Problem Tutorial: "Sheet Metal"

To solve the problem, consider all possible ways to place the rectangular sheet over the form:

- if the form is greater than or equal to the sheet, that is, $w_1 \le w_2$ and $h_1 \le h_2$; then the answer is 0 because the form can cover the sheet entirely.
- if the form is smaller than the sheet, that is, $w_1 > w_2$ and $h_1 > h_2$, then, in any case, the sheet would not fall apart into two pieces, and the area is $w_1 \cdot h_1 - w_2 \cdot h_2$.
- Let's suppose $w_1 \le w_2$ and $h_1 > h_2$. Then you should divide the sheet into two equal pieces; in this case, the maximum remaining area is minimized. The answer is $\frac{w_1 \cdot (h_1 - h_2)}{2}$.
- If $w_1 > w_2$ and $h_1 \le h_2$, then, similarly, dividing the sheet into two equal pieces gives $\frac{h_1 \cdot (w_1 - w_2)}{2}$.

Of course, there are other cases when one of the rectangles is rotated 90 degrees. It would be convenient to extract the part above as a separate function to avoid code duplication. Then, take a minimum of two calls to that function.

Problem Tutorial: "Lunchtime Fruits"

Let's convert the problem statement to a more formal one: we have objects of four types A, B, C, and D. We can combine them into sets $AABC$, $AACC$, and $ABCCD$. Let's denote $X = AB$, $Y = AC$, $Z = BCD$. Then these sets turn into XY, YY, and YZ. Unfortunately, we can't determine X, Y, and Z uniquely; if we assemble them differently, we can get different results. But, if we know those values, we can determine if it's possible to assemble some predetermined number of sets.

Notice that each set has a Y. We can subtract the number of Ys at the start, and now we only have to deal with X, Y, and Z. One of the possible ways to proceed is as follows: (a is the number of objects of type A; b is the number of objects of type B, and so on; k is the total number of sets we would like to check). If $b + c \le a$, then we can assemble at most $b + c$ sets, so we check that $b + c \ge k$. In another case, we first create Xs and Ys evenly (if the number of objects of some type is too low, then take the closest to half), so you have as many Zs left as possible. The formula is $min(b, c, d, \frac{b+c-a}{2}) \ge k - a$.

Since we figured out how to check if we can assemble k sets, we can run a binary search to find the maximum k.

Problem Tutorial: "Sliding Dominoes"

Consider any cell covered by a domino tile. Consider the moment when it becomes empty for the first time. How does it happen? We would need to move the tile covering it in the direction away from the cell. This means that on the previous turn, some cell at a distance of two was empty.

Thus, for every cell A, there is a unique cell B that must be uncovered before we can move the domino tile and uncover A. We will call cell B a parent of cell A. Considering that there is one initially empty cell, every cell is either unreachable or makes up a rooted tree, which root is the initially empty cell.

What's left is to solve the problem for a tree: we traverse the tree, starting from the root, and we have to maximize the sum of visited vertices. This can be solved with dynamic programming: let `dp[v]` be the answer for the subtree v, including v itself. Then `dp[v]` $= \max(0,$ `cost[v]` $+ \sum_u$ `dp[u]`$)$. The solution works in linear time.

Problem Tutorial: "Lost in Translation"

Let's take on subtasks one at a time. We'll start with $m = 2n$, $k = 4$. You can, for example, replace all `0` with `AB`, and `1` with `CD`. Now, for example, if all occurrences of `C` were removed, you can remove all `A`s and recover the initial bitstring.

Consider the case $k = 3$ when you can only use three types of characters. Replace `0` with `AC`, and `1` with `BC`. If all `C`s are removed, you essentially get the original string. Cases when either `A` or `B` are removed are similar; let's consider what happens if the `A`s are removed. If the first character in the string (on the second stage) is `B`, then it is followed by `C`, and that means that the original string began with `1`. Otherwise, if the encoded string starts with `C`, it means that originally it had `AC`, and we deduce that it has to start with `0`.

There is a corner case with this solution: we can not distinguish the two cases when we remove `A` in `ACACAC A`, or `B` in `BCBCBC`, you need to handle strings like `000` and `111` in some particular way.

Let's try to generalize. Previously, we tried replacing `0` and `1` with some codes to recover the initial string using the knowledge of their structure. We can try encoding longer binary chunks of length c with codes over the alphabet with `A`, `B`, `C`. Then, to encode the binary string split it into chunks and encode each chunk individually. Encoded chunks must satisfy the following rule: for any letter deleted (`A`, `B`, or `C`), all the chunks must be different. It's not always sufficient because we can still get equal strings after the concatenation of two or more chunks. For example, there exists an encoding of 2 bits into 3 characters, but you can not make it into the encoding of 4 bits into 6 characters.

But, if the encoding is random enough, then it often happens that it's possible to recover the original string. Let's find the number of ways to restore the string, using dynamic programming on prefixes: let `dp[i][j]` be the number of ways to represent the first i characters as a concatenation of j chunks with some characters removed). If it's equal to 1, then, during the second run, you can restore the string using it (you can either hard-code it into your source or precalc each time). Based on the effectiveness of your implementation, you can build the mapping of 10-bit chunks into codes of length 19 or 18.

Finally, there exists a mapping of 20-bit chunks into codes of length 36, such that each code contains exactly 12 characters of each type `A`, `B`, and `C`. The advantage of using this code is that every string can be decoded uniquely because if you delete any character, each remaining code's length is exactly 24.

Problem Tutorial: "Summer School"

We are given an implicit bipartite graph: the first part is students, and the second is parallels. We need to find the size of the maximum subset of students such that there is a matching covering it. The first subtasks can be solved using bipartite matching and maximum flow algorithms.

To get the complete solution, we will use the generalized formula from Hall's theorem: the size of the minimum subset of students that can not be invited is $\max\left\{0, \max\limits_{A} |A| - f(A)\right\}$. The usual Hall's theorem checks if this is 0.

Notice that the set of good parallels for each level is a segment $[L_i, R_i]$. These segments are, in a sense, monotone: $L_i \le L_{i+1}$ and $R_i \le R_{i+1}$.

Let a_i be the number of students of level i, then $|A| - f(A) = a_i - k \cdot (R_i - L_i + 1)$, where A denotes the set of students of level i.

Our job is to find $\max\limits_{A} |A| - f(A)$, so we need to select some levels to maximize the expression. The problem is the segments in the second part can intersect, but we only need to count them once.

Let's define $d(i)$ as the maximum value of $|A| - f(A)$ if we only selected levels $\le i$ and level i is selected. Then, either $d(i) = d_j + a_i - k \cdot (R_i - L_i + 1)$ for some $j < i$, or $d(i) = d(i-1) + a_i - k \cdot (R_i - R_{i-1})$. This is based on the fact that if some two levels are selected, and their segments $[L_i, R_i]$ and $[L_j, R_j]$ intersect, then all levels between them should also be selected because the size of the right part is the same, but the size of the left part is increased.

In the first case, $d(i) = d_j + a_i - k \cdot (R_i - L_i + 1)$, we suppose that $[L_i, R_i]$ and $[L_j, R_j]$ don't intersect, in the second one that they do intersect. But, in the first case, you can consider all possible $j < i$ because if there is an intersection, then it counts all the vertices in the right part twice, which gives a smaller result.

To calculate $d(i)$, we need to store $d(i-1)$ and $\max\limits_{j<i} d_j$.

For simplicity, we'll refer to $a_i - k \cdot (R_i - L_i + 1)$ as $open_i$, and to $a_i - k \cdot (R_i - R_{i-1})$ as add_i. Then the formulas look like $d(i) = d_j + open_i$ and $d(i) = d_{i-1} + add_i$.

We can think about it as a vector (x, y) (in which $x = d(i-1)$, and $y = \max\limits_{j<i} d_j$). The transition from i to $i+1$ is a new vector, which is calculated as $(\max\{y + open_i, x + add_i\}, \max\{y, y + open_i, x + add_i\})$.

You can represent it as a matrix with an unusual multiplication definition $c_{ij} = \max\limits_{k} a_{ik} + b_{kj}$. With these definitions, the transition calculation becomes:

$$\begin{pmatrix} add_i & open_i \\ add_i & \max\{0, open_i\} \end{pmatrix} \oplus \begin{pmatrix} d(i) \\ \max\limits_{j<i} d_j \end{pmatrix} = \begin{pmatrix} d(i+1) \\ \max\limits_{j\le i} d_j \end{pmatrix}$$

Now we can finally solve the problem using a segment tree that stores matrices and their multiplications. For every query, only one leaf of the segment tree changes its value. Thus, the solution works in $\mathcal{O}((n+m)\log n)$.