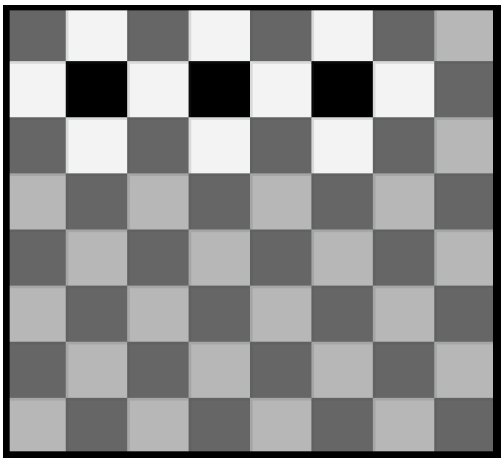
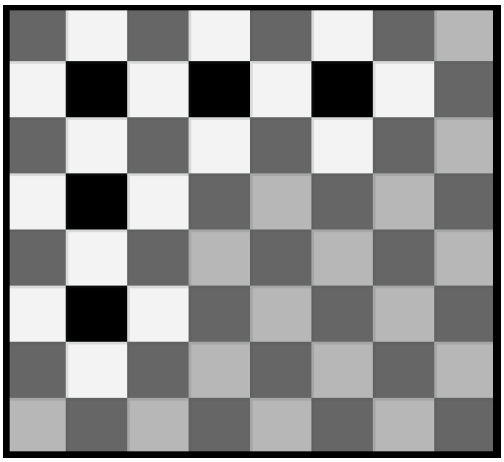
Problem Tutorial: "Designing a New Logo"

For the first subtask, you can fill the logo from top to bottom taking two horizontally adjacent cells at a time.

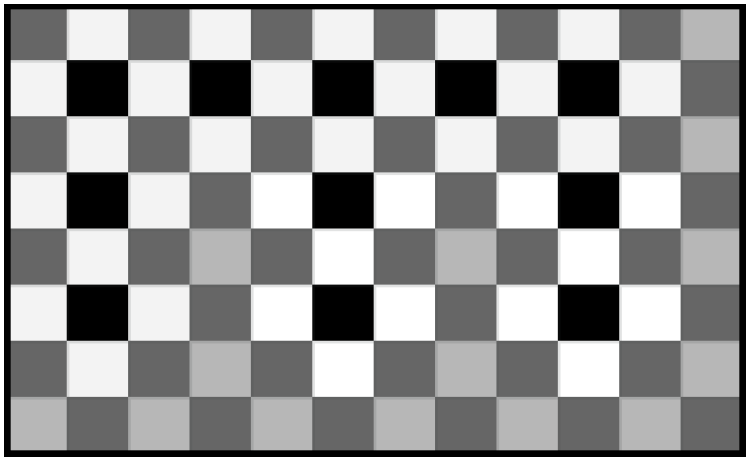
For the second subtask, select cells $(2,2),(2,3),(2,4),\ldots(2,2b)$ (this gives you b black and $b-1$ white cells). Then, add white cells adjacent to selected black cells. Since we can add any number of white cells up to $2b+2$ white cells, we can get to any number of white cells.



For the third subtask, select cells $(2,j)$ $(2 \le j \le 4m-2)$, then cells $(i,2)$ $(3 \le i \le 4n-2)$. Similarly, add adjacent white cells to the base solution to match the required count. This solution gets up to $b=(2m-1)+(2n-2)$ black cells, with $b-1$ white cells between them, then we can add any number of white cells up to $2b+2$.



For the last subtask, again, start with the base solution that looks like $(2,j)$ $(2 \le j \le 4m-2)$, and $(i,2+4j)$ $(3 \le i \le 4n-2, 2 \le 2+4j \le 4m-2)$, then add the necessary amount of white cells. The number of black cells in this solution is bounded by $b=(2m-1)+(2n-2)\cdot m = 2nm-1 \ge nm$, which is enough to solve the problem.



Problem Tutorial: "Even Tree"

First, note that the problem asks to find a spanning tree with an even number of edges of odd weight. Consider the connected components restricted to only even edges. Within each component, we can exactly choose a spanning tree with all even edges, or a spanning tree with exactly one odd edge, if a connected component contains one. So there are three cases:

1. We have an odd number of even-edge connected components — then we need to build each internal part of the components from all even parts and connect all the components;
2. We have an even number of components, and there are no odd edges inside the components (that is, between two vertices from the same component) — then we can't build a spanning tree of even weight;
3. We have an even number of components, and there is an odd edge inside the components — then we must take this edge and take the remaining edges in the same way as in the first case (that is, we will replace one even edge with an odd one).

This process can be implemented, for example, as follows: let's build a tree traversal in the DFS order. If this tree is already even (that is, the total weight of the edges is even), then we output it. Otherwise, we need to find a back edge (in the DFS tree) that can be replaced by an edge of different parity. If we fail to find such a pair of edges, the answer does not exist.

Problem Tutorial: "Primle"

Let's discuss solutions for all subtasks in order.

The first two subtasks were designed to make any non-trivial solution pass. For example, you can ask random numbers/random primes and then figure out the answer based on the information received. With high probability, we will see at least one `+` in each of the five positions. To do it in hard mode, you need to be able to generate a random prime number.

In the third and fourth subtasks, you can come up with a set of 9 numbers such that every position contains nine out of ten digits. In easy mode you can use `11111`, `22222`, ..., `99999`. In hard mode, you should find nine numbers with these properties (except for the least significant digit). Given this information, you can find out the secret prime number.

In the fifth and sixth subtasks, you can optimize the previous solution. Ask `01234` and `56789` (or `65423` and `91807` in hard mode). We learn the set (without multiplicities) of digits in the secret prime using the responses to these queries. Similarly to the previous solution, figure out the answer using only four queries.

Finally, there are many possible solutions for the last two subtasks. Let's describe one possible approach. During the process of guessing the number, at each step, we have a set of prime numbers that satisfy all the information we have. Every query splits this set into 3^5 subsets, out of which we only retain one. Whenever the current set has only one number left, it is the secret prime. Since the problem asks for you to find the secret prime in L queries for all possible secret prime, it makes sense that we want to make the "worst"possible case as good as possible. Thus, let's pick a new guess in such a way that minimizes the maximum of all 243 sets. If you follow this strategy, you solve almost all primes in ≤ 4 queries, and only about 50 primes will require 5.

Bonus: come up with a strategy that solves any prime number in 4 queries.

Problem Tutorial: "Add and Multiply"

First, if $a_i \leq b_i$ for all i (or the other way around), the solution exists if and only if the two input arrays are the same. In this case, print $c_i = 0$, otherwise, there is no solution.

Now, we will constructively prove that if there is a pair i, j, such that $a_i < b_i$ and $a_j > b_j$, then there is a solution.

First, let's solve for $n = 2$. Without loss of generality, $a_1 < b_1$, $a_2 > b_2$. Consider $c_1, c_2 \geq 0$, where $(a_1+c_1)(a_2+c_2) = (b_1+c_1)(b_2+c_2) \Rightarrow (a_1+c_1)(a_2+c_2)-(b_1+c_1)(b_2+c_2) = 0$. Expanding the products, we get $a_1a_2-b_1b_2+c_1(a_2-b_2)+c_2(a_1-b_1) = 0$. Rearranging the terms, $c_1(a_2-b_2)-c_2(b_1-a_1) = a_1a_2-b_1b_2$. Turns out, this linear Diophantine equation always has a nonnegative solution.

Let's apply the extended Euclidean algorithm. Recall that for this algorithm to work, we need $a_1a_2 - b_1b_2$ to be divisible by $d = \gcd(a_2-b_2, b_1-a_1)$. This is always true, since if $a_2 = b_2 \pmod d$, $b_1 = a_1 \pmod d$, then $a_1a_2 - b_1b_2 = b_1b_2 - b_1b_2 = 0 \pmod d$. Since $a_2 - b_2 > 0$ and $b_1 - a_1 > 0$, by taking the solution of $(a_2 - b_2)x - (b_1 - a_1)y = 0$ and repeatedly adding it to the base solution, we can always make it nonnegative.

Let's solve the general case. To do it, we will basically merge all i with $a_i > b_i$ into a single pair. Consider the example:

a	9	10
b	5	3

Notice that if we add 6 to the second column, and take the product, 9 cancels out:

a	9	16
b	5	9

Now we can replace these two pairs with a pair $a = 16$ and $b = 5$. Adding x to the new pair is the same as adding x to both original pairs.

In general, take all columns with $a_i > b_i$, sort them in order of decreasing b_i, and, in this order, make a_i and b_{i+1} equal. On each step we add some nonnegative number to c_i, because $b_{i+1} \leq b_i < a_i$. Thus, in $O(n)$ time we turn all columns with $a_i > b_i$ into one. We apply a similar process to pairs with $a_i < b_i$. All that's left is to solve for two pairs.

Problem Tutorial: "Draft Laws"

In this problem, we need to find the number of ways to color the tree in k (modulo a prime number), given colors for some vertices.

Subtask 1 can be solved with brute force over all k^n colorings.

Subtask 2 is based on the fact that there are precisely two ways to color a tree in two colors: after fixing the color of vertex 1, the colors of all other vertices can be determined uniquely. Thus, we can try both colorings and check whenever they satisfy the conditions.

In *subtask 3* there are no pre-colored vertices. It is easy to see that the answer is $k(k-1)^{n-1}$: for vertex 1, we can choose any of k colors; then, for all neighbors of vertex 1 we can choose any of $k-1$ remaining colors; for all vertices at a distance 2 from the first vertex we also have $k-1$ possible options, and so on.

Subtasks starting from 4 can be solved by a dynamic programming approach. Root the tree at the vertex 1. Let $\mathrm{dp}_{v,c}$ be the number of ways to color the subtree of v (satisfying all conditions) such that vertex v has color c. Then, the value $\mathrm{dp}_{v,c}$ can be calculated knowing the dp values of all children. In the case $a_v = 0$ we get:

$$\mathrm{dp}_{v,c} = \prod_{u \text{ is a child of } v} \sum_{d \neq c} \mathrm{dp}_{u,d}$$

because for each child u we can independently choose any color $d \neq c$. In the case $a_v \neq 0$ the formula is the same except that $\mathrm{dp}_{v,c} = 0$ when $c \neq a_v$.

The naive way of calculating this dp gives an $O(nk^2)$ solution (for each of $n-1$ edges we go through all $O(k^2)$ pairs of colors (c,d)), which passes *subtask 4*.

We can easily optimize this solution in the following way: when calculating $\mathrm{dp}_{v,c}$ and when child u is fixed, instead of calculating the sum over all $d \neq c$, let's also calculate $\mathrm{sum}_v = \sum_{c=1}^{k} \mathrm{dp}_{v,c}$ for all vertices beforehand. Then, we can just take $\mathrm{sum}_u - \mathrm{dp}_{u,c}$ instead. This is an $O(nk)$ solution, which solves *subtask 5*.

To solve other subtasks, we need to remove k from the time complexity. Out of all k colors, only at most n are present in the tree (on pre-colored vertices). Let v be any vertex. Note that if c_1 and c_2 are two colors that are not present in the subtree of v, then the following equality holds:

$$\mathrm{dp}_{v,c_1} = \mathrm{dp}_{v,c_2}$$

We can prove it by establishing a one-to-one correspondence between colorings of the subtree of v, where v has color c_1, and colorings, where v has color c_2: it is sufficient to swap colors c_1 and c_2 (color in c_2 all vertices which were colored in c_1 and vice versa). This transformation doesn't break any constraints. Thus, there are equally many of these colorings. This gives us the **key idea**: for all colors which are not present in the v subtree, we store one common dp value, which we denote as other_v.

To solve *subtask 6*, it is sufficient to get rid of all colors which are not present on the entire tree. Let's implement "coordinate compression": re-enumerate all colors present on the tree with numbers $0, 1, \ldots, D-1$, where D is the number of distinct colors in the tree. Then we calculate similar $\mathrm{dp}_{v,c}$, but with $0 \leq c < D$, and at the same time calculating and considering other_v. Thus, when $a_v \neq 0$ we get:

$$\mathrm{dp}_{v,c} = \prod_{u \text{ is a child of } v} \left(\mathrm{sum}_u + (k-D) \cdot \mathrm{other}_u - \mathrm{dp}_{u,c}\right)$$

This solution works in $O(nD)$ (where D is the number of distinct colors on the tree). In particular, we can estimate this as $O(n \cdot \min(n,k))$ or as $O(n^2)$.

In *subtask 7* the tree is a path (a bamboo). Let's split it into parts surrounded by pre-colored vertices. Previously we calculate A_m — the number of colorings of a path with m vertices, where the colors of ending vertices are fixed and equal; and B_m — the same quantity, but if the colors of ending vertices are fixed and different. We can calculate those using the following recurrence relations:

$$A_m = (k-1)B_{m-1}, \quad A_1 = 1$$
$$B_m = A_{m-1} + (k-2)B_{m-1}, \quad B_1 = 0$$

Each part can be independently colored in A_m or B_m ways (depending on the colors of the ending vertices). Thus we can take the product over all parts.

Now let's move to *full solution*. In the same way we will be calculating subtree dp. For vertex v we will store values $\mathrm{dp}_{v,c}$ (for colors c which are present in the subtree of v) in an associative array H_v (for example, std::map in C++) in which colors c will be keys and numbers $\mathrm{dp}_{v,c}$ will be values. We will also

store sum_v — the sum of all the values in H_v, cnt_v — the number of values in H_v and other_v — the dp value for all other colors.

We will use the "smaller to larger" technique. For each vertex v we consider its child b with the largest size of H_b. Firstly, in $O(1)$ we set $\text{H}_v := \text{H}_b$ (with some details to make values correct), and then for all other children $u \neq b$ we will move all elements of H_u into H_v straight-forward, by going through all of the container H_u. We claim that these iterations in subtrees of other children will take $O(n \log n)$ moves in total. It is because we always move an element into a container that previously had more elements. So, the size of the container of color c at least doubles each time c is moved. It means that color c will move at most $O(\log n)$ times.

At first, let's figure out how to achieve the initial dp values for vertex v from values in H_b. For each color c in H_b, we want to calculate:

$$\text{dp}_{v,c} := \text{sum}_b + (k - \text{cnt}_b) \cdot \text{other}_b - \text{dp}_{b,c}$$

In other words, if we denote $T = \text{sum}_b + (k - \text{cnt}_b) \cdot \text{other}_b$, we need to firstly set $\text{H}_v := \text{H}_b$ and then for each value x in H_v we need to apply linear function:

$$x \mapsto -x + T$$

These operations can be performed in a "lazy" way: instead of applying this function for all values by manually going through them, we will save this function next to the container and denote it as $f_v(x)$. To get the actual value of $\text{dp}_{v,c}$ we just need to apply function $f_v(x)$ to the number that is actually stored in H_v. Initially, $f_v(x)$ is equal to the identity function $x \mapsto x$. When operation "apply linear function $g(x) = px + q$ to all values in H_v" takes place, we will update coefficients of $f_v(x)$ and values of sum_v and other_v in such way (before we had $f_v(x) = ax + b$):

$$\begin{aligned} f_v(x) &:= g(f_v(x)) = (pa)x + (pb + q) \\ \text{sum}_v &:= p \cdot \text{sum}_v + q \cdot \text{cnt}_v \\ \text{other}_v &:= p \cdot \text{other}_v + q \end{aligned}$$

Thus, processing child b is now a single operation that we perform lazily in $O(1)$ time.

Now consider other children $u \neq b$. If color c is not in H_u, then

$$\text{dp}_{v,c} := \text{dp}_{v,c} \cdot (\text{sum}_u + (k - \text{cnt}_u - 1) \cdot \text{other}_u)$$

Or, if we denote $Q = \text{sum}_u + (k - \text{cnt}_u - 1) \cdot \text{other}_u$:

$$\text{dp}_{v,c} := \text{dp}_{v,c} \cdot Q$$

If c is in H_u, then:

$$\text{dp}_{v,c} := \text{dp}_{v,c} \cdot (\text{sum}_u + (k - \text{cnt}_u) \cdot \text{other}_u - \text{dp}_{u,c})$$

Let's first apply linear function $x \mapsto Qx$ to all values in H_v. Then, we will manually go through all colors c in the subtree of u, and for each one of them we will make the following change:

$$\text{dp}_{v,c} := \text{dp}_{v,c} \cdot Q^{-1} \cdot (\text{sum}_u + (k - \text{cnt}_u) \cdot \text{other}_u - \text{dp}_{u,c})$$

Here, Q^{-1} is the multiplicative inverse of Q modulo $M = 10^9 + 7$. However, this fails when $Q \equiv 0 \pmod{M}$. In this case we proceed the other way: let's first calculate and save the new values of $\text{dp}_{v,c}$ for all colors c in H_u. After operation "multiply all the values in H_v by 0" all values in H_v will become zero. So, we can completely clear H_v and also set $\text{sum}_v = \text{other}_v = 0$ and set $f_v(x)$ equal to the identity function $x \mapsto x$. After that we can again go through all colors c in H_u and set the previously saved values.

Finally, we achieved an $O(n \log^2 n)$ solution. Note that we can also get $O(n \log n)$ by using std::unordered_map instead of std::map. However, in practice it is usually slower.