MITIT 2024 Spring Invitational Editorials: Final Round

The MITIT Organizers

December 6, 2024

1 Distance Mod 5

Problem Idea: Anton Trygub
Problem Preparation: Benson Lin
Analysis By: Anton Trygub

Let's try to understand which edges could be present in the graph.

1. First condition is trivial: for an edge (u, v) to be present in a graph, we need $B_{u,v} = 1$;
2. Second condition is a bit less trivial: for any edge (u, v), and for any node w, we must have $|A_{u,w} - A_{v,w}| \leq 1$. So, the value $|B_{u,w} - B_{v,w}|$ has to be in $\{4, 0, 1\}$ for all w.

It turns out that these conditions are sufficient to determine all edges (assuming there is a solution). Indeed, assume that these conditions hold for some u, v. Assume that $A_{u,v} = k \geq 6$ with $k = 1 \bmod 5$. Then consider the shortest path from u to v in this graph: $x_0 = u, x_1, \ldots, x_k = v$. Then $A_{u,x_2} = 2$ and $A_{x_2,v} = k - 2$, so $B_{u,x_2} = 2, B_{x_2,v} = 4$, violating the second assumption.

So, the algorithm is simple:

1. Check all $O(n^2)$ pairs of vertices in $O(n)$ time; add those which satisfy the conditions to the graph G_1;
2. Find all pairs shortest distances in G_1; check if they match values $B_{u,v}$. If they do, output G_1, else there is no solution.

Note: this idea works for any modulo ≥ 5 (but doesn't for 3 or 4).

2 Two Tokens

Problem Idea: Sam Zhang
Problem Preparation: Arul Kolla
Analysis By: Sam Zhang

Let V and E be the vertex and edge sets of the graph. The key idea is the following:

Claim 2.1. *Let $S \subseteq V$ such that $1, 2 \in S$. Then S is exactly the set of vertices that can be painted, if and only if both of the following holds:*

(a) The subgraph induced by S is artistic; and

(b) For all $v \in V \setminus S$, either v is connected to all vertices in S, or v is connected to none of them.

Proof. **Necessity of (a):** No token can leave the subgraph induced by S if S is the set of vertices reachable by a token. Then the requirement that all vertices in S can be painted corresponds precisely to the induced subgraph being artistic.

Necessity of (b): Consider some $v \in V \setminus S$. If v is adjacent to exactly one of vertices 1 and 2, then we can move a token to v on the first move, contradicting $v \notin S$, so assume v is adjacent to either none of or both of 1 and 2. Suppose for contradiction that (b) is false; then there exists $u \in S$ such that exactly one of vertex u and vertex 1 is adjacent to v. Since $u \in S$ we can consider a sequence of moves that eventually move a token to u. As we go through the sequence, at some point, the number of tokens that are adjacent to v must change. This number starts at either 0 or 2 and changes by at most 1 on each move, so after some number of moves, exactly one token is adjacent to v. This lets us move a token to v, contradiction. (Thanks to Richard Qi for finding this proof idea!)

Sufficiency of (a) and (b): Since the induced subgraph of S is artistic, we can paint all vertices in S. However, we cannot paint anything outside of S because moving a token outside S will necessarily cause it to become adjacent to the other token. □

Let $f(N)$ be the number of artistic graphs on N vertices. We will compute $f(N)$ using $f(2), \ldots, f(N-1)$. We start with $2^{N(N-1)/2}$ graphs on N vertices, and, for each $2 \leq i < N$, we subtract out graphs with $i = |S|$ reachable vertices, which can be counted using the claim. The formula is

$$f(N) = 2^{N(N-1)/2} - \sum_{i=2}^{N-1} \binom{N-2}{i-2} f(i) 2^{N-i} 2^{(N-i)(N-i-1)/2},$$

since there are $\binom{N-2}{i-2} f(i)$ ways to choose S and the artistic induced subgraph, 2^{N-i} ways to choose whether each $v \in V \setminus S$ is adjacent to S, and $2^{(N-i)(N-i-1)/2}$ ways to connect vertices in $V \setminus S$.

3 Cycling Competition

Problem Idea: Danny Mittal
Problem Preparation: Danny Mittal
Analysis By: Danny Mittal

Let's first try to understand the properties of a cycling competition better. A given cyclist will be eliminated once they are adjacent to cyclists of higher skills on both the left and right.

So let our cyclist be x, let l be the closest cyclist to the left of x with a higher skill, and let r be the closest cyclist to the right of x with a higher skill. x will be eliminated once they are adjacent to l and r – i.e. once all the cyclists between l and x have been eliminated and the cyclists between x and r have been eliminated. (l, r can't have been eliminated earlier because until that happens, they are each adjacent to either x themselves or one of the cyclists in between, which have even lower skills than x).

Therefore, the elimination time of x is 1 more than the maximum elimination time of any cyclist between l, r other than x itself, i.e. 1 more than the maximum elimination time of any cyclist with less skill than x such that there is no cyclist with higher skill than x in between.

This leads to the following observation: if we have two cyclists x, y such that no cyclist in between x and y has higher skill than both of them, then the one of x, y who is eliminated later has higher skill.

This observation is enough to allow us to solve the problem. The problem itself is to determine the skills of the cyclists, i.e. to sort them by skill. This suggests using a standard (efficient) sorting algorithm. Furthermore, because we want to sort in an exact number of steps, we want to avoid randomized sorts such as quicksort – the most obvious choice remaining is then mergesort, which is very easy to analyze.

Mergesort can be described as follows: we have divided the entire input list of length n into "runs", i.e. groups of elements which we have already sorted. Initially, each element is its own run. In each step, we pair up all of the current runs, and merge each pair of runs into a single run by determining the relative order of their elements, which for normal sorting can be done in time linear in the length of the runs. The number of runs is then divided by 2, so we use $\lceil \log_2 n \rceil$ steps.

The only actual work done by mergesort is the merging, so we want to be able to merge, i.e. determine the relative order of elements in two runs using cycling competitions. Let these runs be r, s, so that r, s are lists of cyclists sorted from lowest skill to highest skill. Now let r' be r in reverse order, and imagine doing a cycling competition on $t = r' + s$.

Specifically, consider some cyclist $x \in r$ and some cyclist $y \in s$. In t, in between x and y,

we first have all of the cyclists in r with lower skills than x, and then the cyclists in s with lower skills than y. This means that there are no cyclists in between x, y with higher skills than both, and so the elimination times will tell us which of x, y has higher skill.

Therefore, running a single cycling competition on t gives us complete information about the relative skills of cyclists in t. Note that the cycling competition does not have to only contain t – the observation applies as long as t appears in the cyclist list. This means that using only one cycling competition, we can merge all of our pairs of runs, completing one step of mergesort.

We can therefore sort the cyclists by skill using $\lceil \log_2 n \rceil$ cycling competitions. This is not quite the optimal number of competitions – we can slightly optimize by noticing that we already know the cyclist with most skill from the first query, and so don't need to involve it in the remaining sorting. Thus, if we make an arbitrary first query, extract the cyclist with most skill, then organize the remaining cyclists into runs of length 2, we are essentially applying the same algorithm to $n-1$ cyclists, meaning that we use $\lceil \log_2(n-1) \rceil$ cycling competitions.

Specifically, when n is of the form $2^k + 1$, we save a cycling competition.

Proof of optimality An adversary can act as follows to force $\lceil \log_2(n-1) \rceil$ cycling competitions.

On the first cycling competition, pick an arbitrary cyclist w to be the one with the highest skill, then divide the remaining cyclists into those with high skill and those with low skill. Specifically, the cyclist after w will have low skill, the one after that will have high skill, the one after that will have low skill, and so on, alternating. The actual skills can be assigned arbitrarily, ensuring that the low skills are $1, \ldots, \lceil \frac{n-1}{2} \rceil$ and the high skills are $\lceil \frac{n-1}{2} \rceil, \ldots, n-1$.

By doing this, all the cyclists with low skill will be eliminated in the first round, and so the solver will have no information about their relative skills. Specifically, there are $n-1$ cyclists other than w, and half (rounding up) will have low skill, so the solver will have no information about the relative order of those $\lceil \frac{n-1}{2} \rceil$ cyclists.

In future competitions, we ignore the cyclists which already have high skill, so we are dealing only with the cyclists about whom the solver has no information relative to each other. We then apply the same procedure: starting after w, the first cycling we care about gets low skill, then high skill, etc. The solver will now still have no relative information about the lower half of these cyclists.

Thus, in each competition, this set of cyclists about which the solver has no relative information is divided by half (rounding up); the solver only knows the true skills once this set is of size at most 1, so since we start from $n-1$ cyclists (as we count out w from the start), the solver needs at least $\lceil \log_2(n-1) \rceil$ cycling competitions.

4 Cycle Constrained Graph

Problem Idea: Sam Zhang
Problem Preparation: Claire Zhang and Richard Qi
Analysis By: Sam Zhang

Here, a *walk* is a cycle that may repeat vertices or edges.

Given a graph G with NP vertices, we may assume that cyclically permuting vertices $1, 2, \ldots, P$ does not change G; intuitively this should be relatively clear since if cyclic permutations generated different graphs, they will cancel themselves out modulo P after considering P permutations. Similarly we may simultaneously assume that cyclically permuting vertices $P+1, \ldots, 2P$ does not change G, and the same is true for each block of P vertices, up to $(N-1)P+1, \ldots, NP$.

Here is how the above claim can be proven formally. Consider the group of order P^N, isomorphic to $(\mathbb{Z}/P\mathbb{Z})^N$, formed by all possible compositions of cyclic permutations of each block of P vertices. The group acts on the set S of simple graphs on NP vertices with no simple cycle of length P. Each orbit of this action has size that is a power of P (since it must divide P^N, the order of the group, by the orbit-stabilizer theorem). Since we want to compute $|S|$ modulo P, it suffices to count orbits of size 1. But these are precisely the graphs that are fixed by all cyclic permutations of each block of P vertices.

After making this assumption, we cannot have any edges internal to any of the N blocks, since cyclic permutations of the block will cause that edge to induce a cycle of length P. Moreover, for any two blocks, either every pair of vertices from the two blocks are adjacent, or no pairs are. This lets us define a graph G', with N vertices (each representing a block of P vertices in G), such that two vertices in G' are adjacent if the two corresponding blocks in G are. A simple cycle of length P in G becomes a walk of length P in G', which must give rise to a simple odd cycle in G' of length at most P. Conversely, if there is a simple odd cycle in G' of length at most P, then by inserting extra vertices, we can produce a simple cycle of length P in G. We have now reduced the problem to counting graphs G' with no simple odd cycles of length at most P, or equivalently, graphs G' with no walks of length P.

Now let $f(N, P)$ be the number of graphs G' with N vertices and the given P.

Claim 4.1. *If $N \geq P$, then $f(N, P) \equiv f(N-P+1, P) \pmod{P}$.*

Proof. By a similar argument as we used before, we may assume that cyclically permuting the first P vertices of G' does not change it. Then, again there are no edges internal to the first P vertices, and the P vertices must all have the same neighborhood. For the purposes of determining whether a walk of length P exists, these P vertices can all be combined, so we get $N-P+1$ vertices afterwards. □

Using the claim, we may assume $N < P$. The problem now reduces to counting the number of simple bipartite graphs on N vertices, since a walk of length P exists if and only if an odd cycle exists. This is a fairly standard problem: see `https://codeforces.com/blog/entry/77468` for an example of how to solve it.

5 Avoid XOR Zero

Problem Idea: Anton Trygub
Problem Preparation: Anton Trygub
Analysis By: Anton Trygub

Let's analyze when a particular array $A_1, A_2, \ldots, A_N$ is unavoidable.

1. If $A_i = 0$ for some i, we can ignore it. Let's assume that all A_i are nonnegative.
2. Let's denote by $tr(x)$ the number of trailing ones of x (so that x in binary ends with $tr(x)$ ones). Let's consider some bit $b \geq 0$ and some number x. We will analyze how many numbers among (y, z) can have bit b set, over $y + z = x$.
   (a) If $tr(x) \geq b + 1$, then **exactly one** of y, z will have bit b set: x ends in $b + 1$ ones, so these ones will be distributed between y and z.
   (b) If $tr(x) \leq b$, we can always find $y, z \geq$ with $y + z = x$, which **don't contain** bit b. Indeed: consider $y_1 = \frac{(x-(2^{tr(x)}-1))}{2}$, $z_1 = y_1 + (2^{tr(x)} - 1)$. Then y_1, z_1 disagree only in last $tr(x)$ bits and agree on all previous bits. If y_1, z_1 both contain bit b, then numbers $(y_1 + 2^b, z_1 - 2^b)$ both won't contain bit b.
   (c) If $tr(x) \leq b$, and $x \geq 2^{b+1}$, we can always find $y, z \geq$ with $y + z = x$, which **contain** bit b. Indeed: choose y_1, z_1 as above; if both don't contain bit b, and $z_1 \geq y_1$, then numbers $(y_1 + 2^b, z_1 - 2^b)$ are both nonnegative and contain bit b.
3. Let $m = \max_{1 \leq i \leq n} tr(A_i)$. Let's show that if there exists a number $\geq 2^{m+1}$, then the array is avoidable. Indeed: for each A_i we can split it into $B_i + C_i = A_i$ so that B_i, C_i don't contain bit m, and for some A_i we can split it into $B_i + C_i = A_i$, so that B_i, C_i both contain bit m. Then no matter which $X_i \in \{B_i, C_i\}$s we choose, their XOR will always contain bit m.
4. So, all numbers are $\leq 2^{m+1} - 1$. Since $tr(A_i) \leq m \forall i$, all numbers are $\leq 2^{m+1} - 2$. Then the number with $tr(A_i) = m$ has to be equal to $2^m - 1$.
5. Let's denote the i-th bit of x as x_i. Let's denote $parity(x, (b_1, b_2, \ldots, b_t))$ as $x_{b_1} \oplus x_{b_2} \oplus \ldots x_{b_t}$.

   Lemma 5.1. *For some $1 \leq k \leq m - 1$, assume that one of the following conditions holds:*

   (a) $tr(x) \neq k, tr(x) \leq m$

   (b) $tr(x) = k$, but $2^m \leq x < 2^{m+1}$

   Then exist nonnegative $y+z = x$ with $parity(y, (k-1, k, m)) = parity(z, (k-1, k, m))$.

Proof. First, split x into $x = y_1 + z_1$, where $y_1 = \frac{(x-(2^{tr(x)}-1))}{2}, z_1 = y_1 + (2^{tr(x)} - 1)$. y_1, z_1 differ only in last $tr(x)$ bits. So, if $tr(x) \neq k, tr(x) \leq m$, y_1 and z_1 won't differ in bit m, and will differ either in both or in none of bits $k-1, k$. This proves a).

If $tr(x) = k$ and $2^m \leq x < 2^{m+1}$, then $parity(x, (k-1, k, m)) = 0$, so we can split x into $(x, 0)$, proving b). □

6. Assume that there exists $1 \leq k < m$, such that there is no A_i with $a_i < 2^m$ and $tr(A_i) = k$. Let's show that the array is avoidable.

   For every $A_i \neq 2^m - 1$, we can split it into $A_i = B_i + C_i$ so that $parity(B_i, (k-1, k, m)) = parity(C_i, (k-1, k, m))$. So, no matter what x_i we choose, for their xor X the value $parity(X, (k-1, k, m))$ is fixed. Now, just split $2^m - 1$ so that this final $parity(X, (k-1, k, m))$ is odd: into $(2^m - 1, 0)$ if we don't want to flip parity of X, and into $(2^m - 1 - 2^k, 2^k)$ if we do.

7. Now assume that for all $1 \leq k < m$ there **exists** A_i with $tr(A_i) = k$ and $A_i \leq 2^m - 1$. Then the array is unavoidable.

   If $tr(A_i) = k$, then for any nonnegative B_i, C_i with $B_i + C_i = A_i$, $B_i \oplus C_i$ ends with $0\underbrace{11\ldots1}_{k \text{ ones}}$.

   Lemma 5.2. *Assume m numbers $x_1, x_2, \ldots, x_m \leq 2^m - 1$, are such that x_i binary ends with $0\underbrace{11\ldots1}_{i \text{ ones}}$. These m numbers form a XOR basis (so we can represent any number $\leq 2^m - 1$ as XOR of some of them).*

   Proof. Note that $x_m = 2^m - 1, x_{m-1} = 2^{m-1} - 1$, so we can get $m-1$-st bit as their combination.

   If 2^t is in their combination for each $t = k, k+1, \ldots, m-1$ for some k, then consider $x_{k+1} \oplus x^k$: its smallest nonzero bit is $k-1$, so 2^{k-1} is also represented as XOR of som x_is. So, we can show that it's true for all k by induction. □

   So, no matter what B_i, C_i are chosen:

   - If $A_i \geq 2^m$, choose $X_i < 2^m$
   - Otherwise, start with $X_i = B_i$. Note that we have an option to XOR the final XOR by $B_i \oplus C_i$.
   - By our lemma, we can XOR this XOR by any number $\leq 2^m - 1$, and therefore can make it 0.

Now, we can formulate the final condition for an array to be unavoidable:

Claim 5.3. *Array $(A_1, A_2, \ldots, A_N)$ is unavoidable if and only if all its elements are 0, or there exists some $m \geq 1$, such that:*

- $0 \leq A_i \leq 2^{m+1} - 2$ *for all* i
- *For every $1 \leq k \leq m$, there exists A_i with $A_i < 2^m$ and $tr(A_i) = k$*

With this in mind, counting the number of unavoidable arrays is easy. Fix m, then all numbers have to be $\leq 2^{m+1} - 1$, and for each $1 \leq k \leq m$, we have to select at least one number with $A_i < 2^m$ and $tr(A_i) = k$. We can denote the number of such numbers by t_k, and do a standard dynamic programming $dp[len][k]$: how many arrays of length len are there, where all numbers are $< 2^m$, and for each $1 \leq k_1 \leq k$ there exists a number x with $tr(x) = k_1$. Transition is just: $dp[len][k] = \sum_{0 \leq len_1 < len} dp[len_1][k-1] \cdot \binom{len}{len_1} \cdot t_k^{len - len_1}$.

For a fixed m, we can calculate this in $O(n^2 k)$, which gives us the final runtime of $O(n^2 k^2)$.