red vertex u on P, it follows that v is red if and only if $d(u, v) \equiv 0 \pmod{k}$. □

Claim 2.4. *Suppose there is a path of length $k - 1$ not touching D. Then, there are at most two ways to colour D which can be extended to a complete valid colouring.*

Proof. Let $P = P(u, v)$ be a path of length $k - 1$ not touching D. As D is a diameter, this implies:

$$d(u, z_l), d(u, z_r), d(v, z_l), d(v, z_r) \geq k - 1.$$

As P does not touch D, we have $r(u) = r(v) = r$. Assume that $d(r, u) \geq d(r, v)$, implying $d(r, u) \geq (k-1)/2$. Observe that on $P(u, z_l)$ every k^{th} vertex must be red and the same holds for $P(u, z_r)$. It is easy to see that this can only happen if r is red or k is odd and r is in the middle of two consecutive red vertices on D. □

Definition 2.5. Let U be a tree rooted at a vertex r and let t be a nonnegative integer. Define the depth of a vertex $v \in U$ as $d(r, v)$. A complete colouring of U is a *t-covering* of U if:

- every red vertex is at depth at most t; and
- for every vertex u such that $d(r, u) \geq t$, there is exactly one red vertex on $P(r, u)$.

Definition 2.6. We call a vertex $v \notin D$ *pinned* if

$$h(v) + \min\{d(v, z_\ell), d(v, z_r)\} \geq k - 1, \tag{2}$$

and *loose*, otherwise.

Claim 2.7. *Let $v \notin D$ be a loose vertex. Without loss of generality, assume $h(v) + d(v, z_\ell) < k - 1$ and $h(v) + d(v, z_r) \geq k - 1$. Let c be a valid partial colouring in which the following vertices are coloured:*

- *all vertices in D,*
- *all vertices on the path $P(v, r(v))$ except v.*

Consider extensions of c obtained by colouring the vertices of T_v in some way. Then

a) if there is a red vertex in c on the path $P(v, z_r)$, there is a unique valid extension,

b) otherwise an extension is valid if and only if it forms a t-covering of T_v with $t = k - 1 - d(v, z_r)$.

Proof. We consider the two cases in order.

a) Then for any $u \in T_v$, we have $d(u, z_\ell) + h(u) < k - 1$, implying $d(u, z_r) + h(u) \geq k - 1$. By assumption there is a red veretx on $P(u, z_r)$ so by Claim 2.3, the colour of u is uniquely determined.

b) Note that any non-crossing path of length $k - 1$ with one endpoint in T_v has the other endpoint in $P(r(v), z_r)$. This follows directly from $d(u, z_\ell) < k - 1$ and the fact that D is a diameter. Now consider a path $P(u, w)$ of length $k - 1$ where $u \in T_v$ and $w \in P(r(v), z_r)$. By assumption there are no red vertices in $P(v, z_r) \setminus \{v\}$. Hence, for c' to be valid there must be exactly one red vertex in $P(u, v)$. In other words, c' forms a t-covering of T_v. It is easy to see that the converse also holds: if an extension of c obtained by colouring T_v forms a t-covering of T_v, then it is valid. □

Let us summarize what we have shown so far. We may delete all vertices satisfying (1). Given a colouring of D, consider the vertices $v \in T \setminus D$ which are loose but their parent is on D or is pinned. Then, either the colouring of T_v is uniquely determined or there is an integer t such that the colouring of T_v is any t-covering of T_v. The colours of all other vertices are uniquely determined by the colouring of D. Now, consider a colouring c satisfying these properties and suppose P is a path of length $k - 1$ invalid under c. By Claim 2.2, P is non-crossing. Recall that we first colour D and then extend this colouring to other vertices in a way that paths touching D are valid, so by construction, P does not intersect D. By Claim 2.4, if such a path P exists there are at most two ways to colour D and these two ways can be easily determined. Consider one such colouring and extend it to all pinned vertices in a unique way and to all loose vertices as described by Claim 2.7. By construction, any path invalid path under such a colouring is disjoint from D and contains only pinned vertices.

3 The algorithm

First, remove all vertices satisfying (1) and multiply the number of colourings by 2 for each deleted vertex. For any vertex $v \notin D$ such that v is loose but its parent is on D or is pinned, using dynamic programming calculate the number x_v of t-coverings of T_v with the appropriately chosen value of t. Then, for a colouring of D for which the colouring of T_v is not uniquely determined (as given by Claim 2.7), multiply the number of ways to extend this colouring by x. Finally, if there is a path of length $k-1$ not touching D, then there are at most two colourings of D and for each of them we can easily check whether they can be extended to a complete colouring.

The algorithm can be implemented to run in time $\mathcal{O}(n)$. Let us elaborate. Finding the diameter and determining which vertices satisfy (1) and which are pinned can easily be done in linear time. Computing the number of t-coverings is done in linear time using dynamic programming. If there is a path of length $k-1$ not touching D there are at most two colourings of D we need to consider. The validity and number of extensions of each of these can be done in linear time. Finally, assume there is no path of length $k-1$ disjoint from D. Consider a vertex $v \in T \setminus D$ which is loose but its parent is on D or pinned, that is, v is a vertex for which we need to calculate the number of t-coverings, denoted by x_v, for some $t = t(v)$. As v is loose, either $h(v) + d(v, z_\ell) < k-1$ or $h(v) + d(v, z_r) < k-1$ and assume the former holds, the other case being analogous. By Claim 2.7, for any colouring of D for which all the vertices on $P(v, z_r)$ (which is uniquely determined) are blue, we need to multiply by x the number of ways to extend this colouring of D. In other words, if we enumerate the k possible colourings of D in the natural way, we arrive at the following problem. We are given at most n cyclic intervals I_j of residues modulo k, where each interval I_j, ends with 0 (if $d(v, z_\ell) < k-1$) or starts with $d(z_\ell, z_r) \bmod k$ (if $d(v, z_r) < k-1$), and values x_j for each of these intervals. For each residue $y \in [0, k-1)$, we need to calculate the product

$$\prod_{j \mid\, y \in I_j} x_j.$$

This can be done in time $\mathcal{O}(n)$, without calculating any modular inverses, by considering separately the intervals starting with 0 and those ending with $d(u_\ell, u_r) \bmod k$ and calculating prefix products in linear time.

4 Subtasks

In this last section, we describe possible partial solutions to this problem. Observing some of the facts mentioned in Section 2 can lead to various solutions. It seems difficult to anticipate all possible variations and here we describe only some of the possibilites. All of the described solutions can be implemented with or without counting the number of valid colourings which yields 60% of points for the subtask.

Note that whenever a t-covering is required in the above solution, one can replace it with a 0-covering, that is, only colour the root of the subtree. It is easier to show this is valid (if there exists a valid colouring at all) and it is simpler to check as it is only a single colouring. This way a solution worth 60% of points can be obtained (for any subtask).

4.1 $\mathcal{O}(N^2K)$ – 30 points

We represent the colouring of D by K Boolean variables $x_0, x_1, \ldots, x_{K-1}$, where vertex d_i is marked if and only if the value of $x_{i \bmod K}$ is true. We label each vertex d_i with $i \mod K$ and extend this labelling to all pinned vertices as follows. Let v be a pinned vertex. If there exists a vertex u on D whose distance from v is divisible by K, then give v the label of u. Otherwise, label v with -1 (meaning that it cannot possibly be marked). We know that exactly one of $x_0, x_1, \ldots, x_{K-1}$ is true, and that x_i can be true if and only if i occurs exactly once on every path of length $K-1$ consisting only of pinned vertices/vertices on D. Finally, for each $i \in \{0, 1, \ldots, K-1\}$ with this property (call such an i *good*), we calculate the corresponding number of colourings as previously described. This approach has complexity $\mathcal{O}(N^2K)$ because there are $\mathcal{O}(N^2)$ paths of length $K-1$ and naively updating the set of good indices takes $\mathcal{O}(K)$ time.

4.2 $\mathcal{O}(N^2)$ – 20 points

The approach in this case is basically the same as in the previous subtask, except that we more efficiently maintain the set of good indices while traversing the paths of length $K-1$. We traverse the tree starting at every possible vertex and keep a list of indices that do not occur exactly once on the current path but are not yet marked as bad. When we reach the end of a path of length $K-1$, we mark all vertices in the list as bad and clear the list. The list can be maintained by keeping for each index the number of its occurrences on the current path and its index in the list (or -1 if it is not an element of the list). In this way, the operations of adding elements to the list and removing elements from the list can be implemented to run in constant time.

4.3 $\mathcal{O}(N \log N)$ – 20 points

Consider one of the K possible colourings of D. Additionally, colour red any vertex which is at distance divisible by k from any red vertex on D. Finally, check whether this colouring is valid. Observe that this way each vertex is coloured red at most twice across all K colourings of D. Suppose we fixed a colouring of D, extended it as described, yielding M red vertices in total. Using the auxiliary tree formed by these vertices (the tree containing the M vertices and all pairwise lowest common ancestors), which has size $O(M)$, it is possible to check the vailidity of the given colouring in time $\mathcal{O}(M \log M)$. Hence, this algorithm can be implemented to run in time $\mathcal{O}(N \log N)$. However, the implementation of this approach is very difficult.