# EJOI 2019 solutions

August 27, 2019

## 1 Hanging Rack

Let's look on the level 0. First coat should go in the left subtree, second should go in the right subtree, and so on. So we can calculate, in which subtree is $k$-th coat by looking on the parity of number $k$. Now we know, which subtree we need, let's calculate how many coats are in this subtree. Now have the same task for the subtree, do it recursively. We have $n$ layers of recursion, and on each step we solve problem in $O(1)$ time, so the total time will be $O(n)$.

## 2 XORanges

Since $x \oplus x = 0$, after simplification of the final $\oplus$-sum each element will be presented at most once. To determine if the element is presented in the final $\oplus$-sum, we need to calculate the number of segments it belongs to. If this number is odd, then the element will be in the final $\oplus$-sum.

The element $i$ of segment $[l, r]$ belongs to $(i - l + 1)(r - i + 1)$ subsegments. This number is odd if both $(i - l + 1)$ and $(r - i + 1)$ are odd. If $l$ and $r$ has different parity, then it is impossible, so the final $\oplus$-sum will be 0, else $(i - l + 1)(r - i + 1)$ will be odd for $i = l, l + 2, l + 4, \ldots, r$. Now we need data structure to calculate sums like $x_l \oplus x_{l+2} \oplus x_{l+4} \oplus \ldots \oplus x_r$. We can use two segment trees (or Fenwick trees), one for elements with odd indices, another for element with even indices. So, the queries can be answered in $O(\log n)$ time.

## 3 T-Covering

Let's look on some small examples first. If we have one special cell, and we can rotate the tetramino in all 4 directions. Then the answer is sum all neighbouring cells except the cell with minimal value.

If two special cells share a side, then there is unique way to rotate the tetraminos. If two special cells share a corner, then there are two ways to rotate the tetraminos, but the set of covered cells is the same.

If two special cells are 1 cell away from each other, then we have 7 neighbour cells, and we need to cover 6 of them. It's easy to see that we can cover any 6 of

them, so if we need to maximize sum, we need to cover all except the minimal one.

Now let's look on the full task. Let's build a graph. The vertices are the special cells, and two special cells are connected if they share side, or corner, or if they are 1 cell away. Let's find connected components in this graph. If connected component contains $k$ special cells, then there are no more than $4n - (n-1) = 3n + 1$ neighbour cells, that can be covered by the tetraminos, and we need to cover $3n$ of them. So ow we have three cases:

1. If this number is less than $3n$ than it is obviously impossible to put all tetraminos.
2. If this number is $3n$ than it can be shown that it is always possible to place all tetraminos. So we can just add their sum to the answer.
3. If this number is $3n + 1$ than it can be shown that for each neighbour cell it is always possible to place all tetraminos in such a way that all cells except selected one are covered. So we can just add the sum of all elements except the minimal one to the answer.

Total time complexity of this solution is $O(mn)$.

# 4 Awesome Arrowland Adventure

Let's change the task in the following way. Imagine that students go from cell $(0, 0)$ to cell $(m-1, n-1)$, and they rotate the arrows to match the direction of their movement. It's easy to see that in the optimal path they should not visit the same cell more than once, so we can build the following directed graph: vertices of the graph are the cells, and we add edge from vertex $v$ to vertex $u$ of weight $w$, if after $w$ rotations, the arrow in cell $v$ is directed to cell $u$.

Now we can simply find the shortest path in this graph. We can use Dijkstra algorithm with priority queue, or even breath-first search, because all weights are small integers. Time complexity will be $O(mn \log mn)$ or $O(mn)$.

# 5 Tower

First, let's try to build maximal possible number on each step. We will have tower like this: $1, 1, 2, 4, 8, 16, \ldots$. On step $i$ the maximal number we can produce is $2^{i-1}$. So it's obvious that number $x$ cannot be produced in less than $\lceil \log_2 n \rceil + 1$ operations. We will show later that this lower bound is actually achievable. But first let's discuss partial solution that make about twice more operations.

Let's split given number $x$ into sum of powers of two. Now, first spend about $\log_2 n$ operations to make all required powers of two, then spend another $\log_2 n$ operations to copy the required powers of two to the end of the list, and finally make operation that calculates sum of these powers of two.

Now let's discuss the optimal solution. First, let's build the following sequence $1, 1, 2, 4, 8, 16, \ldots, 2^k$, where $2^k$ is the smallest power of two such that $2^k \geq x$. Now let's try to modify this sequence. We can decrease any element of the sequence by one by shifting the left border of the segment from 1 to 2. If we decrease $i$-th element by 1, then $i+1$-th element will be decreased by 1, $i+2$-th by 2, $i+3$-th by 4, and so on. So the $k$-th element will be decreased by $2^{k-i-1}$. Now let's split the difference $d = 2^k - x$ into the sum of powers of two, and decrease the corresponding elements of the sequence to make the last elements equal to $x$.

For example, if $x = 10$, then first we make sequence $1, 1, 2, 4, 8, 16$. Now we need to decrease the last number by $6 = 4 + 2$. First, let's decrease it by 4. This can be done by decreasing the second number from 2 to 1: $1, 1, 1, 3, 6, 12$. Now we need to decrease it by 2. This can be done by decreasing the third number from 3 to 2: $1, 1, 1, 2, 5, 10$.

This solution uses exactly $\lceil \log_2 n \rceil + 1$ operations and it's time complexity is $O(\log n)$.

# 6 Colouring a rectangle

Let's look on some diagonal of the first ($\searrow$) type. If we don't paint this diagonal, we are forced to paint all diagonals of the second ($\nearrow$) type, that intersect our diagonal. Notice that these diagonals always form a continuous segment in the list of diagonals.

Now the task can be modified in the following way. There are $n + m - 1$ elements $a_i$ (corresponding to diagonals of the first type), and $n+m-1$ elements $b_j$ (corresponding to diagonals of the second type). For each $i$ we need either to take element $a_i$, or take all elements $b_j$ for all $j$ in $[l_i, r_i]$ (values $l_i$ and $r_i$ can be precalculated based on $n$ and $m$). Find the minimal cost of taken elements.

This task can be solved by dynamic programming. First, let's sort elements by $l_i$. Now let's say $d[i, j]$ is the minimal cost to satisfy elements $a[0..i-1]$ in such a way that elements $b[l_i..j]$ are taken. The transition is: either take the element $a[i]$, or all elements $b[j..r_i]$. This solution works in $O((n+m)^2)$ time. It can be improved to $O((n+m)\log(n+m))$ by using the segment tree.