

## Problem A. Alphabet City

*Problem author and developer: Mikhail Ivanov*

Let us enumerate all English letters with integers  $[0..25]$ . Prepare a two-dimensional array  $\text{dp}[1..n][0..25]$  where  $\text{dp}[i][j]$  is equal to how many occurrences of letter  $j$  there are in  $s_i$ . Also compute an array  $\text{sum}[0..25]$  where  $\text{sum}[j] = \sum_{i=1}^n \text{dp}[i][j]$ . Then the task was to find, for each  $\ell \in [1..n]$ , the maximum number  $k \geq 0$  such that, for every  $j \in [0..25]$ ,

$$\sum_{\substack{i \in [1..n] \\ i \neq \ell}} m \text{dp}[i][j] \geq \text{dp}[\ell][j] + \sum_{\substack{i \in [1..n] \\ i \neq \ell}} k \text{dp}[i][j].$$

Using our array  $\text{sum}$ , we can simplify this to  $m \text{sum}[j] - m \text{dp}[\ell][j] \geq k \text{sum}[j] - (k-1) \text{dp}[\ell][j]$ . Let  $r = m - k$ , then we have:  $m \text{sum}[j] - m \text{dp}[\ell][j] \geq (m - r) \text{sum}[j] - (m - r - 1) \text{dp}[\ell][j]$ . After a bit of reordering, we get:  $r(\text{sum}[j] - \text{dp}[\ell][j]) \geq \text{dp}[\ell][j]$ . To minimize  $r$  (and thus maximize  $k$ ), let us consider cases:

- if both  $\text{dp}[\ell][j] = 0$  and  $\text{sum}[j] = 0$ , then this holds for any  $r$ ;
- if  $\text{dp}[\ell][j] = \text{sum}[j]$  but  $\text{dp}[\ell][j] > 0$ , the inequality  $r(\text{sum}[j] - \text{dp}[\ell][j]) \geq \text{dp}[\ell][j]$  never holds, and we can print the answer  $-1$  right away;
- otherwise,  $r \geq \left\lceil \frac{\text{dp}[\ell][j]}{\text{sum}[j] - \text{dp}[\ell][j]} \right\rceil = \left\lfloor \frac{\text{sum}[j] - 1}{\text{sum}[j] - \text{dp}[\ell][j]} \right\rfloor$ .

So, to solve the problem, find  $r$  as the maximum value over  $\ell \in [0..25]$  of  $\left\lfloor \frac{\text{sum}[j] - 1}{\text{sum}[j] - \text{dp}[\ell][j]} \right\rfloor$  (skipping all values of  $\ell$  where  $\text{dp}[\ell][j] = \text{sum}[j] = 0$ , but terminating on such values of  $\ell$  where  $\text{dp}[\ell][j] = \text{sum}[j] > 0$ ) and print  $\max\{m - r, -1\}$ . The time and memory complexity of this solution equals  $\mathcal{O}(|\Sigma|n + L)$  where  $|\Sigma| = 26$  is the size of the alphabet,  $n$  is the size of Alphabet City, and  $L$  is the total length of all strings in the input.

## Problem B. Battle of Arrays

*Problem author: Mikhail Ivanov; problem developer: Dmitry Petrov*

### Optimal strategy

The optimal strategy is for each player to choose the maximum element from their own array and use it to attack the maximum element from the opponent's array.

### Proof of strategy optimality

**Definition 1.** We say that an array  $a$  is **no better than** an array  $b$  (and that array  $b$  is **no worse than**  $a$ ), if after sorting both in non-increasing order the following holds:

$$\text{len}(a) \leq \text{len}(b), \quad \text{and} \quad \forall i : a_i \leq b_i$$

We say that the elements  $a_i$  and  $b_i$  correspond to each other. This is written as:  $a \leq b, b \geq a$ .

**Definition 2.** An array  $b$  is said to **majorize** an array  $a$  ( $a \preceq b$  or  $b \succeq a$ ) if, after sorting both arrays in non-increasing order, the following inequalities hold:

$$\forall k \in \{1, \dots, n\} : \sum_{i=1}^k a_i \leq \sum_{i=1}^k b_i$$

where  $n = \max(\text{len}(a), \text{len}(b))$ , and we define  $a_i = 0$  for  $i \geq \text{len}(a)$ , and  $b_i = 0$  for  $i \geq \text{len}(b)$ . We say that the elements  $a_i$  and  $b_i$  correspond to each other. This is written as:  $a \preceq b$  or  $b \succcurlyeq a$ .

**Definition 3.** We will call a pair of arrays  $(a, b)$  winning if, when the first player has the array  $a$  and the second player has the array  $b$ , the first player will defeat the second under optimal play.

**Theorem 1.**

If a pair of arrays  $(a, b)$  is winning, and  $b' \leq b$ , then  $(a, b')$  is also winning.

Can be proven by induction.

**Theorem 2.**

If a pair of arrays  $(a, b)$  is winning, and  $a' \succcurlyeq a$ , then  $(a', b)$  is also winning.

Can be proven by induction.

**Theorem 3.**

If a pair of arrays  $(a, b)$  is a winning one, and we replace  $\max(b)$  with  $\max(\max(b) - \max(a), 0)$ , then the resulting array  $b'$  is such that the pair  $(b', a)$  is a losing one.

**Proof:**

Indeed, since the pair  $(a, b)$  is a winning one, there exists a winning move for the first player.

Let us denote this winning move as  $(a_i, b_j)$  — that is, attacking element  $b_j$  with element  $a_i$ .

According to Theorem 1, if we replace the attacking element with  $\max(a)$ , the resulting move  $(\max(a), b_j)$  will also be a winning move.

According to Theorem 2, if the pair  $(a, b)$  is **not** winning, then any pair  $(a', b)$ , where  $a' \preceq a$ , is also **not** winning.

Therefore, if we replace the element being attacked with  $\max(b)$ , the situation for the second player does not change (if they were not winning before, they still will not be winning). Thus, we can replace  $b_j$  with  $\max(b)$  without worsening the situation.

Therefore, we can make the move  $(\max(a), \max(b))$ , which is also a winning move, which was to be proved.

## Solution

Let's simulate the optimal strategy. Put all elements of the arrays into heaps (priority queues) for maximum values. Then, while at least one of the heaps is not empty, alternately extract the maximum element from Alice's heap and use it to attack the maximum element in Bob's heap, then vice versa, and so on. Continue this process until a winner is determined.

## Time complexity

Let  $N = \text{len}(a) + \text{len}(b)$  denote the total size of the arrays,  
and  $M = \max(\max(a), \max(b))$  denote the max element of the arrays.

Each move takes  $O(\log N)$  time.

In each pair of moves, at least the maximum is subtracted from the sum of all  $N$  numbers. Therefore, the total sum is multiplied by at most  $1 - \frac{1}{N}$  in each step.

Initially, the sum is at most  $NM$ , so after at most

$$\frac{\log(NM)}{\log\left(1 - \frac{1}{N}\right)} = O(N(\log N + \log M))$$

iterations, someone is guaranteed to lose.

Thus, all queries can be processed in

$$O(N \log N (\log N + \log M))$$

total time.

## Problem C. Cacti Classification

*Problem author: Ivan Safonov; problem developer: Egor Kulikov*

### Core cactus fact

If you delete two distinct edges from the same simple cycle of a cactus, the graph becomes disconnected.

### Brackets

Process edges in increasing label order. Divide edges into 16 brackets. For  $i = 0, \dots, 14$ , bracket  $i$  contains edges that:

- are not bridges, and
- have exactly  $i$  earlier edges on their (unique) cycle.

Bracket 15 contains all remaining edges (bridges and edges that are  $\geq 16$ -th on their cycle).

For each  $i = 0, \dots, 14$  maintain a base set (graph)

$$G_i = \mathcal{E} \setminus \{\text{edges already placed into bracket } i\}.$$

Because bracket  $i$  removes at most one edge per cycle and no bridges, every  $G_i$  stays connected.

### Pass 1: find the bracket of every edge (4 queries per edge)

For the current edge  $e$ , consider the predicate

$$P(i) : \text{ASK}(G_i \setminus \{e\}) = 1,$$

where  $\text{ASK}(S) = 1$  iff the graph with edge set  $S$  is connected, and  $\text{ASK}(S) = 0$  otherwise.

Then  $P(i)$  is monotone in  $i$ :

- if  $e$  is a bridge, then  $P(i) = 0$  for all  $i$ ;
- if  $e$  is on a cycle and is the  $(b+1)$ -st edge of that cycle by label with  $b \leq 14$ , then  $P(i) = 0$  for  $i < b$  and  $P(i) = 1$  for  $i \geq b$ .

So we binary search the smallest  $i \in [0, 14]$  with  $P(i) = 1$ . If none exists, put  $e$  into bracket 15; otherwise into that bracket  $i$  and update  $G_i \leftarrow G_i \setminus \{e\}$ . This takes exactly 4 queries per edge.

## Pass 2: determine bridge / cycle length

### Brackets 14 and 15 (1 query)

If  $e$  is in bracket 14 or 15, ask once:

$$\text{ASK}(\mathcal{E} \setminus \{e\}).$$

If the answer is 0, then  $e$  is a bridge. If the answer is 1, then  $e$  lies on a big cycle (length  $\geq 15$ ).

### Brackets 0 to 13 (up to 4 queries)

Let  $b$  be the bracket of  $e$  (so  $b \leq 13$ ). For  $i = 0, \dots, 14$ , query  $\text{ASK}(G_i \setminus \{e\})$ , but treat  $i = b$  as bad regardless (since  $e \notin G_b$  by construction). Define

$$\text{Bad}(i) = \begin{cases} 1, & i = b, \\ 1, & \text{ASK}(G_i \setminus \{e\}) = 0, \\ 0, & \text{ASK}(G_i \setminus \{e\}) = 1. \end{cases}$$

Then  $\text{Bad}(i)$  is monotone: it equals 1 exactly for  $i < \min(L, 15)$ , where  $L$  is the cycle length of  $e$ . Binary search the largest  $i \in [0, 14]$  with  $\text{Bad}(i) = 1$ ; the result is

$$\min(L, 15) = i + 1.$$

If  $i + 1 \leq 14$ , output that length; otherwise output big cycle.

## Query bound

Pass 1 uses  $4m$  queries. Pass 2 uses at most  $4m$  queries. Total  $\leq 8m$ .

## Problem D. Doorway

*Problem author and developer: Tikhon Evtееv*

It's easy to see that we will only need a single opening, since if there are more than one opening, we could move all the doors between to neighboring openings to one side, making it so that there is one less opening, and the total size does not decrease.

Now, let's move all the doors to the left as far as possible, and for the  $j$ -th door on the  $i$ -th lane let its rightmost point in that position be  $R_{i,j} = x_{i,1} + \sum_{k=1}^j l_{i,k}$ , and  $R_{i,0} = x_{i,1}$ .

Similarly, let's move all the doors to the right as far as possible, and for the  $j$ -th door on the  $i$ -th lane, let its leftmost point in that position be  $L_{i,j} = x_{i,2} - \sum_{k=j}^{k_i} l_{i,k}$ , and  $L_{i,k_i+1} = x_{i,2}$ .

With these definitions, the leftmost point of the largest opening will be one of the points  $R_{i,j}$  and the rightmost point will be one of the points  $L_{i,j}$ , since whenever we have an opening from some  $l$  to some

$r$ , we can move all the doors whose rightmost point is  $\leq l$  as far as possible to the left, and all the other doors as far as possible to the right, without decreasing the size of the opening.

Let's now move all doors to the right as far as possible and go through them in increasing order of  $R_{i,j}$ , moving them one by one as far to the left as possible. Once we have gone through all the doors with  $R_{i,j} \leq l$  for some value  $l$ , we will find the largest opening with the left border at  $l$ , because we can't move any other doors to the left of  $l$ , and it's best to keep all the other doors as far to the right as possible.

To efficiently find the answer during this process we can maintain a multiset of all the values  $L_{i,j}$  for the doors that are still to the right, and remove values from there as we go, so that at any point we know the right boundary of the current opening — the smallest element in the multiset. For the left boundary we can maintain the maximum of  $R_{i,j}$  that are already to the left, and just update the answer on each iteration.

The final time complexity is  $O((n + \sum_{i=1}^n k_i) \cdot \log(n + \sum_{i=1}^n k_i))$

## Problem E. Elevator Against Humanity

*Idea: Maria Zhogova; problem author: Tikhon Evteev; problem developer: Zakhar Iakovlev*

It is a constructive problem with several tricky cases. It can be hard to pass without stress solution.

Let  $x_1, x_2, \dots, x_{2n}$  be the order of floors visited by the elevator either for boarding ( $x_i = s_k$ ) or for disembarkation ( $x_i = f_k$ ). Then, the total time is

$$t = |x_1 - 1| + |x_2 - x_1| + \dots + |x_{2n} - x_{2n-1}|.$$

The solution is focused on how to open the absolute value brackets. We can change the floor order, and, thus, the sign in front of  $x_i$  in the sum for  $t$ . Each  $x_i$  (except the last one) appears twice in the sum, and the sign can be '+' or '-' in front of each summand. However, there are two main restrictions:

- There are a total of  $2n$  plus signs and  $2n - 1$  minus signs (we neglect the starting first floor in the first brackets).
- For each  $i$ ,  $s_i$  should go before  $f_i$ . That implies, for example, that  $x_1$  is the boarding floor and  $x_{2n}$  is the disembarkation floor.

Let's, first, forget about the restriction for the floors order. Consider the array  $y_1 < y_2 < \dots < y_{2n}$ , which is the sorted array  $x$ . The task simplifies to the following: we have  $2n$  distinct integers, each of which is written twice except one ( $x_{2n}$ ), which we choose freely from  $f_i$ . We need to put  $2n$  plus signs and  $2n - 1$  minus signs in between to maximize the total sum

$$t = \max_{\pm: \#(+)=2n, \#(-)=2n-1; k: y_k \in f} (-1 \pm y_1 \pm y_1 \pm \dots \pm y_{k-1} \pm y_{k-1} \pm y_k \pm y_{k+1} \pm y_{k+1} \pm \dots \pm y_{2n} \pm y_{2n}).$$

Qualitatively, we want to place all '+' to the bigger numbers, all '-' to the smaller numbers. Let's virtually separate all  $y$ -s into two halves: big numbers with ids  $> n$  and small numbers with ids  $\leq n$ . We call them the high group and the low group, respectively. There are two distinct scenarios: either all  $s$  are in the low group and all  $f$  are in the high group or at least one  $s$  has an index  $> n$  in sorted order. We start from the second case.

We can iterate over all  $k$ , such that  $x_{2n} = y_k$  and take the optimal answer. There are again two cases:

1. If  $k \leq n$  (in the low group), the sum looks as follows:

$$t = -1 - 2y_1 - \dots - 2y_{k-1} - y_k - 2y_{k+1} - \dots - 2y_n + 2y_{n+1} + \dots + 2y_{2n}.$$

2. If  $k > n$  (in the high group), the sum looks as follows:

$$t = -1 - 2y_1 - \dots - 2y_{n-1} - y_n + y_n + 2y_{n+1} + \dots + 2y_{k-1} + y_k + 2y_{k+1} + \dots + 2y_{2n}.$$

It can be shown that such sums are achievable, reordering  $y_i$  to get array  $x$ , that gives the desired sum. We prove it for the first case, the second case can be proved similarly.

We make the proof constructively. The elevator alternates between the floors from high and low groups and finishes at the floor  $y_k$ . The elevator starts to the (existing in this case) boarding floor in the high group. Then, the elevator can recursively move to the unvisited floor in the other group. At each turn, at least half of the remaining unvisited floors are in the other group. Suppose that the elevator can't visit them all. This means that they all are disembarkation floors for the people who haven't been taken yet. However, this means that all these people are waiting in the current group. Therefore, the elevator can change the last move and board the person that is going to the other group. The last disembarked person should go to the floor  $y_k$  and the order is constructed.

Finally, we are left with the case in which all  $s_i$  are below all  $f_j$ . If the indices of the topmost  $s$  and bottom  $f$  are different, the elevator just starts at the topmost  $s$  and finishes at the bottom  $f$  with the sum

$$t = -1 - 2y_1 - \dots - 2y_{n-1} - y_n + y_n + y_{n+1} + 2y_{n+2} + \dots + 2y_{2n}.$$

If the topmost  $s$  and the bottom  $f$  have the same indices, there are three values achievable, depending on the order of the visiting, and the maximum of them should be taken

- The elevator starts from the  $y_{n-1}$  and finishes at  $y_{n+1}$

$$t = -1 - 2y_1 - \dots - 2y_{n-2} - y_{n-1} + y_{n-1} - 2y_n + y_{n+1} + 2y_{n+2} + \dots + 2y_{2n}.$$

- The elevator starts from  $y_n$  and finishes at  $y_{n+2}$

$$t = -1 - 2y_1 - \dots - 2y_{n-1} - y_n + y_n + 2y_{n+1} + y_{n+2} + 2y_{n+3} + \dots + 2y_{2n}.$$

- The elevator starts from  $y_n$  and goes down, and finishes in  $y_{n+1}$ , moving down from the higher floor

$$t = -1 - 2y_1 - \dots - 2y_{n-1} + 2y_n - y_{n+1} + 2y_{n+2} + \dots + 2y_{2n}.$$

All the proofs for the corner cases are similar to the given one.

## Problem F. Fragmented Nim

*Problem author: Ivan Safonov; problem developer: Gennady Korotkevich*

Let's call a game state *winning* if the player whose turn it is can win by force, and *losing* otherwise.

Let's show that if Bob chooses pile  $i$  with more than 1 stone for Alice, then Bob loses. Consider the game state  $S$  without this pile. If  $S$  is a losing state, Alice can remove all stones from pile  $i$ , putting Bob into  $S$ . Otherwise, Alice can remove all stones *but one*, and in the next turn, Alice can choose the same pile  $i$  for Bob, which will force Bob to put Alice into the winning state  $S$ .

Thus, the players have to choose a pile with 1 stone for their opponent while they can. After that, the current player to move will win, unless there are no more piles (if every pile initially consisted of 1 stone), in which case they will lose.

Here is a correct solution in Python:

```
for t in range(int(input())):
    n = int(input())
    a = list(map(int, input().split()))
    moves = a.count(1) + (1 if max(a) > 1 else 0)
    print("Alice" if moves % 2 == 1 else "Bob")
```

## Problem G. Greta's Game

*Problem author and developer: Denis Mustafin*

Let  $b_i$  be the number of rounds where the  $i$ -th participant's number was strictly larger than the  $(i \bmod n) + 1$ -st participant's number. Then  $a_i = b_i + b_{i-1}$  where  $b_0 = b_n$ .

Let's determine the minimum possible number of rounds  $k$  if all  $b_i$  were known. In each round, some  $b_i$  increase by 1. They can't all increase by 1, since then each participant's number would be strictly larger than the next one's and they stand in a circle. Any other set of  $b_i$  can increase by 1. In every round  $\max_i b_i$  increases by at most 1, and  $\sum_{i=1}^n b_i$  increases by at most  $n - 1$ , so  $k \geq \max(\max_i b_i, \frac{\sum_{i=1}^n b_i}{n - 1})$ . That lower bound can be achieved. Let  $c_i$  be initially 0, and in each round we will increase at most  $n - 1$  values  $c_i$ , so that they become equal to  $b_i$  after  $k$  rounds. In every round we will increase  $\min(n - 1, \#\{i | b_i > c_i\})$  values  $c_i$  that have the largest  $b_i - c_i$ . Then if at some point there are less than  $n - 1$  indices with  $b_i > c_i$ , then either there is some  $c_i$  which we have increased in every round (then we will keep increasing it in every round, so the number of rounds will be equal to  $b_i$  for that  $i$ ), or every value was not increased at least once (then the difference between any two  $c_i$  is at most 1, so there is at most one round, where we increase  $< n - 1$  values, so the number of rounds is  $\left\lceil \frac{\sum_{i=1}^n b_i}{n - 1} \right\rceil$ ).

Note that  $\sum_{i=1}^n a_i = 2 \sum_{i=1}^n b_i$ , so we can always determine  $\sum_{i=1}^n b_i$ .

If  $n$  is odd, then  $b_1 = \sum_{i=1}^n b_i - a_3 - a_5 - \dots - a_n$ , so we can determine  $b_1$ , and then determine all  $b_i$ , since  $b_i = a_i - b_{i-1}$ . So in that case we already know that the answer is  $\max(\max_i b_i, \left\lceil \frac{\sum_{i=1}^n b_i}{n - 1} \right\rceil)$ .

If  $n$  is even, then let  $\hat{b}_1 = 0$ , and then let  $\hat{b}_i = a_i - \hat{b}_{i-1}$  for  $2 \leq i \leq n$ . Note that if the  $a_i$  data is correct, then  $\hat{b}_1 + \hat{b}_n = a_1$ . Then for any  $\Delta$ ,  $b_i = \hat{b}_i + (-1)^i \cdot \Delta$  will satisfy all  $a_i = b_i + b_{i-1}$  conditions, and any  $b_i$  values that satisfy those conditions can be obtained this way for  $\Delta = -b_1$ . So now we need to find  $\Delta$ , such that all  $b_i \geq 0$  and  $\max_i b_i$  is as small as possible. Let  $l_0$  and  $r_0$  be the smallest and largest of  $\hat{b}_i$  for even  $i$ , and  $l_1$  and  $r_1$  be the smallest and largest of  $\hat{b}_i$  for odd  $i$ . Then we need to minimize  $\max(r_0 + \Delta, r_1 - \Delta)$  for integer  $\Delta$  satisfying  $-l_0 \leq \Delta \leq l_1$ . If at least one of  $\left\lfloor \frac{r_1 - r_0}{2} \right\rfloor$  and  $\left\lceil \frac{r_1 - r_0}{2} \right\rceil$  lies inside  $[-l_0, l_1]$ , then the optimal  $\Delta$  is that value. Otherwise, it's whichever one of  $-l_0, l_1$  is closer to  $\frac{r_1 - r_0}{2}$ . So the answer is

the maximum of  $\left\lceil \frac{\sum_{i=1}^n \hat{b}_i}{n - 1} \right\rceil$  and  $\max(r_0 + \Delta, r_1 - \Delta)$  for the optimal  $\Delta$ .

## Problem H. Honey Cake

*Problem author and developer: Georgiy Korneev*

First, let's compute the number of pieces  $w_p, h_p, d_p$  along each axis. We must ensure that  $n = w_p \cdot h_p \cdot d_p$  and we know that  $w_p | w, h_p | h, d_p | d$ , where  $x | y$  notation means "x divides y". If we multiply all these divisibility constraints we get  $w_p \cdot h_p \cdot d_p | w \cdot h \cdot d$ , so  $n | w \cdot h \cdot d$  — that is the condition for solution to exist.

To find a factorization of  $n$  into  $n = w_p \cdot h_p \cdot d_p$  we can use the greedy approach, taking the largest possible factors as  $w_p, h_p$ , and  $d_p$  using greatest common divisor (gcd). In more detail:

- Let  $w_p = \gcd(n, w)$  and let  $n_w = n/w_p$ .
- Let  $h_p = \gcd(n_w, h)$  and let  $n_h = n_w/h_p$ .
- Let  $d_p = \gcd(n_h, d)$  and let  $n_d = n_h/d_p$ .

The answer exists if the remaining  $n_d = 1$  and the answer is  $w_c = w_p - 1, h_c = h_p - 1, d_c = d_p - 1$ .

## Problem I. Irrigation Interlock

*Problem author: Maksim Turevskii; problem developer: Niyaz Nigmatullin*

### Build Convex Hull and Classify Points

Let  $A$  denote one of the sets and  $B$  denote the other one. Build  $\text{conv}(B)$  (Convex Hull of  $B$ ) in  $O(m \log m)$  time. For each point in  $A$ , test whether it lies inside or outside  $\text{conv}(B)$  in  $O(\log m)$  time per point, for a total of  $O(n \log m)$  time. We can do this for both choices of  $A$  and  $B$ .

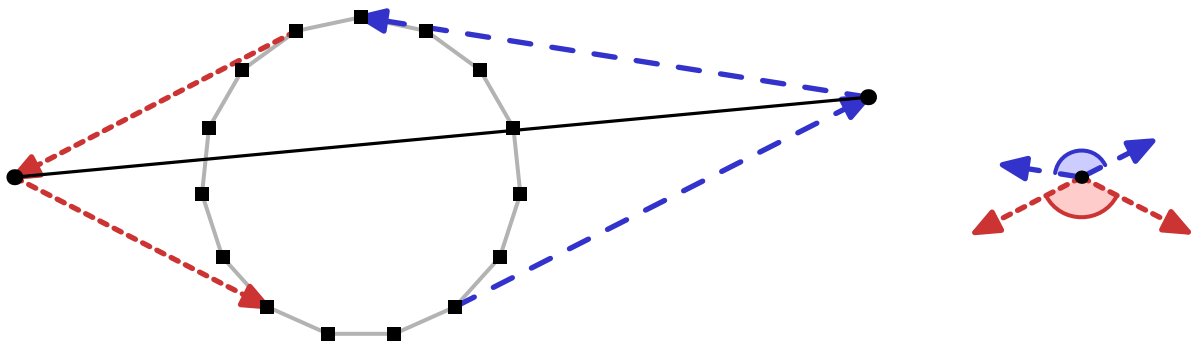
We now have two cases:

- **Case 1:** For one of the sets (say  $A$ ), there exists a point inside  $\text{conv}(B)$  and another point outside  $\text{conv}(B)$ . The segment connecting these two points must cross the boundary of  $\text{conv}(B)$ , and we can find the intersection with one of the hull edges.
- **Case 2:** For both sets, all points are either entirely inside or entirely outside the other set's convex hull. Note that both sets cannot have all their points inside the other's convex hull simultaneously (since the sets are disjoint and non-empty). Therefore, for at least one set, all points lie outside the convex hull of the other set.

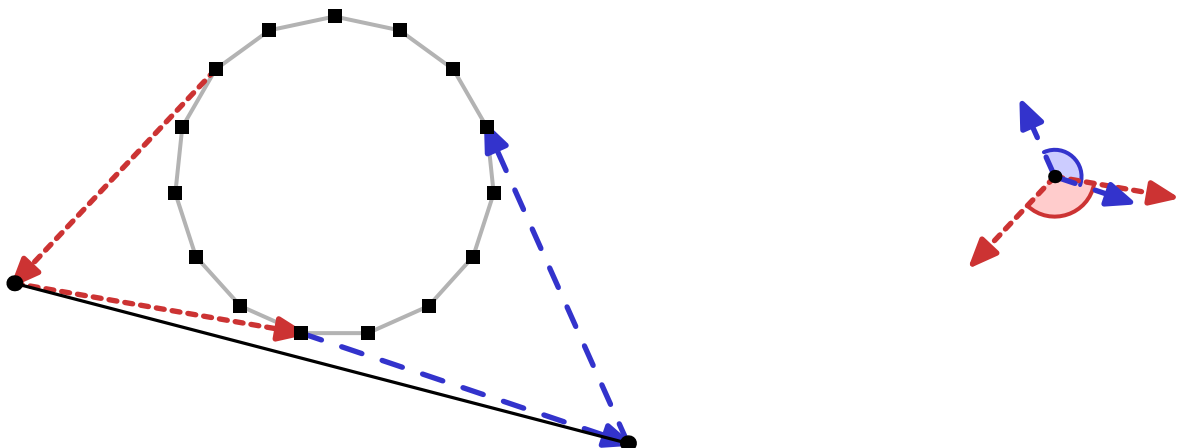
Without loss of generality, swap the sets if necessary so that all points of  $A$  lie outside  $\text{conv}(B)$  (when in Case 2), or arrange so that  $A$  has both inside and outside points (when in Case 1).

### All Points Outside the Hull

If all points from set  $A$  lie outside  $\text{conv}(B)$ , the problem becomes more interesting. For a segment  $pq$  from set  $A$  to intersect a segment from set  $B$ ,  $p$  and  $q$  must not “see each other” because of  $\text{conv}(B)$ , and in that case  $pq$  will intersect some side of  $\text{conv}(B)$ . On building two tangents from each of  $p$  and  $q$ , directing the left tangent towards the point, and the right tangent towards the polygon, and these two tangents form some arc on the unit circle. The “see each other” is equivalent to the case when these two arcs don't intersect.

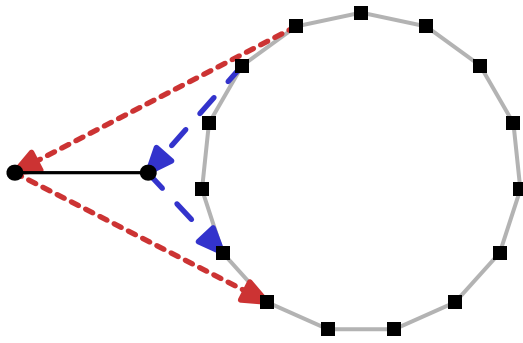


Intersection exists, the arcs don't intersect  
 The right picture illustrates all the tangent directions and their arcs





No intersection, the arcs intersect



No intersection, the arcs intersect

To find such  $p$  and  $q$ :

1. Generate all directed tangents as vectors
2. Sort all of them by angle
3. Enumerate each vector with distinct integer — its index in the sorted array
4. Find two non-intersecting ranges on a circle: can be done by generating start and end events of ranges, and sweep-line.

### Time Complexity

- Building convex hull:  $O(m \log m)$
- Testing  $n$  points for containment:  $O(n \log m)$
- Finding tangents for  $n$  points:  $O(n \log m)$
- Sweep line with  $O(n)$  events:  $O(n \log n)$  for sorting

Total time complexity:  $O((n + m) \log(n + m))$

## Problem J. Jinx or Jackpot

*Problem author and developer: Maksim Turevskii*

Let's think that we are picking the sequence of  $k$  outcomes before, that is we are picking the  $p_i$  uniformly, and then sequence (win,loss,win,win,loss,...) (win is jackpot, loss is jinx) with probability of  $p_i^{wins} \cdot (1-p_i)^{losses}$ . Then we will have a fixed prefix of length  $a+b$  with  $a$  wins and  $b$  losses with probability

$$\text{of } \frac{\sum_i p_i^a (1-p_i)^b}{n}.$$

So we can think as like we have the automaton that produces the next outcome jackpot with probability

$$p(a, b) = \frac{\sum_i p_i^{a+1} (1-p_i)^b}{\sum_i p_i^a (1-p_i)^b} \text{ where } a \text{ is the number of wins so far, and } b \text{ is the number of losses so far.}$$

Then we can think about  $dp(a, b, x)$  as the maximum expected reward if we currently have  $a$  wins so far,  $b$  losses so far and  $x$  money. Then  $dp(a, b, x) = \max_{0 \leq y \leq x, y \in \mathbb{Z}} (p(a, b)dp(a+1, b, x+y) + (1-p(a, b))dp(a, b+1, x-y))$  if  $a+b < k$  and  $x$  if  $a+b = k$ .

Then it is easy to see by decreasing induction on  $a+b$  that  $dp(a, b, x)$  is  $const \cdot x$  and we should either bet  $x$  or 0. Then we can write  $C[a][b]$  meaning  $dp(a, b, x) = C[a][b] \cdot x$  and recalculate  $C[a][b] = \max(p(a, b)C[a+1][b] + (1-p(a, b))C[a][b+1], 2p(a, b)C[a+1][b])$  if  $a+b < k$  and 1 if  $a+b = k$ . Then the final answer is  $1000(C[0][0] - 1)$ . We can compute all  $p(a, b)$  in  $O(n + 100 \cdot k^2)$  and then compute  $C[\square][\square]$  in  $O(k^2)$ , so the final asymptotic would be  $O(n + 100 \cdot k^2)$ .

The relative error will be small enough.

## Problem K. Knit the Grid

*Problem author and developer: Denis Mustafin*

In each cell let's put 0 if that cell lies inside an even number of cycles and 1 otherwise. Then we can think that for the cells outside the grid, the numbers in those cells are 0s. Now a grid line between adjacent grid points is a part of some cycle if the numbers in cells on either side of that line are different. This means that a frog in a cell is green if the sum of numbers in its four adjacent cells is even, and brown if its odd. The fact that cycles don't intersect is equivalent to no  $2 \times 2$  square being filled with numbers in a chessboard pattern (i.e. any 2 adjacent cells have different numbers). The fact that there is a cycle passing through each inner grid point is equivalent to no  $2 \times 2$  square being filled with the same number. Now we will solve the problem in terms of numbers in cells, and in the end, restore cycles using those numbers.

The numbers can be viewed as boolean variables. Note that using frog colors we can derive all the variables from just the top row: we can go from top to bottom, filling rows one-by-one. Then for each cell in the last filled row, we know the sum of numbers in four adjacent cells modulo 2, as well as 3 of those numbers. So we can determine the fourth number, which is in the next row. If we fill the grid in that way, then all conditions given by frog colors will be satisfied, except for the frogs in the bottom row.

The conditions on  $2 \times 2$  squares mean that for at least one diagonal of the square, the numbers on that diagonal must be different. So we need to find out how the xor of two numbers in diagonally adjacent cells is expressed through the numbers in the top row. Let's find that for every pair of diagonally adjacent cells, such that at least one of them is inside the grid, going from top to bottom. Let's number the rows 1 through  $r$  from top to bottom, and the columns 1 through  $c$  from left to right. Rows 0 and  $r+1$  will represent rows outside the grid, and columns 0 and  $c+1$  will represent columns outside the grid. Then let  $lft_{i,j}$  be the xor in cells in the  $i$ -th row and the  $j$ -th column and in the  $i-1$ -st row and the  $j-1$ -st column. And let  $rght_{i,j}$  be the xor in cells in the  $i$ -th row and  $j$ -th column and in the  $i-1$ -st row and  $j+1$ -st column. Let the number in the first row and  $j$ -th column be  $x_j$  ( $x_0 = x_{c+1} = 0$ ). Let  $P_{i,j}$  be 0 if the frog in the  $i$ -th row and  $j$ -th column is green, and 1 otherwise.

First, if the bottom cell in a pair is in the first row, then the xor is equal to the number in that cell (since the number in the other cell is 0). That means  $lft_{1,j} = rght_{1,j} = x_j$ . For each pair of diagonally adjacent

cells, there is a cell  $p$ , that is adjacent to both of them and in the same row as the top cell (it's the cell on top of the bottom one in a pair). If  $p$  is outside the grid, then so is the bottom cell, and the xor of that pair is the same as the xor of that pair, with the bottom cell replaced with the cell on top of  $p$ . That means  $lft_{i,1} = rgh_{i-1,0}$  and  $rgh_{i,c} = lft_{i-1,c+1}$ . If  $p$  is inside the grid, then we know the xor of all its neighbors, so the xor in our pair is equal to the xor of all  $p$ 's neighbors and the other pair of  $x$ 's neighbors. That means  $lft_{i,j} = P_{i-1,j} \oplus lft_{i-1,j+1}$  and  $rgh_{i,j} = P_{i-1,j} \oplus rgh_{i-1,j-1}$ .

That means that the xor of any pair of diagonally adjacent cells equals either  $x_j$  or  $\neg x_j$  for some  $j$ . So the conditions for  $2 \times 2$  squares can be expressed as a boolean formula of two variables. For frog colors in the bottom corners, we need to add conditions  $lft_{r,2} = P_{r,1}$  and  $rgh_{r,c-1} = P_{r,c}$ , which can be expressed as a boolean formula of one variable. Now let's add conditions on the frogs in the  $r$ -th row and  $j$ -th column for  $j$  from 2 to  $c-1$ . Since we have already added the condition for the frog in the  $r$ -th row and  $j-2$ -nd column, we can use  $rgh_{r+1,j-2}$  and it will be equal to the number in the cell in the  $r$ -th row and  $j-1$ -st column. So the condition is  $rgh_{r+1,j-2} \oplus lft_{r,j+1} = P_{r,j}$ , which is also a boolean formula of two variables. All conditions on variables can be expressed as boolean formulas of at most two variables, so we can find a solution or determine that there is no solution using 2-SAT algorithm. There are  $\mathcal{O}(c)$  variables and  $\mathcal{O}(rc)$  conditions, so the algorithm runs in  $\mathcal{O}(rc)$  time.

## Problem L. LLM Training

*Problem author and developer: Ivan Safonov*

Let's firstly understand how to solve the problem for a fixed context size  $k$ . We can construct the set of  $s$  pairs  $(ctx_i, next_i)$ , obtained for all tokens generated by LLM for all texts. So, the loss is:

$$\mathcal{L}_k(P_k) = \sum_{i=1}^s -\log_2 P_k(next_i|ctx_i)$$

Now we can solve the problem for each context value  $ctx_i$  independently, because we select conditional distributions for contexts independently. For a fixed context value  $ctx$ , we have some list  $N_{ctx}$  of  $next_i$  tokens, for which  $ctx_i = ctx$ . The optimal distribution for the next token, which minimises  $\mathcal{L}_k(P_k)$  is the distribution, proportional to numbers of occurrences of tokens to  $N_{ctx}$ , so  $P_k(next|ctx) = \frac{\text{count}(next, N_{ctx})}{|N_{ctx}|}$ . The optimal loss can be written as (\*):

$$\begin{aligned} \mathcal{L}_k(P_k) &= \sum_{ctx} \sum_{next \in N_{ctx}} -\log_2 \left( \frac{\text{count}(next, N_{ctx})}{|N_{ctx}|} \right) = \\ &= \sum_{ctx} |N_{ctx}| \log_2 |N_{ctx}| - \sum_{ctx} \sum_{next \in T} \text{count}(next, N_{ctx}) \log_2 \text{count}(next, N_{ctx}) \end{aligned}$$

So, we calculate the number of occurrences of each context and add to the answer the sum of  $x \log_2 x$  of these numbers. And for each pair  $(ctx, next)$  we calculate the number of occurrences and subtract from the answer the sum of  $x \log_2 x$  of these numbers. So, the answer can be calculated in a linear time for a fixed context size  $k$  using hashes. But this gives us only  $\mathcal{O}(n^2)$  solution, so we need to optimize more.

To calculate the answer efficiently, let's separate the answer into two parts: for contexts with length less than  $k$  (they are prefixes only), and contexts with length exactly  $k$ .

For the first part, let's calculate the answer using trie. We can build the trie for all given strings in a linear time. After that, each vertex of this trie with depth  $\ell$  will contribute some value to all answers will  $k \geq \ell + 1$ . This value can be calculated by the same formula (\*).

For the second part, let's construct the string  $rev(t_1)\#_1 rev(t_2)\#_2 \dots \#_{n-1} rev(t_n)$  (concatenation of all reversed texts with different separators). Let's construct the suffix array  $a$  with LCP array for this string. Let's go from  $k = n-1$  to  $k = 0$  and maintain the answer for contexts with size exactly  $k$ . At each moment of time, tokens with equal contexts with length exactly  $k$  will be some subsegments of suffix

array, which we can maintain in DSU. Using LCP array we can maintain these subsegments by merging components of suffixes  $a_i, a_{i+1}$  at the moment  $k = LCP_i$ . For each component we can maintain:

- The number of tokens, written by LLM, from which suffixes in the component start.
- Some map/hashmap of counts of these tokens.

We can merge these data structures for two components, by adding the numbers of tokens and merging maps of counts using small-to-large. While merging, we can easily recalculate the global answer for the current context size  $k$ . Note, that we reversed texts to make them prefixes of suffixes to use suffix array and maintain such components.

Time complexity is  $O(n \log n)$  or  $O(n \log^2 n)$  depending on what type of map is used.

## Problem M. Medical Parity

*Problem author: Elena Kryuchkova; problem developer: Roman Elizarov*

This problem can be solved with dynamic programming. Let  $x'_i$  and  $y'_i$  be the bits of two input strings ( $x'_i, y'_i \in \{0, 1\}; 1 \leq i \leq n$ ). Define  $D_{i,b}$  ( $0 \leq i \leq n; b \in \{0, 1\}$ ) as the minimal number of bit flips that can be done to the prefixes of  $x'$  and  $y'$  of length  $i$  so that  $b = y_i$  — the last parity control bit of this prefix after bit flips. Initialize dynamic programming with  $D_{0,0} = 0$  and  $D_{0,1} = \infty$ , then compute the values of  $D_{i,b}$  forward for all  $i$  from 1 to  $n$  and for  $b \in \{0, 1\}$  in the following way:

$$D_{i,b} = \min_{a=0..1} \left\{ D_{i-1, (b-a) \bmod 2} + \delta_{x'_i, a} + \delta_{y'_i, b} \right\}$$

where  $\delta_{a,b}$  is defines as:

$$\delta_{a,b} = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

Then the answer to the problem is  $\min(D_{n,0}, D_{n,1})$ .