Problem A. Balanced Arrays

Let us add the number m to the start and the end of the array: it won't affect its balancedness.

From now on, we will count the number of balanced arrays that start and end with m.

Now we can note that the array $a_0, a_1, \ldots, a_{n+1}$ is balanced if and only if $\sum |a_i - a_{i+1}| \le 2m$.

Let us draw a polyline from $(0, 0)$ to $(2k, 0)$ for some $k \le m$ in the following way. There will be $n + 1$ segments with tangent 1 or -1, i-th between points (b_i, c_i) and (b_{i+1}, c_{i+1}), where $b_i = |a_0 - a_1| + \ldots + |a_{i-1} - a_i|$ and $c_i = a_0 - a_i$. Note that $c_i \ge 0$ for every possible i.

We have established a bijection between good arrays and polylines with segments. Let us now forget about the ends of segments. Then, for a fixed polyline, the number of ways to define segments depends only on the number of peaks of this polyline (and is equal to some binomial coefficient). But the number of such polylines with a given number of peaks can be calculated by a simple formula using, for example, cyclic shifts (similarly to counting Catalan numbers using Renyi's lemma).

This way, we can obtain a formula with $O(n^2)$ terms, which can be then calculated in $O(n \log n)$ time using FFT.

Problem B. Bins and Balls

Without loss of generality, $x_1 \le x_2 \le \ldots \le x_n$.

Let us start with some trivial bounds. Firstly, the answer is no more than $\frac{\sum_{i=1}^{n} x_i}{k}$, because we throw away exactly k balls each time.

Next, we may note that the answer is no more than $\frac{\sum_{i=1}^{n-1} x_i}{k-1}$, because each time we throw away at least $k-1$ balls that have colors different from n. By the same logic, we may obtain that the answer is no more than the integer part of

$$M = \min_{t<k} \frac{\sum_{i=1}^{n-t} x_i}{k-t}.$$

In fact, the answer is exactly equal to the integer part of M. Indeed, let us perform the actions one by one. To perform the current action, we will consider k most common colors of balls and throw out exactly one ball of each such color. If there is any choice, we prefer colors with smaller indices.

Suppose that we cannot perform any more actions. Denote the numbers of balls left with colors $1, 2, \ldots, n$ by $y_1, y_2, \ldots, y_n$ respectively. Firstly, we may note that $y_1 \le y_2 \le \ldots \le y_n$, because it is true after every operation. Let us find the maximum possible p such that $y_p \le 1$. Then, in every action, we threw out at least one ball of every following color: $p+1, p+2, \ldots, n$.

Therefore, the number of actions is at least $\frac{\sum_{i=1}^{p} x_i - y_i}{k+p-n}$. Now, note that $\sum_{i=1}^{p} y_i$ is less than $k + p - n$ (there are at most $k-1$ colors with $y_i > 0$, and $p+1, p+2, \ldots, n$ make $n-p$ of them), and $\frac{\sum_{i=1}^{p} x_i}{k+p-n} \ge M$. Therefore, $\frac{\sum_{i=1}^{p} x_i - y_i}{k+p-n} > M - 1$. There is only one integer in the range $(M-1, M]$: exactly the integer part of M.

The complexity is $O(n \log n)$, with sorting being the bottleneck.

Problem C. Cards

A quadratic solution is trivial. We can just calculate $dp[x][y]$: the number of ways to obtain the secret number equal to y after x steps.

There are plenty of ways to get a faster solution. Probably the simplest one is divide-and-conquer, but I will explain a bit unusual one.

Define a polynomial $Q(t) = x_{-2}t^{-2} + x_{-1}t^{-1} + x_0 + x_1 t + x_2 t^2$. Without the condition that Nikita immediately loses after dropping below zero, we could have solved the problem just by calculating the m-th power of Q.

Define polynomials $P_c(t) = \sum t^k \text{Ways}(c, k)$, where $\text{Ways}(c, k)$ is the number of ways in which the secret number will change by c (c may be positive or negative) in k steps if we forget about the condition that the secret number should be nonnegative all the time (then the initial value of the secret number doesn't matter). Every single $P_c(t)$ may be calculated in $O(n \log^2 n)$ time: $\text{Ways}(c, k)$ is just the coefficient at t^c in polynomial $Q^k(t)$, and all such coefficients can be calculated by divide-and-conquer.

Then, let us define two polynomials $R_0(t)$ and $R_{-1}(t)$. Here, $R_i(t) = \sum t^k B(k, i)$, where $B(k, i)$ is the number of ways to lose exactly after k steps, finishing at number i. If we compute these polynomials, we will easily get the total probability of losing.

But we may note that:

$$(R_0, R_{-1}) \cdot \begin{bmatrix} P_0 & P_{-1} \\ P_1 & P_1 \end{bmatrix} = (P_{-n}, P_{-n-1})$$

Here, we mean a vector of polynomials and a matrix of polynomials. Why? On the right side, there is the number of ways in which the secret number can be changed to 0 or -1 respectively, and on the left side, there is the number of ways to reach 0 or -1 for the first time multiplied by the number of ways to reach 0 or -1 from 0 or -1.

Then, we can invert the matrix and obtain (R_0, R_{-1}): exactly what was required.

The complexity is $O(n \log^2 n)$.

Problem D. Fairy Chess

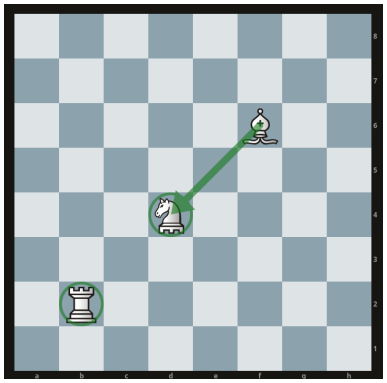
In this problem, a bit mask is an unsigned 64-bit integer.

For each of the 64 squares, precalculate 3 bit masks: the squares attacked by a knight, a bishop, and a rook, *assuming there are no other pieces on the board*. We can consider attacks on an empty board because the problem prohibits one piece from attacking another. There are two possible situations: either the piece we want to place on the board attacks another piece (in this case, we ignore this move), or it doesn't, and in this case, we can consider the board as empty and use the precalculated values.

The game state is represented by a bit mask of the previously placed pieces and a bit mask of the squares attacked by the placed pieces. To generate moves, we iterate over the complement of the bitwise-or operation of these two bit masks (we iterate over the squares where there are no pieces and which are not attacked by the previously placed pieces). In the end, we need to write a recursive function. If we find a move that leads to a position where the opponent loses, then the current position is a winning position. If there is no such move, then the position is a losing position.

As an optional optimization, on the first move, we can consider placing a piece only on squares `a1`, `b1`, `b2`, `c1`, `c2`, `c3`, `d1`, `d2`, `d3`, `d4` (due to the symmetry of attacking pieces).

Explanation of placing a bishop on square `f6`. We don't care that on an empty board, we attack the rook on square `b2`. We discard this move anyway, since the bishop is attacking the chancellor on square `d4`.



Problem E. Fair Elections

Consider the following $O(n^3)$ dynamic programming. Imagine that the first $a+b+c$ guys have already voted, and a guys have chosen the first candidate, b chose the second one and c chose the third one. Who will win?

Initial states are the ones with $a+b+c=n$. The value of $dp[a][b][c]$ can be calculated from $dp[a+1][b][c]$, $dp[a][b+1][c]$, and $dp[a][b][c+1]$: we choose the best of these three values for $a+b+c+1$-st voter.

The easiest way to speed it up is the following: consider fixed a and $a+b+c$ and store the dynamic programming values as segments of equal values. Then, we may recalculate dynamic programming in $O(S_{a+b+c})$, where S_{a+b+c} is the number of segments with fixed $a+b+c$. The total complexity is $\sum S_i$, and it somehow works quickly, although we can not prove that it always does.

We have a proven solution as well. Let us say that a triplet (a,b,c) is a *state*, and all triplets with $a+b+c=l$ are a *level*. Two states (a_1,b_1,c_1) and (a_2,b_2,c_2) on the same level are *adjacent* if $|a_1-a_2|+|b_1-b_2|+|c_1-c_2|=2$.

Let us say that a state s is below t in direction i if there is a sequence of adjacent states $v_0=s, v_1, v_2, \ldots, v_y=t$, such that i-th coordinate of states $\{v_k\}$ is strictly increasing.

We may prove the following by induction:

(1) If state s_1 is below state s_2 in direction 1 and state s_2 is below s_3 in direction 1, and we know that $dp[s_1]=1$ and $dp[s_2]=2$, then $dp[s_3]$ cannot be 3. The same is true for all permutations of $(1,2,3)$.

(2) If states s_1 and s_2 have the same first coordinate, and s_1 is below s_2 in direction 2, and we know that $dp[s_1]=2$, then $dp[s_2]$ cannot be 3. The same is true for all permutations of $(1,2,3)$.

Let us denote by T_i (i-th lower set) the set of all states for which there aren no states below them with dp equal to i (for $i=1,2,3$).

Then we may prove that

(3) Any state is in some T_i.

Look at some lower set (without loss of generality, T_1). Then, by property (2), for any fixed first coordinate a_0 there is some number $bar[a_0]$ such that, if the second coordinate of a point in the lower set (b_0) is less then $bar[a_0]$, then $dp[a_0][b_0][c_0]=3$, else it is 2. Moreover, it is enough to store lower sets (and they can be stored in linear memory) and functions bar to restore all dp's by property (3), and values of dp may be restored in $O(1)$.

How to recalculate this information? We may note that if, for some state (a,b,c), the points $(a+1,b,c)$, $(a,b+1,c)$, and $(a,b,c+1)$ were in T_i on the previous level, then point (a,b,c) will be in T_i in this level.

So we may take such points in T_i's (and recalculate bar's) and then expand sets T_i element by element. The complexity is $O(n^2)+T$, where T is the number of expand operations. But, on each level, the total size of sets before expansion decreases at most linearly, so the total number of expands is at most quadratic, and the complexity is $O(n^2)$.

If anyone has a simpler proven solution, or an idea how to prove the first solution, please tell us.

Problem F. Exactly Three Neighbors

This problem is well suited for solving in a team: while the teammates write code for other problems, one can take the time to draw the answers on paper. Here are the possible successful outcomes:

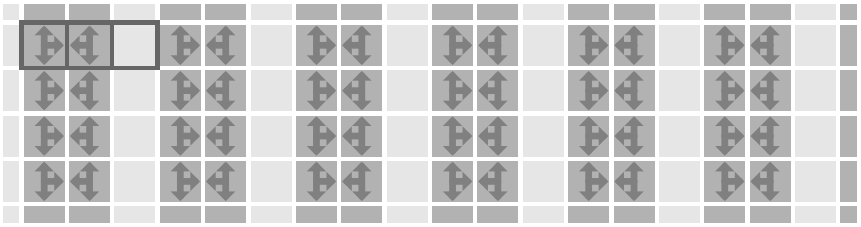
- drawing all interesting cases;
- coming up with an efficient way to brute force the possible rectangles;
- coming up with a way to construct an answer for any fraction.

Let us consider these in order.

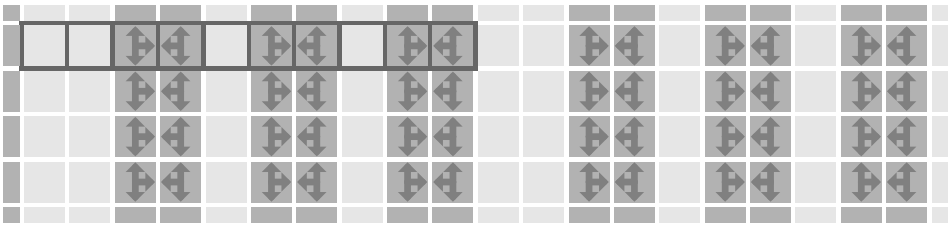
Solution 1: case analysis on paper.

Observation: we are constructing a rectangle with glued edges, so we can consider the solution lying on the surface of a torus.

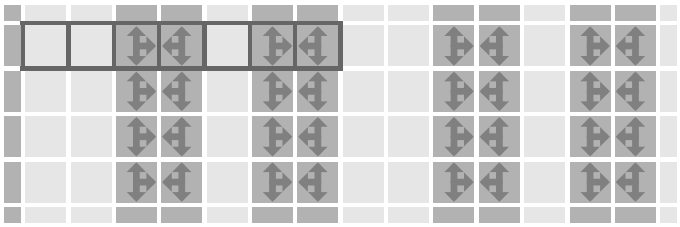
The simplest construction is obtained for the fraction 2/3: a strip of white squares along with two strips of black squares. The size of the minimum rectangle is just 1×3:



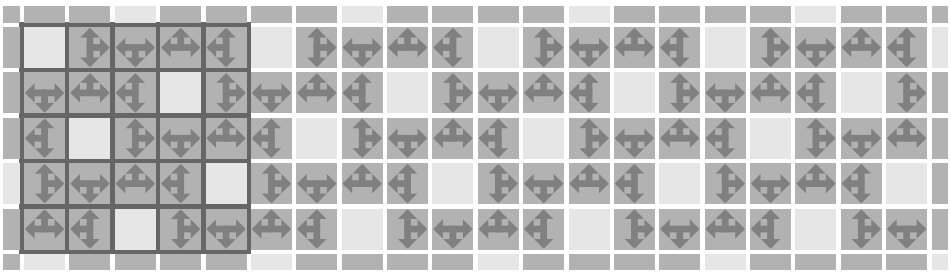
We can enlarge the white strips to obtain smaller fractions. For example, here is the solution for 3/5 with a 1×10 rectangle:



And here is the solution for 4/7 with a 1×7 rectangle:



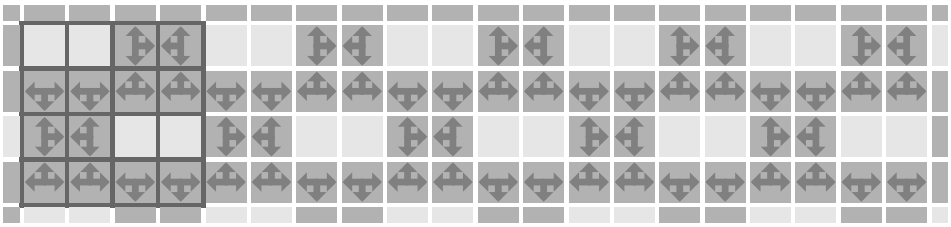
The limit is fraction 4/5, white squares are a chess knight's move apart. The solution with a 5×5 rectangle:



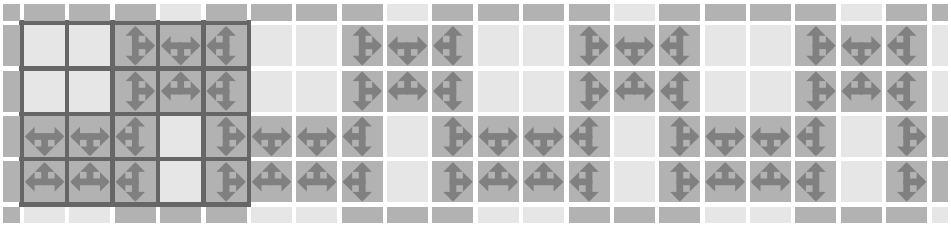
Here, each white square already has exactly four black neighbors out of the four possible ones. Therefore, obtaining fractions greater than 4/5 is impossible.

What is left is to solve the problem for fractions between 2/3 and 4/5. There are a few such fractions with a denominator up to 10. Below we show the most compact example for each of them.

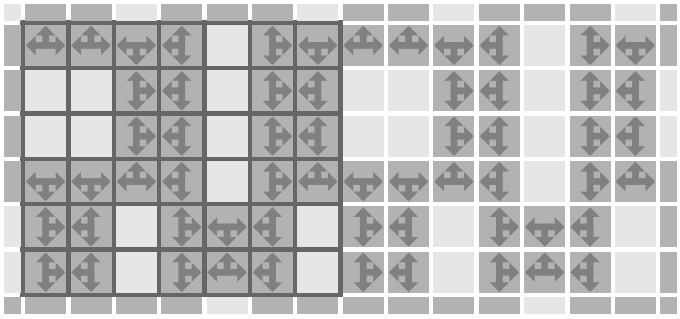
Here is a solution for the fraction 3/4 with a 4×4 rectangle:



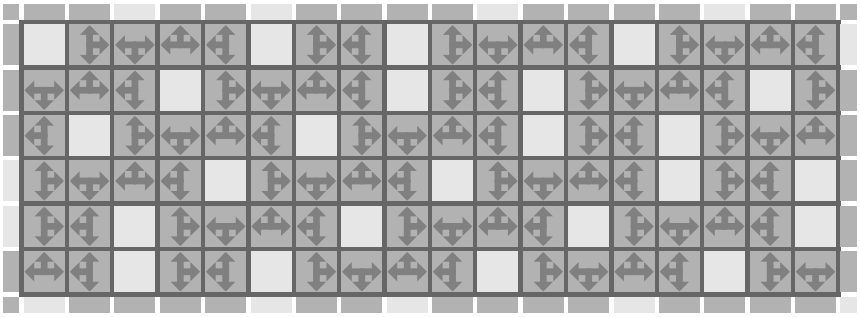
Here is the answer for 7/10 with a 4×5 rectangle:



Here is the answer for 5/7 with a 6×7 rectangle:



Finally, here is the answer for 7/9 with a 6×18 rectangle:



Solution 2: brute force search over rectangles.

When constructing solutions, we can observe and explain the following properties:

- White squares connected by sides form rectangles.
- There cannot be a black square between two white squares.

This is enough to perform a recursive brute force search for the possible rectangles.

- Fix the rectangle width w: for example, from 1 to 10.
- List the *valid* rows in this rectangle: these are colorings in which there is no black square between two whites (cyclically); there are a total of 2^w possible rows, and even fewer valid ones.
- The rows are short, so it is convenient (but not necessary) to store them as bit masks.
- Fix the first two rows of the rectangle.

Each subsequent row of the rectangle is constructed as follows.

- First, copy the previous row.

- Each segment (cyclic) of white squares is either left entirely white or turned entirely black.
- For each black square in the previous row that already has three black neighbors, make the square below it white.
- Check that a valid row has been obtained.

Fix the maximum depth h of the search in advance: for example, 25.

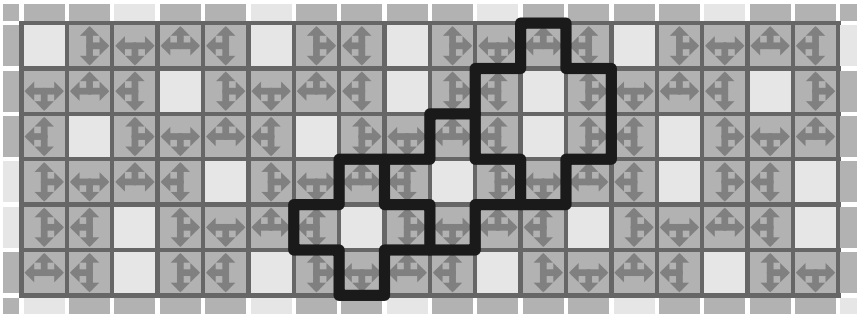
- Recursively construct the next rows up to this depth in all possible ways.
- As soon as the last two constructed rows match the first two rows, we have obtained a periodic tiling: the period is the rectangle without the last two rows.
- In the obtained tiling, calculate p/q, and if we got a desired fraction, remember the solution.

As the above solutions show, it is enough to perform the search with parameters $w = 6$ and $h = 20 + 2$. The search completes almost instantly.

The search can also be done from the other side, by swapping rows and columns, as shown in the images. In this case, it is sufficient to use $w = 20$ and $h = 6 + 2$, and this search takes a few seconds.

Solution 3: constructing the answer by the power of thought.

Let us take a close look at the image for 7/9:



In fact, the periodic part in it is much smaller than 6×18, it just does not have a straight boundary. The periodic part highlighted in the image consists of two separate white squares and a white rectangle 2×1, as well as the black squares that share a side with them. The distance between neighboring white squares in one periodic part is a knight's move. The distance between the outermost white squares in different periodic parts is also a knight's move.

Using "crosses" of 1×1 and 2×1 white squares with the surrounding black squares, we can obtain any answer from 3/4 to 4/5. If the coloring is already constructed, the corresponding rectangular period can be found by checking all possible sizes. For fractions less than 3/4, simpler constructions can be invented.

Problem G. Lake

Solution 1. A maximum of 4 adults and 3 children can sit in a vehicle, or 3 adults and 4 children. Therefore, if there are more adults than children, we seat 4 adults and 3 children, otherwise we seat 4 children and 3 adults. We repeat this process until everyone is transported.

Solution 2 using formulas:

- The number of home trips is at least $\lceil n/4 \rceil$.
- The number of home trips is at least $\lceil m/4 \rceil$.
- The number of home trips is at least $\lceil (n+m)/7 \rceil$.

We can prove that these cases actually provide an exact estimate. Indeed, if the ratio is within [3/4, 4/3], we seat them as in the first solution. But if it is not, those who are significantly more in number dominate, even if they are always seated in the middle seat, and this is taken into account in the formula.

Problem H. Forbidden Set

If the set does not contain at least one of the digits 2, 3, 5, 7, then the minimum prime number is one-digit integer formed by the smallest of these digits that is missing in the set.

If all of these digits are present, but the digit 1 is missing, then the answer is 11.

Thus, we are left with the set of digits $\{0, 4, 6, 8, 9\}$.

Note that if the set only lacks even digits greater than 2, then the resulting number will always be even and greater than 2, which means there are no corresponding prime numbers, and the answer is -1.

Similarly, if the set only lacks digits from the set $\{0, 6, 9\}$, then the resulting number will always be greater than 3 and divisible by three.

The remaining situation is when the set lacks 9 and at least one of the remaining digits that are not divisible by three (that is, 4 or 8).

Let us generate prime numbers up to 1000 and find those whose digit set satisfies these requirements. The first three of them are 89, 409, and 449. So, if the set does not contain 8 and 9, the answer is 89. If the set contains 8, but lacks 4 and 0, the answer is 409. If the set contains 8 and 0, but lacks 4 and 9, the answer is 449.

Problem I. Colorful Cycles

Note that we may consider all biconnected components separately. From now on, we will consider only biconnected graphs.

Consider such two obviously necessary conditions for a "colorful cycle" to exist (because even the cycle itself already satisfies these conditions):

1. There are edges of all 3 colors in the graph.
2. Let us call a vertex *monochrome* if all edges adjacent to it are of the same color. There must be at least 3 non-monochrome vertices in the graph.

It can be proven by induction or case analysis that these two conditions are sufficient. Both of them are easy to check in linear time.

Complexity: $O(n + m)$.

Problem J. Range Sets

First, let us consider a simpler version of the problem, where x can only take one value. This problem can be solved using a single data structure, `std::set`, in which we will store all the intervals that contain x. With each modification operation, some intervals may be removed, but no more than one new interval can be added, and at most two old intervals can be modified. Queries of the form "`?`" are processed trivially. It is easy to see that the amortized time complexity of this solution is $O(q \log q)$.

Now, let us generalize this solution to solve our problem. We will create such a data structure for each possible value of x. It remains to notice that each addition, deletion, or modification of an interval also changes the answer value by one on some interval. To quickly handle such changes, we can use an implicit segment tree, in which we will maintain the answers to all possible queries of the form "`?`". It is easy to see that the total number of increment and decrement queries on an interval is bounded by $O(q)$, and each query of the form "`?`" can be expressed as a single query to compute the value at a point. The overall time complexity is $O(q(\log q + \log n))$.

As an alternative approach, instead of using a pair of `std::set` and an implicit segment tree, you can use your own implementation of some binary search tree and achieve a time complexity of $O(q \log q)$.

Problem K. Integer Half-Sum

We can solve the problem entirely by solving cases, from small to large:

- If there is one number, it will remain.
- If there are two numbers, they have different parity, so they cannot turn into one.
- If there are at least three numbers in the segment $[\ell, r]$, we can always obtain $r-1$:
  $\underline{1}\ 2\ \underline{3} \to \underline{2}\ \underline{2} \to 2$
  $\underline{1}\ 2\ \underline{3}\ 4 \to \underline{2}\ \underline{2}\ 4 \to \underline{2}\ \underline{4} \to 3$
  $\underline{1}\ 2\ \underline{3}\ 4\ 5 \to \underline{2}\ \underline{2}\ 4\ 5 \to \underline{2}\ \underline{4}\ 5 \to \underline{3}\ \underline{5} \to 4$
  $\underline{1}\ 2\ \underline{3}\ 4\ 5\ 6 \to \underline{2}\ \underline{2}\ 4\ 5\ 6 \to \underline{2}\ \underline{4}\ 5\ 6 \to \underline{3}\ \underline{5}\ 6 \to \underline{4}\ \underline{6} \to 5$
  And so on.

Problem L. esreveR Order

Let us represent each number as an array v of 8 bytes and compare it lexicographically with the array v' obtained by reading the vector v from right to left. If $v > v'$, then the control bit is set to 1, otherwise the control bit is set to 0.

Knowing the value of the control bit for each number, we can restore the original byte order (if the computed value after receiving does not match the transmitted value, we change the order).

Now, the control bits need to be transmitted. The total number of control bits does not exceed 1000. We will transmit the bits in the control integers b_i (appended at the end of the array after the original numbers). In this case, the 63rd bit of each b_i will be `0` for all i, and the 7th bit will be `1`. This will allow us to restore the correct byte order in b_i itself: if the 63rd bit of the received number is `1`, we change the order.

Thus, each b_i will have 62 bits remaining, resulting in no more than $\lceil 1000/62 \rceil = 17$ additional numbers. To simplify the restoration of the answer, we will add 17 numbers regardless of n (unused control bits do not affect the restoration; they can be filled with zeros, for example).

Problem M. Good Splits

Suppose that we already have some splitting into pairs. How to check whether it is good or not?

Consider the following graph: vertices correspond to the segments, there is an edge between two vertices if and only if their segments intersect, but are not contained in each other. Then, the splitting is good if and only if this graph is bipartite.

Let us first solve a simpler problem: calculate the number of ways to obtain a good coloring together with coloring of the corresponding graph in two colors (denote this as $col[2t]$). But this is simple: we should just choose which vertices correspond to the segments below the line and which correspond to the segments above the line, and then connect these vertices.

Then we may calculate $conn[2n]$: the number of ways to obtain a good splitting together with coloring of the corresponding graph in two colors, with an additional condition that this graph is connected. How to calculate this? Consider some splitting into pairs. If it has more than one component, consider the component that contains the leftmost vertex. Then, the vertices of this component will split all vertices into several parts, and there are no pairs containing vertices from different parts, cause in this case this segment will be in the same component with the first vertex. Therefore, we may calculate *conn* using *col* and the previously computed values.

But then the number of good splittings of $2t$ vertices with a single connected component is just $\frac{conn[2t]}{2}$. Then, we may calculate the total number of good splitting with dynamic programming similar to the one above: iterate over the connected component with the first vertex and go to the values of dynamic programming on parts.

Asymptotic complexity is $O(n^3)$. We may use FFT to speed up solution to $O(n^2 \log n)$, but it is not necessary.

Problem N. Shoes

Note that the time spent shopping on a particular day depends only on the number of visited stores and the positions of the two outermost stores. Let's call a day *interesting* if we managed to visit stores on both sides of the hotel (that is, stores with strictly positive and strictly negative coordinates). The key observation necessary to solve the problem is that there exists an optimal solution in which the segments of visited stores on interesting days can only be nested within each other: for example, a configuration where we visited stores with coordinates -2 and 1 on one day, and stores with coordinates -1 and 2 on another day, is not allowed. To prove the existence of such a solution, we can consider an optimal solution that minimizes the total amount of time spent outside the hotel. If this solution contains a "forbidden" configuration, we can change our plans to further reduce the amount of time spent shopping.

Now, notice that since we can rearrange the days in any order, we can assume that we always visit either the store with the highest coordinate among the remaining ones, the store with the lowest coordinate, or both. Moreover, thanks to the aforementioned observation, we can assume that if today is an interesting day, we must visit both the store with the lowest coordinate and the store with the highest coordinate, as both of these stores cannot be visited on two different interesting days. Also, without loss of generality, we can assume that each day we visit several (possibly zero) stores with the lowest remaining coordinates and several (also possibly zero) stores with the highest remaining coordinates. Now we can write the following dynamic programming: `dp[`ℓ`][`r`]` is the minimum number of days required to visit all stores with numbers from ℓ to r after sorting them by coordinates. Currently, this dynamic programming has $O(n^2)$ states and $O(n^3)$ transitions: two transitions for uninteresting days and potentially a linear number of transitions for interesting days. There are two ways to optimize the number of transitions.

The first way is to replace the coordinates of the dynamic programming with a pair of its length and left endpoint. In this case, all transitions for interesting days can be expressed as a single operation of finding the minimum on a segment. However, solutions that use data structures that add a logarithmic factor to the overall complexity, such as segment trees or sparse tables, may have a problem with the time limit, and solutions with sparse tables may not fit within the memory limit, as they require $\Theta(n^2 \log n)$ memory. Therefore, it is recommended to use more advanced data structures, such as a version of a sparse table that can be built in $O(n\alpha(n))$ time and answer queries in $O(\alpha(n))$ time, where α is the inverse Ackermann function, or specific data structures for the RMQ problem with linear construction and constant-time query processing (for example, the well-known Farach-Colton and Bender structure or the Fischer and Heun structure used in some of the author's solutions).

The second way is to change the dynamic programming itself. To do this, we need to add an additional dimension to the dynamic programming: the mask of which of the two edges we visit. Also, we replace the value of the dynamic programming with a pair of the number of days and the time spent in the last unfinished day. In this case, the number of states in the dynamic programming will still remain quadratic, as the new dimension can only take a constant number of different values, but the number of transitions can also be reduced to a constant number, as we only need to consider cases when a new day begins and when we visit one of the two possible unvisited stores.

Both solutions work in $\Theta(n^2)$ time and require $\Theta(n^2)$ memory.