Problem Tutorial: "Dress to Impress"

Let's build a bipartite graph where one part corresponds to types of clothes and the other one — to colours. The edge in the graph is a piece of clothes. One of the upper bounds for the answer is the minimum degree of the vertices in the first part. From each attractive set we leave only k pieces of different colours. These pieces correspond to some matching of size k. We will try to choose the maximum number of such matchings. Minimum of this number and minimum degree in first part is the answer.

We want to check if we can make x matchings of size k. Suppose we found such matchings. The union of all these matchings forms the graph with degree not exceeding x. Let's find subgraph with xk edges with such property. To do this, we build a network. We add edges from source to the first part of our graph with capacity x, orient edges of the graph from first part to second and add edges from second part to target with capacity x. If maximum flow in this network is less than xk, then the answer is less than x. It remains to prove the opposite.

Let's leave in the graph only edges with non-zero flow. First of all, we want to split this graph on any x matchings. In each iteration we want to choose some matching which covers all vertices of maximum degree. It is easy (using Hall's theorem) to prove that we can choose a matching which covers all vertices of maximum degree in one part. If we unite such matchings for two parts, we will have graph consisting of paths and cycles. Cycles and paths of odd length are not a problem — we can choose a matching from such subgraphs which covers all vertices. Also we can notice that every even path has at least one end which is not a vertex of maximum degree. So in these paths we can choose necessary matching too. Maximum degree of the graph doesn't exceed x, so we split graph in x (possibly empty) matchings.

Finally, we need to change these matchings. Let's take two matchings: one with less than k edges and another with more than k edges. Let a and b are sizes of these matchings. Again, consider the union of these matchings. It has at least $b-a$ odd paths. So we can rearrange edges between these matching such that difference in their sizes will be any number not exceeding $b-a$. In particular, we can rearrange edges such that one of them has exactly k edges. Using such operations we can construct x matchings of size k and we are done.

Determining maximum x can be done in $O(c^2)$ with Ford-Fulkerson algorithm, because necessary flow isn't exceed c. The trivial upper bound on x is c/k and running time of finding one matching is $O(kc)$. So finding partition into matchings works in $O(c^2)$ too. Each exchange work in $O(n)$ and we will have at most x exchanges. Hence the last part works in $O(nc)$ and overall complexity is $O(c^2)$.

Problem Tutorial: "Somewhere Over the Rainbow"

Let's do something similar to building convex hull. In each step we will have stack of points (x_i, y_i) with value d_i, where (x_i, y_i) is a subset of initial constrains. We want the equation $a_{x_i} = y_i$ and inequality $a_{x_i+1} - a_{x_i} \le d_i$ to be hold.

Having triple (x, y, d) we want to determine if it is possible to satisfy next condition $a_{x_i} \ge y_i$. Maximum value of a_{x_i} is $y_{max} = y + (x_i - x)d - \frac{(x-x_i)(x-x_i-1)}{2}$. So we need to check that $y_{max} \ge y_i$. If this condition doesn't hold, then we need to erase last triple from stack. Otherwise we want to minimize sum on interval $(x, x_i]$.

To get value y_{max} we assumed that a takes maximum possible values under constraints producing by triple (x, y, d). If we decrease a_j by 1, where $j \in (x, x_i]$, then value in a_{y_i} decreases by $y_1 - j + 1$ and sum decreases by $\frac{(y_1-j+1)(y_1-j+2)}{2}$. So we need to find maximum value of the expression $\sum \frac{b_k(b_k+1)}{2}$, where $1 \le b_k \le x_i - x$ and $\sum b_k = y_{res} - y$. The function $\frac{b_k(b_k+1)}{2}$ is convex and increases on $[1, +\infty)$, so we need to maximize the number of terms b_k which is equal to $x_i - x$ and possibly there will be one more term, which is not equal to $x_i - x$. Note that with such construction we also minimize the number of terms. Hence we maximize new $d_i = d - (x_i - x) - (\text{number of } b_k)$ and we can guarantee that any function, that satisfies first i constrains and has point (x_i, y_i), has value $a_{x_i} - a_{x_i-1}$ less than our function. So this construction provides minimum constraint on d_i for function. Finally, we need to push triple (x_i, y_i, d_i) in stack.

Lifehack. If you are having troubles working with the strict second constraint and wish to make it not strict (that is, replace $>$ by $\geq$), you can achieve that by subtracting some h_i from each a_i so that $2h_i = h_{i-1} + h_{i+1} + 1$, solve the problem with new constraints ($a'_0 = -h_0$, $a'_m = -h_m$, $a'_{x_i} \geq y_i - h_i$, $2a'_i \geq a'_{i-1} + a'_{i+1}$) and then add sum of all h_i to the answer. For any fixed k choosing h_i as $k + (k-1) + \ldots + (k-i+1) = (2k-i+1)i/2$ would suffice. One can pick $k = 0$, $k = 10^9$, $k = \lfloor \frac{m}{2} \rfloor$ or anything else.

This trick simplifies calculations (slightly) in the solution above, as well as in every other solution we know.

Problem Tutorial: "Goldberg Machine"

Let us root the tree at the vertex 1, and for each vertex v construct an order N_v of its neughbours so that the parent of v (if there is one) is last in N_v, and N_v is a cyclic shift of e_v. Let us also construct the Euler tour $T = (t_1, \ldots, t_{2(n-1)})$ of edge transitions with respect to the chosen children orders.

Let a *phase* be a sequence of steps between two adjacent traverses of $t_{2(n-1)}$. Observe that:

- on any phase, each transition t_i is made at most once;
- on any phase, transitions are made in order of T;
- if a vertex v was first visited on phase k, then a transition (v, u) is made on phase $k+1$ if (v, u) precedes the current edge of v in N_v, otherwise it is made on phase k;
- all phases starting with n-th will traverse the whole tree.

Let k_i be the number of the first phase the transition t_i is made. One can see that:

- if for each vertex v the first transition of N_v is active, all $k_i = 0$;
- changing an active edge of any vertex v changes k_i on a segment by ± 1;
- the total number of transitions made on the first K phases is $\sum_{i=1}^{2(n-1)} \max(0, K - k_i)$.

We can now process all queries with a standard sqrt-decomposition RSQ structure and binary searches in $O(\sqrt{n \log n})$ per query.

Problem Tutorial: "Grid Game"

Let us make a few observations.

WLOG assume $(x, y) = (0, 0)$. Suppose that we now have to choose a move for a black chip. If $x_i \neq y_i$, the following proposition implies that it is always better to move towards the main diagonal $x = y$.

Observation 1. Let p be any position where white moves first, and there is a winning strategy for black. Let i be a black chip with $x_i - 1 \geq y_i + 1$. Then the position p' with (x_i, y_i) replaced with $(x_i - 1, y_i + 1)$ still has a winning strategy for black.

Proof. The chip i will make $\lceil (x_i + y_i)/2 \rceil > y_i$ moves until it can meet the white chip. Consider movement of the chip i in the winning strategy of p. In any branch, if the chip i always moves down, y_i becomes negative and it is irrelevant. Otherwise, replace the first `L` move with a `D`. We've obtained a winning strategy for p'.

Thus, the only ambiguity is how to move the black chips that lie on the main diagonal (observe that all such chips have odd $x_i + y_i$ at the start). Informally, this only matters if such a chip has to catch the white chip if it chooses to run in one direction.

Observation 2. Observation 1 still holds if $x_i - y_i = 1$, unless in all winning strategies in p the chip i counters (captures on) the white strategy `R`* (always go right).

Proof. This condition means there is no sequence of all `D` moves, hence replacement is again possible.

We now consider an intermediate position of a white cell (x, y), and all possible arising positions of black chips where they can win. For a starting position, let $P(x, y)$ be the set of all positions such that:

- the white chip is at (x, y) and moves first;
- all black chips made $x + y$ moves arbitrarily (without capturing the white chip);
- there is a winning strategy for black.

Can we somehow combine a pair of winning position in $P(x+1, y)$ and $P(x, y+1)$? Yes, unless it means that a chip with odd coordinate sum can be "tricked" into countering both strategies `R`* (run to the right) and `U`* (run to the left). Note that an even-sum chips can adapt to the white chip's movement, hence this is not issue.

Proposition 1. Let $p_R \in P(x+1, y)$, $p_U \in P(x, y+1)$ such that:

- chip i counters `R`* in a winning strategy of p_R or can meet the white chip at $(x+1, y)$;
- chip j counters `U`* in a winning strategy of p_U or can meet the white chip at $(x, y+1)$;
- either $i \neq j$, or $i = j$ is a chip with $x_i + y_i \equiv x + y \pmod 2$.

Then there is a position $p \in P(x, y)$ such that i counters `R`* and j counters `U`*.

Proof. Apply Observation 1 (and its symmetric counterpart) to p_R and p_U while possible. After this for every black chip k, either it always moves in one direction (hence e.g. we can not change $x_i \to x_i - 1$), or it is on or next to the main diagonal of (x, y). Further, for every black chip k it is possible to place it in p_R and p_U at most one unit apart. Indeed, this is not the case only when k is just below the diagonal of $(x+1, y)$ in p_R, and on top of the diagonal of $(x, y+1)$. This means that $x_i + y_i$ is odd, and either $k \neq i$ or $k \neq j$, hence Observation 2 is applicable in one of the cases. Now we can construct p by placing each black chip k so that both its positions in p_R and p_U are immediately reachable.

Let $R(x, y)$ and $U(x, y)$ be the sets of black chips countering `R`* and `U`* respectively in any $p \in P(x, y)$. Denote $R'(x, y) = R(x, y) \cup \{k \mid \text{the black chip } k \text{ can meet the white chip at } (x, y)\}$, and $U'(x, y)$ defined similarly.

The computation rule confirms the above intuition: if we can not cover one of the running strategies, we have a losing position (thus both sets empty), or borrow the countering sets from adjacent positions, except if an odd-sum chip has to counter the running strategy on one side, it must **now** move there and cannot counter anything on the other side. In particular, if i is an odd-sum cell which is the only one that can cover any of the running strategies, white can outplay it on this very move and black loses.

Rule. $R(x, y)$ and $U(x, y)$ can be computed as follows:

- If any of $R'(x+1, y)$ or $U'(x, y+1)$ is empty, then $R(x, y) = U(x, y) = \varnothing$.
- If $R'(x+1, y) = \{k\}$ with $x_k + y_k$ odd, then $U(x, y) = U'(x, y+1) \backslash \{k\}$, otherwise $U(x, y) = U'(x, y+1)$.
- If $U'(x, y+1) = \{k\}$ with $x_k + y_k$ odd, then $R(x, y) = R'(x+1, y) \backslash \{k\}$, otherwise $R(x, y) = R'(x+1, y)$.

Proof. If, say, $R'(x+1, y) = \varnothing$, then it is impossible to counter `R`*, hence there are no winning strategies. Any element of $R(x, y)$ has to be in $R'(x+1, y)$ by definition (the strategy `R`* begins with moving to $(x+1, y)$). Using Proposition 1 we can include any i in, say, $R(x, y)$ unless $x_i + y_i$ is odd and it is the only element of $U'(x, y)$ respectively, in which case it must be above the diagonal of (x, y) before the white move to counter U*, thus it can not ever counter `R`*.

Note that $R(x, y) = U(x, y) = \varnothing$ if $x > C$ or $y > C$. The above rule allows us to compute sets $R(x, y)$ and $U(x, y)$ for all positions (x, y) with $x, y \le C$, say, by decreasing (x, y) lexicographically. White can win only if $R(x, y) = U(x, y) = \varnothing$.

Problem Tutorial: "k-coloring"

Let's solve the problem separately for different k.

If $k = 1$ we just have to find an Eulerian path starting from the first vertex. We need to output -1 if we didn't succeed.

If $k = 2$ we can always construct the necessary path. Run dfs (depth-first search), and write down vertices when we come to them from their ancestor, or descendant, ignoring the reverse edges (edges that lead to visited vertices, from which we have not yet come out). It is clear that the constructed path will paint all edges. Arbitrary edge will be painted either when we go through it for the first time, or when we go back through it.

If $k = 3$ we use the same traversal as in $k = 2$. Let's call this path $P_1, \ldots, P_m$. Let's follow this path, but sometimes go back to the previous vertex. Let's assume that we have already processed vertices $P_1, \ldots, P_{4i}$, and stand in the vertex P_{4i}, then we perform the movement pattern $++-+++$ (here "+" is the movement to the next vertex in the path P and "−" is to the previous one). Thus we will find ourselves in the vertex $P_{4(i+1)}$, from where we will repeat our actions and so on (if there is no next vertex in some moment because the next vertex doesn't exist then we finish generating a new path). The resulting path will be the answer. We paint 2nd and 4th edge from the processed 4 ones, so we painted every 2nd edge from the way, and we already know that this will give us painted all the edges (from the case $k = 2$).

If $k > 3$ we reduce the problem to smaller k, because we can always go over an arbitrary edge and then return back to decrease k for the current phase by 2 (except the case with $n = 1$).

Problem Tutorial: "Just Shuffle the Input"

For the sake of simplicity let's say that the strings are zero indexed, as is the permutation.

Map each symbol to a number, random or not, and denote this mapping by $f(c)$. Also pick a big prime number and a random residue r. Consider the following product of two polynomials (of x):

$$\left(f\left(t_0\right) + f\left(t_{p(0)}\right) x + f\left(t_{p^2(0)}\right) x^2 + \ldots + f\left(t_{p^{m-1}(0)}\right) x^{m-1}\right) \cdot \left(1 + r^{p^{m-1}(0)} x + r^{p^{m-2}(0)} x^2 + \ldots + r^{p(0)} x^{m-1}\right)$$

(all powers of r are already taken modulo our big prime). It's easy to see that for any $d < m$ the sum of coefficients of this product at x^d and x^{m+d} equals

$$f(t_0) r^{p^{m-d}(0)} + f(t_{p(0)}) r^{p^{m-d+1}(0)} + \ldots = \sum_{i=0}^{m} r^i f\left(t_{p^d(i)}\right).$$

One can see that this is exactly the hash of the d-th permutation of t (well, actually it will be true if we replace p by its inverse, but this can be verified by the sample tests). So after one fast polynomial multiplication (using fast fourier transform) we get all the hashes of t. If we precalculate hashes of all substrings of s of size m, we can easily find the earliest hash of t's permutations which occurs there.

One modulo is not enough because of the birthdays paradox. Two primes should be sufficient.

Problem Tutorial: "Restoring a Permutation"

The first observation to be made is that if $a_i = a_j$ and $i < j$ then $p_i > p_j$. So the numbers in positions with equal a_i decrease. We can also notice that number on position i must be bigger than some number on position j where $j \in [1, i)$ and a_j equal to $a_i - 1$. The numbers on positions j with $a_j = a_i - 1$ decrease, so we can claim that p_i is bigger than number on maximum position j, where $j \in [1, i)$ and $a_j = a_i - 1$.

Let's build directed graph on n vertices where edges (i, j) means that $p_i < p_j$. We will add only some edges into this graph and then will prove that it is enough. Let $i_1 < \ldots < i_m$ is positions with equal a_i. We add edge (i_{k+1}, i_k) for all k. Also for all positions i we add edge to maximum position $j \in [1, i)$ such that $a_j = a_i - 1$. Previously we shown that for such edges (i, j) the inequality $p_i < p_j$ holds. Similar edges can be drawn for array b.

Obviously, obtained graph is acyclic and has linear number of edges. Consider any topological ordering of this graph and make permutation according this permutation. We want to prove that it is an answer. Suppose that the longest increasing subsequence ending at position i is not equal to a_i and i is the smallest such index. It can't be bigger because all j with $a_j = a_i$ and $j < i$ must stand after i in the ordering. It can't be smaller because there is j with $a_j = a_i - 1$ that must stand before i in the ordering. Similarly for array b. Hence, constructed permutation satisfies all constrains.

Problem Tutorial: "Shadow Companion"

A very useful observation which can help a lot is that in this problem writing a simulator and BFSing programs for small subproblems is almost surely better in terms of human time than trying to solve this on paper.

Another one observation is that you cannot avoid using shadow, because $5_2 = 101$ and $25_2 = 11001$ have different parities, and flipping without shadow doesn't change the parity. You *maybe* can avoid using the exchange operation (we don't know, but we are sure you can), but this operation may shorten your program.

We are sure that there is a plenty of solutions. We describe one of them. For the sake of clarity we will denote some number fragments by binary strings, replacing some bits by the question mark wildcard (`?`) and by letters to show some relations between them and taking bits where we stand in brackets (`[]`).

Our solution uses the following subprograms:

- Flip a single bit and get rid of the shadow if the bit to the left is zero: `?0[x] -> ?0[y]` where $y = \overline{x}$.
- Move a bit two steps to the left if there was `0` before: `0?[x] -> [x]?0`.
- Move a bit one step to the left if there was `0` before: `?0[x] -> ?[x]0`.
- Move a bit one step to the right if there was `0` before: `?[x]0 -> ?0[x]`.
- Put zero to the bit to the left if we are standing at zero: `??[0] -> ?0[0]`.
- Get a conjunction of the two bits, zeroing them: `0x[y] -> [z]00` where $z = x \wedge y$.
- If we are at 1 then replace it by 0 and go two steps left, otherwise do nothing. Requires the adjacent bits to be set to zeroes: `x0[1]0 -> [x]000` and `x0[0]0 -> x0[0]0`.

Let's enumerate all positions from 0 right to left. Let $L = 10$. We divide the positions in the following way:

- The 0-th bit will be zero during almost the whole execution.
- The bits from 1 to $4L$ are divided into L blocks of size 4. i-th of them will look like `000x` during almost the whole execution, where x is the i-th rightmost bit of n.
- Bits from $4L + 1$ to $4L + 5$ are our playfield where we will perform some calculations.
- Bits from $4L + 6$ to $12L + 5$ are divided into $2L$ blocks of size 4. i-th of them will look like `000x` almost all the time, where x is the i-th rightmost bit of the answer, and the algorithm we propose is essentially just updating the answer for the whole time.

- Bits $12L + 6$ and $12L + 7$ are set to ones during almost the whole execution. We call them *the stopping ones*. In fact, we would be good with only one of them, but flipping both is simpler.

So what we are going to use is that if $c_0, \ldots, c_{L-1}$ are the bits of n then

$$n^2 = \left(\sum_{i=0}^{L-1} c_i 2^i\right)^2 = \sum_{i=0}^{L-1} c_i 2^{2i} + \sum_{i<j} (c_i \wedge c_j) 2^{i+j+1}.$$

Our algorithm is as follows:

- For each i from $L-1$ to 0 move the i-th bit to the place $4i+1$, thus $e\ x\ p\ a\ n\ d\ i\ n\ g$ the number.
- Go left till we are at $12L + 6$ and flip, thus creating the stopping ones.
- For each i from 0 to $L-1$ copy c_i one step to the left, then move this copy to our playfield, jumping over the other input bits. Once it is at $4L + 4$, we add 2^{2i} to the answer if it is 1 and don't add otherwise (how to add is explained below).
- For each i from 0 to $L-1$ and for each $j < i$ copy c_i and move this copy to $4L+3$ as in the previous step, then go to the c_j bit and copy it to $4L+2$, then put their conjunction into $4L+4$ and, again, conditionally add some power of two.
- Flip the stopping ones back to zeroes, we don't need them anymore.
- Zero all the input bits (this is why we need the empty 0-th position).
- Move all answer bits from 0 to $2L - 1$ to the places they need to be.

So what remains is how to solve the following subproblem: *we are at the* $4L+4$*, and here is also some bit* c*. We have some number* d *and we want to add* 2^d *to the answer iff* $c = 0$ *and end at the empty* $4L + 4$ *in any case.* An algorithm follows:

- Move this bit c to the $4L + 8 + 4d$ which is a position exactly between the d-th and the $(d+1)$-st answer bits (we call this position the *center* of the corresponding block).
- For each i from $d+1$ to $2L$ flip the single center bit of the i-th block. Then return to the d-th center.
- If it is 1 then move two steps to the right, otherwise don't (for example, `RlRr`).
- $4L$ times call our last oracle (that "if we are at 1 then replace it by 0 and go two steps left, otherwise do nothing" one). After this we actually update all the answer bits properly, and some centers will be set to ones.
- Go about $8L$ times left to ensure that we are beyond the stopping ones. Then do `Lr` $8L$ times, after which we are guaranteed to stand on the left stopping one.
- Go back to the playfield, zeroing each center.

This finishes the algorithm. None of our implementations takes more than 160 000 instructions.

Bonus: here are some of the oracles we found which you may find useful for your algorithms:

- conjunction: `flRLflRsfRfRLfLrflfl`
- xor: `lsLrflRfLrrLfLrflfl`
- disjunction: `lRLflRsfRfRLfLrsxfRlfl`

- move1left: `rflfsRfxflRrflxfRl`
- copy1left: `RLflRfRfRLfsLfrLrflfLrfLflRLrfRlL`
- move1right: `frsRxflRxfRLfxfRf`
- copy1right: `frfsLfrLRfR`
- move2left: `lRLflRfrfLfLrflfl`

Problem Tutorial: "The Older We Are, The Worse It Hurts"

Let $sz(u, v)$ be the size of the subtree of v if u would be its parent, $s(u, v)$ be the sum of all a_i in that subtree, and $f(u, v)$ be the answer to the problem where we traverse the subtree starting with v.

Consider some vertex u. Assume that once we went from u to v, traversed its subtree, then returned to u, immediately went to w, traversed its subtree and returned to u. Let's find out when it's optimal to swap these traversals. Assume that when we entered v it was the $(k + 1)$-st vertex (that is, we got $(k + 1) \cdot a_v$ penalty for it). Then we want to compare the following pair of values (left hand side is what we get now and right hand side is the penalty if we visit w before v):

$$k \cdot s(u, v) + f(u, v) + (k + sz(u, v)) \cdot s(u, w) + f(u, w) \quad \text{vs.} \quad k \cdot s(u, w) + f(u, w) + (k + sz(u, w)) \cdot s(u, v) + f(u, v).$$

Subtract $k(s(u, v) + s(u, w)) + f(u, v) + f(u, w)$ from both sides:

$$sz(u, v) \cdot s(u, w) \quad \text{vs.} \quad sz(u, w) \cdot s(u, v).$$

Since both sizes are positive, we are to compare $\frac{s(u,w)}{sz(u,w)}$ and $\frac{s(u,v)}{sz(u,v)}$. As we want the penalty to be the minimal possible, we should visit vertex v before w if $\frac{s(u,v)}{sz(u,v)} > \frac{s(u,w)}{sz(u,w)}$. Now the solution is quite straightforward: we calculate all $sz[u][i]$ and $s[u][i]$ (here $s[u][i] = s(u, v)$ where v is the i-th neighbor of u in its adjacency list), then sort each vertex' neighbors by $\frac{s(u,v)}{sz(u,v)}$ in reverse order (of course, it's better to perform all comparisons in integers), then fix some root (say, 0), calculate all $f[u][i] = f(u, v)$ where v is not u's parent, then recalculate the answer for parents.

To calculate some $f(u, v)$ we take all other neighbors $(w_1, w_2, \ldots, w_c)$ of v in the required order and calculate the following:

$$f(u, v) = a_v + \sum_{i=1}^{c} f(v, w_i) + 1 \cdot s(v, w_1) + (1 + sz(v, w_1)) \cdot s(v, w_2) + (1 + sz(v, w_1) + sz(v, w_2)) \cdot s(v, w_3) + \ldots$$

When we calculate it for u being the v's parent, we can do it explicitly. When we are at some v and calculating this $f(u, v)$ for all its sons u we in fact can say that w_i are all neighbors of v, and for each its son we exclude its additor from its position, calculate the corresponding $f[u][v]$ and return it back. This exclusion and returning can be maintained using Fenwick tree.

After we know all $f(u, v)$ it's easy to obtain the answer to the problem just iterating over all possible roots and calculating the sum above for all its neighbors.

Problem Tutorial: "Three Vectors"

Let v_1, v_2 and v_3 be the input vectors. Replace each variable x_i by $\overline{x_i}$ if he majority of vectors has the i-th bit equal to 1 and flip this bit. Now for each bit there is at most one vector where this bit is set to one.

One can see that if these three vectors satisfy any 2-CNF formula then the zero vector also satisfies it. Indeed, consider any clause $\pm x_i \vee \pm x_j$. In at least two of our three vectors $x_i = 0$, and in at least one of them $x_j = 0$ as well. Since this vector satisfies the formula, it satisfies the clause, hence the zero vector also satisfies it. That is, the zero vector satisfies every clause we consider.

It turns out that there is a formula where nothing else satisfies the formula. Indeed, for each i where $v_{1i} = v_{2i} = v_{3i} = 0$ we add a clause $\overline{x_i} \vee \overline{x_i}$. Next, if $(i_1, i_2, \ldots, i_c)$ are all the bits from v_1 which are set to 1 then we add clauses $(\overline{x_{i_1}} \vee x_{i_2}) \wedge (x_{i_1} \vee \overline{x_{i_2}})$, $(\overline{x_{i_2}} \vee x_{i_3}) \wedge (x_{i_2} \vee \overline{x_{i_3}})$ and so on (in total $2(c-1)$ clauses), thus forcing these bits to be equal. Do the same for nonzero bits from v_2 and v_3. Then we have two cases.

If each of the vectors is nonzero then take one nonzero bit from each of them, let these indices be j_1, j_2 and j_3. We must force that no two of them can be set to 1 at the same time. It can be done by adding clauses $(\overline{x_{j_1}} \vee \overline{x_{j_2}}) \wedge (\overline{x_{j_2}} \vee \overline{x_{j_3}}) \wedge (\overline{x_{j_3}} \vee \overline{x_{j_1}})$.

If there is the zero vector among these three (say, v_3), then we add just one clause $(\overline{x_{j_1}} \vee \overline{x_{j_2}})$.

One can see that in any case we used no more than $2n$ clauses.