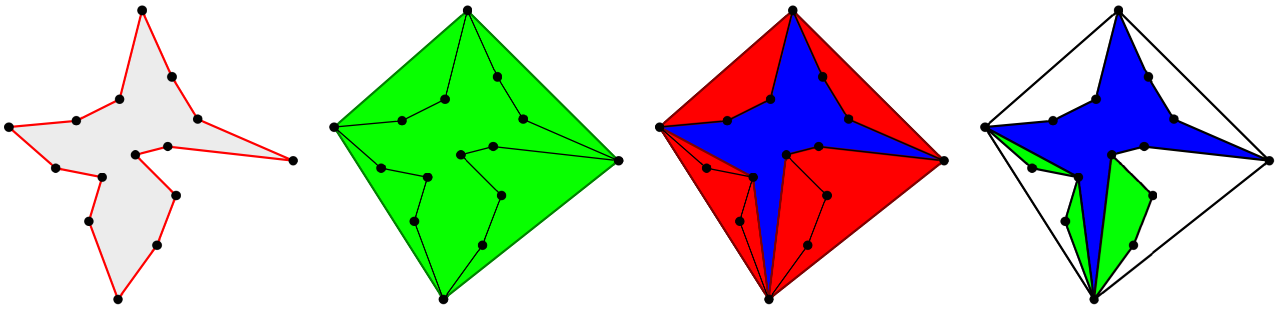
Problem Tutorial: "Alien Invasion"

We describe a recursive solution to this problem. Get area S of the convex hull of all n vertices. Next, find out what vertices c_i ($c_1 < \cdots < c_k$) make up the convex hull of the polygon. To do so, for each vertex i, make a query about all vertices of the polygon except the i-th one and get area S_i. It is easy to see that vertex i is in the convex hull iff $S_i < S$.

Now we have to subtract the areas of the polygons P_1 with vertices in $[c_1, c_2]$, P_2 with vertices in $[c_2, c_3]$, …, P_k with vertices in $[c_k, n] \cup [1, c_1]$. To compute these areas, apply the described algorithm to all these polygons, but with the following observations. If we already know that some vertices c_k and c_{k+1} are in the convex hull, then we do not need to check them once again because they are in the convex hull of polygon P_k, too. So, for each "inner" polygon two extra queries are saved.



If the polygon has only 3 vertices, it is convex, so we just ask for its area by one query and do nothing else. Some vertices c_k and c_{k+1} of the convex hull may be adjacent, in that case the polygon P_k is degenerate and its area is zero. Also, be careful about polygons containing more than two adjacent points on a straight line. Depending on the implementation, the recursive algorithm may get stuck on a degenerate polygon with all its vertices lying on a single line. To avoid such situations, return immediately if the area of a polygon is zero.

It can be proved by induction that the algorithm needs no more than $\dfrac{n(n-1)}{2}$ queries to get the area of the polygon. The upper bound is reached at the polygon which is constructed as follows. Pick a triangle $v_1v_2v_3$, add a new vertex v_4 strictly inside the triangle, connect it to vertices v_2 and v_3 and remove side (v_2, v_3). Then add another vertex v_5 inside triangle $v_2v_3v_4$, connect it to v_3 and v_4 and remove side (v_3, v_4). Proceed with adding new vertices this way until the polygon has n vertices. On each step of the recursive solving algorithm, the number of vertices decreases by 1, so the total number of queries is $(n+1) + (n-2) + (n-3) + \ldots + 3 + 1 = \dfrac{n(n-1)}{2}$.

Problem Tutorial: "Bus Stop"

Denote the time until a bus of route i arrives by t_i. Then t_i is a continuous random variable which is uniformly distributed on the interval $[0; d_i]$. The cumulative distribution function of t_i is

$$P(t_i \le x) = \begin{cases} 0, & x < 0, \\ \dfrac{x}{d_i}, & 0 \le x \le d_i, \\ 1, & x > d_i. \end{cases}$$

If $t = \min\limits_{i=1}^{n} t_i$ (the time until a bus of any route arrives) and $d = \min\limits_{i=1}^{n} d_i$ (the latest time until some bus certainly arrives), then $P(t \le x) = 1 - P(t > x) = 1 - \prod\limits_{i=1}^{n} (P(t_i > x)) = 1 - \prod\limits_{i=1}^{n} (1 - P(t_i \le x))$. Therefore,

$$F_t(x) = P(t \le x) = \begin{cases} 0, & x < 0, \\ 1 - \prod\limits_{i=1}^{n} \left(1 - \dfrac{x}{d_i}\right), & 0 \le x \le d, \\ 1, & x > d. \end{cases}$$

$F_t(x)$ is a polynomial of degree n when $x \in [0, d]$: $F_t(x) = \sum_{i=0}^{n} a_i x^i$. The expected value of the random variable t is $\mathbb{E}\{t\} = \int_0^d x F_t'(x) dx = \sum_{i=0}^{n} \frac{i a_i}{i+1} d^{i+1}$, which is computable in $\mathcal{O}(n)$ time, once we know the coefficients a_i. We can calculate them using divide-and-conquer technique. If number-theoretic transform (NTT) modulo 998 244 353 is used for multiplication of two polynomials, then the total running time will be $T(n) = \mathcal{O}(n \log n) + 2T\left(\frac{n}{2}\right) = \mathcal{O}(n \log^2 n)$.

Problem Tutorial: “Calculating Average”

For problems with average value as an answer function, it is common to use binary search for the answer. Here, we need to find many answers, so “parallel” binary search is required.

Now we need to show how to write a check function for the search.

Let’s fix some v and check if the answer for the v-th element can be greater than or equal to k_v. In other words, we need to check if there exist l and r such that $1 \le l \le v \le r \le n$ and

$$\frac{\sum_{i=l}^{r} a_i}{r - l + 1} \ge k_v.$$

After transforming the equation, we get:

$$\sum_{i=l}^{r} a_i - k_v(r - l + 1) \ge 0,$$

$$\sum_{i=1}^{n} a_i - k_v n - \left(\sum_{i=1}^{l-1} a_i - k_v(l-1)\right) - \left(\sum_{i=r+1}^{n} a_i - k_v(n - r)\right) \ge 0.$$

Let $f_i(x) = \sum_{j=1}^{i-1} a_i - x(i-1)$ and $g_i(x) = \sum_{j=i+1}^{n} a_j - x(n-i)$. Then the expression above can be transformed:

$$f_{n+1}(k_v) - f_l(k_v) - g_r(k_v) \ge 0.$$

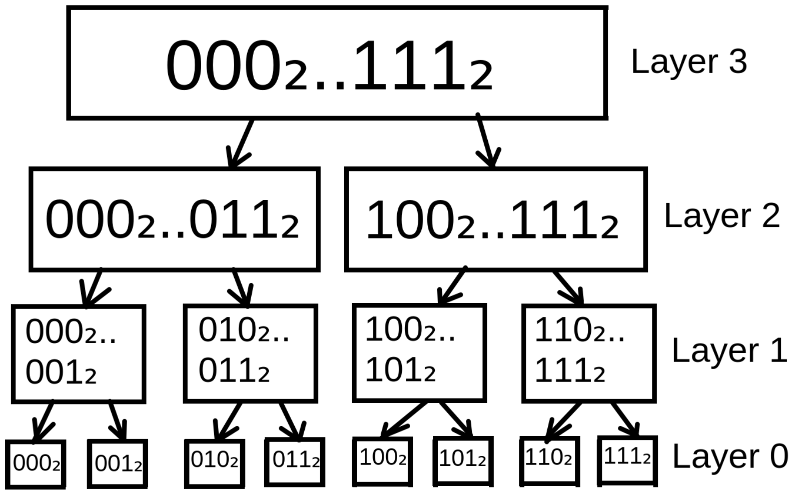
Now we need to maximize $f_{n+1}(k_v) - f_l(k_v) - g_r(k_v)$ and check if this maximum value is not less than zero. Since the first part is a constant for fixed k_v, and the second and the third parts are independent, we need to minimize $f_l(k_v)$ and $g_r(k_v)$ (do not forget that $l \le v$ and $v \le r$ must apply).

It is easy to do it in $\mathcal{O}(n)$, but we need to check the condition above for all the elements. Since $f_i(x)$ and $g_i(x)$ are linear functions, we can use convex hull trick to do it in $\mathcal{O}(\log n)$ for one element, thus the check function is $\mathcal{O}(n \log n)$.

The overall complexity is $\mathcal{O}(n \log n \log C)$.

Problem Tutorial: “Data Structure Problem”

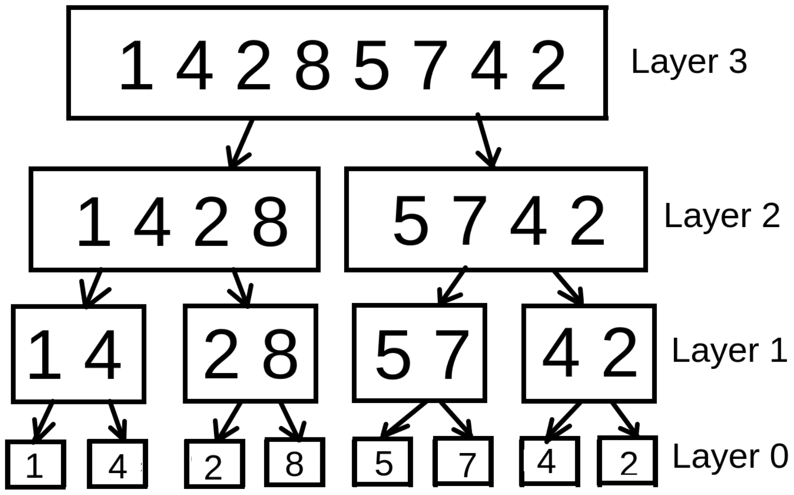
Let’s build a segment tree. You can see its structure on the picture below, for convenience segment bounds are numbered in binary base:



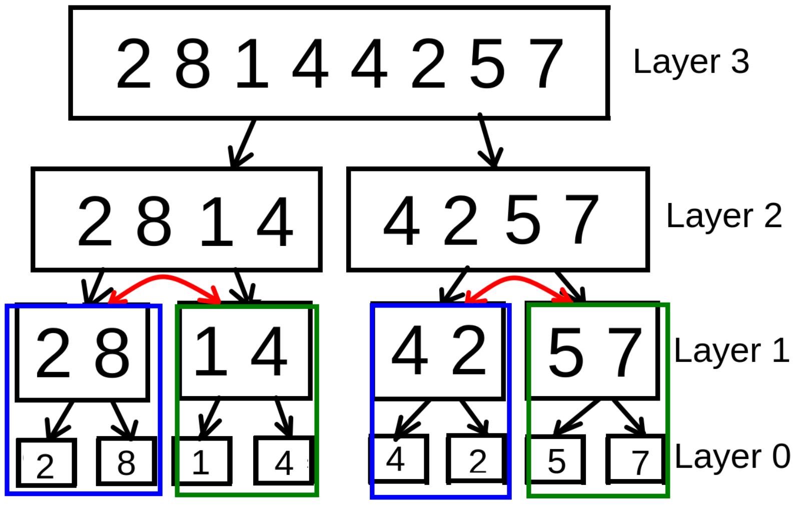
The first two types of queries are trivial, let's learn how to process the last three types.

It is easy to see that the tree is divided into layers. Each node on the i-th layer contains all the items with fixed prefix and any combination of the last i bits. So, it is "friendly" for bit operations on the array indices.

First, learn to answer `xor` queries. We can perform it for each bit independently. Consider the following example:



Suppose we have a query `xor` $2^2 = 4$. We can see that we have to swap the left and the right children for all the nodes on the second layer:

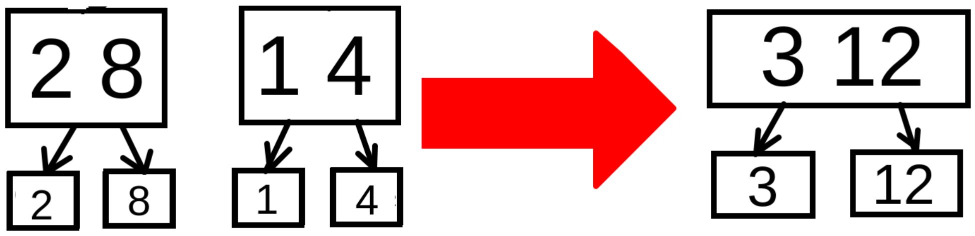


But swapping the children naively still requires $\mathcal{O}(2^p)$ time, which is too slow. To handle it efficiently, we can store a flag for each layer which indicates whether the left and right children are swapped on this layer.

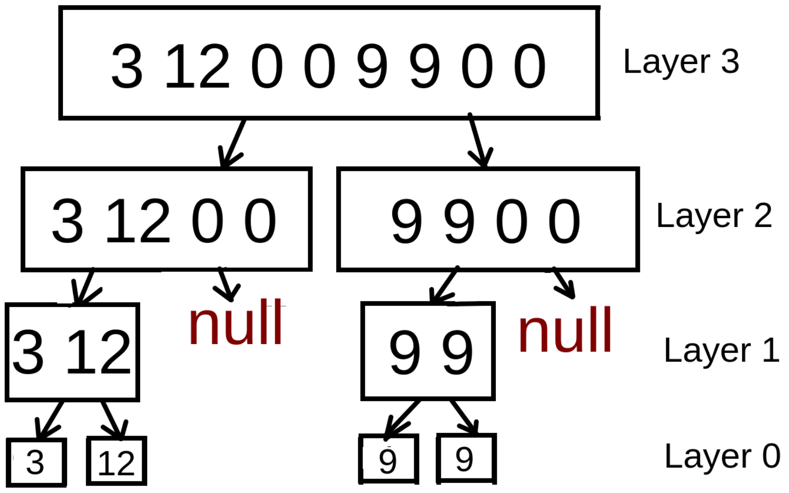
So, we managed to solve the problem for `xor` queries. But we also have `and` and `or` queries. These queries are also performed independently for each bit.

Suppose we have a query `and` $2^2 = 4$. Here, we need to merge the right child into the left child, for all the nodes on the second layer. The right children are removed (consider they are filled with zeroes).

Merging subtrees is just summing their corresponding items. The process looks like this:



So, after we perform the `and` query, we get the following tree:



`Or` queries are processed in a similar way, but we merge the left child to the right child.

It may seem that merging trees is quite slow. But a proper implementation of this algorithm gives about $\mathcal{O}(w)$ merges, where w is the total number of edges added during the entire algorithm. Note that $w = \mathcal{O}(2^p + pq)$, because `add` query can add extra p nodes into the tree.

Some notes about the efficient implementation:

1. Write the segment tree using pointers to the left and right children.
2. Merge the subtrees of size a and b in $\mathcal{O}(\min(a, b))$ time.
3. Try to prevent cases when you merge two empty subtrees, or merge an empty subtree into a non-empty one.
4. The good strategy is to always merge the right child into the left one (as if there are no swap flags) and update the swap flag if you need to do the opposite. So, each node can move to the another tree on each layer no more than once, thus it moves no more than p times in total.

This gives the time complexity $\mathcal{O}(2^p + p^2q)$. But on practice, it works in $\mathcal{O}(2^p + pq)$, because it is hard to construct the worst case (where many nodes "travel" from one tree to another $\mathcal{O}(p)$ times, and $\mathcal{O}(pq)$ nodes are added).

Open question: what is the time complexity, when we have `add on segment` or `assign on segment` queries?

Problem Tutorial: "Expected Cost"

Solution 1 (slow and hard). Suppose we have fixed a tree. We need to find this sum: $\min\limits_{i=1}^{n} \sum\limits_{j=1}^{n} dist(i,j)$.

It is easy to see that vertex i is one of the centroids of the tree. If n is odd, there is only one centroid in the tree. If n is even, there can be two centroids. Suppose we have only one centroid, then we can write a dynamic programming $f[n][t]$ and $g[n][t]$. $g[n][t]$ means number of rooted trees ($t = 0$) or forests ($t = 1$) on n vertices. Each tree in a forest is also rooted. Another restriction is that the size of each subtree does not exceed $\lfloor \frac{n}{2} \rfloor$. $f[n][t]$ means the sum of depths of all n vertices in all trees ($t = 0$) or forests ($t = 1$). Then the answer is $f[n][0] \cdot g[n][0]^{-1}$. These values are calculated as follows:

$g[n][1] = \sum_{j=1}^{min(\lfloor \frac{n}{2} \rfloor, n-1)} g[j][0] \cdot g[n-j][1] \cdot \binom{n-2}{j-1}$

$g[n][0] = n \cdot \sum_{j=1}^{min(\lfloor \frac{n}{2} \rfloor, n-1)} g[j][0] \cdot g[n-j][1] \cdot \binom{n-2}{j-1}$

$f[n][1] = \sum_{j=1}^{min(\lfloor \frac{n}{2} \rfloor, n-1)} f[j][0] \cdot g[n-j][1] \cdot \binom{n-2}{j-1} + f[n-j][1] \cdot g[j][0] \cdot \binom{n-2}{j-1}$

$f[n][0] = (n-1) \cdot g[n][0] + n \cdot \sum_{j=1}^{min(\lfloor \frac{n}{2} \rfloor, n-1)} (f[j][0] \cdot g[n-j][1] \cdot \binom{n-2}{j-1} + f[n-j][1] \cdot g[j][0] \cdot \binom{n-2}{j-1})$

If n is even, we need to subtract trees with two centroids. If there are two centroids in a tree, then we have an edge that divides our tree into two components $\lfloor \frac{n}{2} \rfloor$ size each. So, we subtract trees with two centroids. The resulting sum of depths in all labeled trees is $f[n][0] - f[\frac{n}{2}][0] \cdot \binom{n}{\frac{n}{2}} \cdot g[\frac{n}{2}][0] - \frac{n}{4} \cdot \binom{n}{\frac{n}{2}} \cdot g[\frac{n}{2}][0]^2$.

The time complexity is $\mathcal{O}(n^2)$.

Solution 2 (fast and easy). For arbitrary edge, let's assume it splits the tree into two components with sizes k and $n-k$. Each edge adds $\min\{k, n-k\}$ to the answer. There are $\binom{n-2}{k-1}$ ways to choose vertices for one part of the tree and k^{k-2} and $(n-k)^{n-k-2}$ possible subtrees from each side of the edge. So, the total answer will be $\frac{1}{n^{n-2}} \cdot \frac{n(n-1)}{2} \sum\limits_{k=1}^{n} \min\{k, n-k\} \cdot \binom{n-2}{k-1} k^{k-2} (n-k)^{n-k-2}$.

The time complexity is $\mathcal{O}(n \log n)$.

Problem Tutorial: "Fractional XOR Maximization"

Initially we have two conditions for XOR operands: $x \in [a,b]$ and $y \in [c,d]$. We will consider bits of the answer from the greatest (which is 2^{56}) to the lowest, and we will change bounds of segments $[a,b]$ and $[c,d]$ in order to obtain highest possible value.

Suppose we are currently at bit 2^i. We will maintain a, b, c, d in such a way that a, b, c, d will have zeros in all bits greater than 2^i. Let's write a bit string $wxyz$, where w and x are lowest and highest possible values of current bit on a segment $[a,b]$, and y and z are lowest and highest possible values of current bit on a segment $[c,d]$. Obviously, $0 \le w \le x \le 1$ and $0 \le y \le z \le 1$. Then we change a, b, c, d to their new values a', b', c', d' and set bits in the answer value as described in the following table:

Bit string	XOR bit	a'	b'	c'	d'	Comment
0000	0	a	b	c	d	No choice
0011	1	a	b	c	d	No choice
1100	1	a	b	c	d	No choice
1111	0	a	b	c	d	No choice
0001	1	a	b	$f_i(d,0)$	d	Pick subsegment with bit 1 from $[c,d]$
1101	1	a	b	c	$f_i(c,1)$	Pick subsegment with bit 0 from $[c,d]$
0100	1	$f_i(b,0)$	b	c	d	Pick subsegment with bit 1 from $[a,b]$
0111	1	a	$f_i(a,1)$	c	d	Pick subsegment with bit 0 from $[a,b]$
0101	2*					Breaking from cycle

In the table above, $f_i(x,z)$ denotes the number that we obtain from x changing all bit values in bits lower than 2^i to z.

While the other cases are processed quite obviously with picking the bit value that maximizes the answer, the case 0101 is special. As we can obtain both 0 and 1 in the current bit from both segments $[a,b]$ and $[c,d]$, the number 2^i which has bit 2^i set to 1 and all other bits set to 0 can be obtained from both segments. Moreover, the number $2^i - 2^{-p}$ which has only bits $2^{i-1}, \ldots, 2^{-p}$ set to 1 can also be obtained from both segments if p is a large enough integer. Taking XOR value of these two numbers results in number that has bits $2^i, 2^{i-1}, \ldots, 2^{-p}$ set to 1 and equals $2^{i+1} - 2^{-p}$, and this number can have values that are arbitrarily close to 2^{i+1} but do not reach 2^{i+1}. So we should add $2 \cdot 2^i$ to the answer and stop the process.

If our case is not 0101, after the transformation described by the table we change value of bit 2^i in numbers a, b, c, d to 0 since it is useless after this step.

To deal with assignments to infinite number of bits, we can maintain four boolean values f_a, f_b, f_c, f_d which will indicate that all bits that are less than or equal to current bit 2^i in numbers a, b, c, d are changed to 0, 1, 0, 1, respectively. Initially these values are false, and the respective value becomes true if the respective number changes its bits according to table above.

Obviously the process described above can be infinite. Let's denote by $r(x)$ the length of repetend in binary representation for a rational number x (if the binary representation of x contains a finite number of ones, we consider $r(x)$ to be equal to 1 and repetend to be 0). If we currently have segments $[a,b]$ and $[c,d]$ and $r(a)$, $r(b)$, $r(c)$, $r(d)$ do not change, then bit values of all four numbers repeat each $L = lcm\,(r(a), r(b), r(c), r(d))$ bits if we consider only bits not greater than 2^{-5}. This is true since the first occurence of repetend in binary representation (i.e. the occurrence that covers the greatest bits) of each fraction with denominator not greater than 30 cannot start later than at bit 2^{-5} (since $30 < 2^5$). Note that $r(a)$, $r(b)$, $r(c)$ and $r(d)$ are known in advance and can change only to 1 in case if corresponding value f_i becomes true.

We will do the above process until we proceed to bit 2^{-5} (or stop earlier). Then, if we haven't stopped yet, we continue the same process maintaining the number $L = lcm\,(r(a), r(b), r(c), r(d))$ and the number $lastChange$ of bits we have passed since the last time any of f_a, f_b, f_c, f_d became true. If at any moment $lastChange = L$, the repetend consisting of lowest L obtained digits of XOR value will appear if we continue the process, so we can stop and find the final answer adding the value of repetend divided by $2^L - 1$ to obtained XOR value. Note that since there are only 4 variables f_i and $lcm\,(r(a), r(b), r(c), r(d)) \le 13860$ for all fractions a, b, c, d with denominator up to 30, we can make at most $4 \cdot 13860 + 56 + 5 = 55501$ steps of the algorithm for a single test before we stop.

When denominators are up to 30, it turns out that:

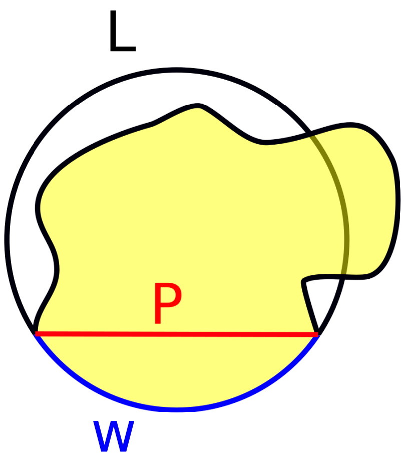
- the greatest number of bits we can go through is 369;
- the greatest possible denominator of the answer is approximately $4.041 \cdot 10^{43}$ which does not fit into `__int128` type, so usage of long arithmetics is required.

Also, both these values attain the greatest value when $a = b = 1/29$ and $c = d = 1/23$ (but not only in this case).

Problem Tutorial: "Go West"

It is clear that it is optimal to use the whole length of the rope because extra length gains more area to the answer. Also, the region is optimal only if it is convex. Hence, we need to attach both ends of the rope to the stick to get maximum.

Let P $(0 \le P \le \min\{K, L\})$ be the length of the used part of the stick that borders with the region, and the rope is tied to the ends of this part. It is claimed that if the rope makes an arc of a circle, then the answer is optimal.



Indeed, suppose that there exists some greater answer to this problem. Complete the arc of length L to a full circle with an additional arc of length W from the other side, so now we consider the area of a closed figure with its perimeter $L + W$. The isoperimetric inequality states that in $\mathbb{R}^2$, for the length L of the curve and the area A of the planar region enclosed by the curve, $4\pi A \le L^2$ is true and the equality holds iff the curve is a circle. So, if the optimal answer hadn't been the arc of a circle, we would have got a figure (its "lower" part does not change) that is not a circle. Thus, it would have had less area. Contradiction.

With fixed L, the area $S = \frac{L^2}{2\alpha^2}(\alpha - \sin\alpha)$, where α is the central angle based on the arc with its length L. After transformations, the following equality holds: $\frac{2L^2}{\alpha^2}(1 - \cos\alpha) = P$. It can be shown (by taking derivatives) that the function in the left-hand side of the equation decreases on $\alpha \in (0, 2\pi)$, so we can use binary search to find α if L and P are fixed.

We can prove (by taking derivatives again) that $\frac{\alpha - \sin\alpha}{\alpha^2}$ increases on $\alpha \in (0, \pi]$ and decreases on $\alpha \in [\pi, 2\pi]$. Therefore, the maximum area is reached at $\alpha = \pi$. Finally, we conclude that if $L \le K \cdot \frac{\pi}{2}$, then area S is maximum if $P = \frac{2L}{\pi}$, otherwise the area is maximum if $P = K$.

Problem Tutorial: "Humongous String"

If $k = 1$, then obviously the answer is equal to n.

Consider the infinite string $U = T_0T_1T_2\ldots$ We will name all strings T_i *blocks*, and if the block T_i ends with character s_t (or, equivalently, $i \bmod k = t$), we will call it a *t-block*. Consider two neighboring blocks T_iT_{i+1}. If T_i is a $(k-1)$-block (and T_{i+1} is a 0-block), we will call the concatenation of these two blocks a *big block*, otherwise we will say there is a *border* (denoted by '|') between these blocks. We will also call all t-blocks with $1 \le t \le k-2$ *regular blocks*. By construction neighboring characters s_i and s_j are not separated with border iff $j = (i+1) \bmod k$.

Now, if $k \ge 3$, U looks like $T_0|\ldots|T_{k-2}|T_{k-1}T_k|T_{k+1}|T_{k+2}|\ldots|T_{2k-2}|T_{2k-1}T_{2k}|T_{2k+1}|T_{2k+2}|\ldots$,

and if $k = 2$, $U = T_0|T_1T_2|T_3T_4|T_5T_6|T_7T_8|\ldots$.

Note that if a finite substring of U has at least two borders inside it, it cannot have more than one occurrence in U, since all regular and big blocks have different lengths. Also, note that for each block B with size greater than $k+1$ the largest block to the left of B is the previous big block for B. Lengths of consecutive a-blocks differ by k (if $1 \le a < k-1$), and lengths of consecutive big blocks differ by $2k$.

Let's calculate for each block B (regular or big) the value $old(B,i)$ indicating how many substrings ending in the i-th character of B have occurrences before this character (on a prefix of U made up of all characters up to the $(i-1)$-th character of B). Then the answer is $\dfrac{n(n+1)}{2}$ minus the sum of $old(B,i)$ over the first n characters of U.

Let b be the length of the first subblock of B. We claim that if $k \ge 3$, then

$$old(B,i) = \begin{cases} i, & b \le k,\ i < b; \\ 0, & b \le k,\ i = b; \\ i, & b = k+1; \\ i+b-k-1, & B \text{ is big},\ 2b+1-i \ge 2k; \\ i-k, & B \text{ is big},\ 2b+1-i < 2k; \\ i+2b-3-2k, & b \mod k = 2,\ b-i \ge k; \\ i, & b \mod k = 2,\ b-i < k; \\ i+b-k-1, & b \mod k > 2,\ b-i \ge k; \\ i, & b \mod k > 2,\ b-i < k. \end{cases} \quad (1)$$

and if $k = 2$, then

$$old(B,i) = \begin{cases} i+2b+1-4k, & b \ge 4, 2b+1-i \ge 2k; \\ i-k, & b \ge 4, 2b+1-i < 2k; \\ |i-1|, & b = 2; \\ 1, & b = 1. \end{cases} \quad (2)$$

Proof.

Denote the prefix of S that ends with the i-th character of B by P_{Bi}.

Consider several cases.

If $k \ge 3$:

- If B is one of the first $k+1$ blocks and $i \ne b$ or $i = b = k+1$, then a suffix of P_{Bi} of length i has at least one occurrence starting in the previous block, and all larger suffixes (if they exist) have no previous occurrences since they contain the first occurrence of character s_{b-2}.
- If B is one of the first k blocks and $i = b$, then any suffix of P_{Bi} has no previous occurrences since it contains the first occurrence of character s_{b-1}.
- Otherwise:
  - If B is a regular a-block and $i > b-k$, then a suffix of P_{Bi} of length i has at least one occurrence in the previous big block, and all larger suffixes with one border inside have no previous occurrences, since these occurrences are required to end in a-block (and can fit in no previous a-block).
  - If B is a big block and $i > 2b+1-2k$, then a suffix of P_{Bi} of length $i-k$ has at least one occurrence starting at the first character of B, and all larger suffixes have no previous occurrences, since even suffixes with no borders inside won't fit into any previous block.
  - If B is a regular a-block $(a > 1)$ and $i \le b-k$, then a suffix of P_{Bi} of length i has occurrences in the previous big block and in the previous a-block. If we extend this suffix further to the left, it will have a border inside and thus all its occurrences with one border inside will end in a-block. As the size of regular $(a-1)$-block in which any previous occurrence can start does not exceed $b-1-k$, we can extend the suffix by at most $b-1-k$ characters to the left if we

want it to have previous occurrences. So the largest suffix of P_{Bi} that occurs earlier in the P_{Bi} has length $i + b - 1 - k$.

- If B is a regular 1-block and $i \le b - k$, then the same observations as for other regular blocks are true. The only difference is that a previous occurrence of a suffix of P_{Bi} with one border inside starts in a big block, so the largest suffix of P_{Bi} that occurs earlier in the P_{Bi} has length $i + 2b - 3 - 2k$ since the size of big block in which any previous occurrence can start does not exceed $2b - 3 - 2k$.
- If B is a big block and $i \le 2b + 1 - 2k$, then a suffix of P_{Bi} of length i has occurrences in the previous big block. If we extend this suffix further to the left, it will have a border inside and thus all its occurrences with one border inside will end in $(k-2)$-block. As the size of regular $(k-2)$-block in which any previous occurrence can start does not exceed $b - 1 - k$, we can extend the suffix by at most $b - 1 - k$ characters to the left if we want it to have previous occurrences. So the largest suffix of P_{Bi} that occurs earlier in the P_{Bi} has length $i + b - 1 - k$.

If $k = 2$, the only difference from the $k \ge 3$ case is that there are no regular blocks except the T_0 block; therefore, the only equation that changes is the expression of $old(B, i)$ for $b > 5$ and $i \le 2b + 1 - 2k$, in which $i + b - 1 - k$ changes to $i + 2b + 1 - 4k$ since the block in which any previous occurrence (of suffix with one border inside) can start is the block whose length is $4k$ smaller than the length of B which is $2b + 1$. Note that the value $old(B, i)$ must be calculated separately for a big block with size 5 since it has only one block to the left. This completes the proof of the above huge equations.

How to calculate the sum of $old(B, i)$ over the first n characters of U? From the above equations it is clear that we can split each block into at most two parts such that the summation of $old(B, i)$ over each part is the summation of linear function of i and gives the result of form $AL^2 + BL + C$, where L is the length of part. This result can be calculated using the summation formulas for powers of positive integers up to L. Similarly, we can find out that the sum of $old(B, i)$ over all characters of block B is equal to a polynom $P(b)$, $Q(b)$, or $R(b)$ (that depends on the length of the first subblock of B) if B is an a-block $(1 < a < k - 1)$, a 1-block, or a big block, respectively. Moreover, these polynoms' degrees are at most 2, and we can calculate coefficients of these polynoms using summation formulas for powers of positive integers.

We need to calculate the sum of $old(B, i)$ over all blocks that are fully contained by the prefix of U of length n, and over a block that is not fully contained by this prefix, but has non-empty intersection with this prefix (if such a block exists). In the latter case, we calculate the sum as described in the previous paragraph: split the part of block if it is necessary, and calculate the sum for each part separately. In the former case, we need to calculate the sum

$$C + \sum_{3 \le i \bmod k \le k-1,\ i>k}^{A} P(i) + \sum_{i \bmod k = 2,\ i>k}^{A} Q(i) + \sum_{i \bmod k = k,\ i>k}^{A} R(i)$$

where C is the sum for the first $k+1$ blocks that can be calculated in $\mathcal{O}(1)$ time using summation formulas for powers of positive integers, and A is maximal length of first subblock over all blocks that are fully contained by the prefix of U of length n.

Note that i can take neither of values $k+1, 2k+1, \ldots, pk+1, \ldots$ in the above sums, but we can fix it if we express $R(x)$ as $R(x) = P(x) + P(x+1) + S(x)$ and $Q(x)$ as $Q(x) = P(x) + T(x)$. Then the above sum rewrites as follows:

$$C + \sum_{i=k+2}^{A} P(i) + \sum_{i=1}^{A_1} T(ik+2) + \sum_{i=1}^{A_2} S(ik)$$

where A_1 and A_2 are some constants that can easily be determined.

The first sum can be computed in $\mathcal{O}(1)$ time using summation formulas for powers of positive integers. After some reformulation, the latter two sums can also be computed in the same way since $T(ik+2)$ and $S(ik)$ are still polynoms of variable i of degree 2. Thus, the whole answer can be computed in time $\mathcal{O}(1)$.

Problem Tutorial: "Intellectual Prefix Maxima"

Do centroid decomposition of the tree. For each query u, v find a centroid c that divides the path into two parts from u to centroid c and from the centroid to v. We can precalculate the sum of prefix maxima on the path from u to c using binary lifts. We also precalculate the sum of prefix maxima from centroid c to v. But we need to subtract some prefix of the length l because the maximum x on the path from vertex u to the centroid is bigger than weights of each edge on the prefix, and add $l \cdot x$. We can find such l using binary lifts.

The overall complexity is $\mathcal{O}(n \log^2 n + q \log n)$.

Problem Tutorial: "Jimp Numbers"

Note that a positive integer n is jimp if and only if there exists exactly one pair (x, y) of two positive integers x and y such that $x > y$ and $(x-y)^2 + x^2 + n = (x+y)^2$ (if $y \le 0$, the equation above cannot be satisfied). From this equation we can obtain $n = (4y - x)x$. Hence $n > 0$ is jimp if and only if it has exactly one positive divisor x such that:

- $x + \frac{n}{x} = 4y$ is divisible by 4,
- $y = \dfrac{x + \frac{n}{x}}{4} < x \Rightarrow x > \sqrt{\dfrac{n}{3}}$.

Consider several cases:

- If $n \equiv 1 \mod 4$, then $x + \frac{n}{x} \equiv 2 \mod 4$, so n is not jimp.
- If $n \equiv 2 \mod 4$, then $x + \frac{n}{x}$ is not divisible by 4 since one of two numbers x and $\frac{n}{x}$ is odd and the other one is even.
- If $n \equiv 3 \mod 4$, then $x + \frac{n}{x} \equiv 0 \mod 4$ always holds, so n is jimp if and only if it has no divisors in $\left(\sqrt{\frac{n}{3}}, n\right)$, that is, n is prime.
- If $n = 2^u v$ for some $u \ge 2$ and odd v and $x + \frac{n}{x}$ is divisible by 4, then $x = 2^s t$, where $1 \le s \le u-1$, t divides v and the numbers s and $u - s$ are either both equal to 1 or both not equal to 1. If n is jimp, then exactly one pair $(s, t) = (u-1, v)$ satisfies $2^s t > \sqrt{\frac{n}{3}}$.
  - If $u = 2$ and $v = 1$, n is equal to 4 and is jimp.
  - If $u = 2$, $v > 1$, n is jimp and q is the least prime factor of v, then $\frac{n}{2q} = 2^{u-1}\frac{v}{q} \le \sqrt{\frac{n}{3}} < 2^{u-1}v = \frac{n}{2}$ holds. Hence $3n \le 4q^2$. Let α be the highest degree of q dividing n. Then $12q^\alpha \le 3n \le 4q^2$, and $\alpha = 1$.
    Since $n = 4qz$ for some $z \ge 1$ not divisible by 2 or q, $3n \le 4q^2 \Rightarrow z \le \frac{q}{3} \Rightarrow z = 1$. That means in this case n is jimp if and only if $\frac{n}{4}$ is an odd prime.
  - If $u = 3$, n is not jimp since the numbers x and $\frac{n}{x}$ are either both odd or one is divisible by 4 and the other is divisible by 2 and not by 4.
  - If $u = 4$ and $v = 1$, n is equal to 16 and is jimp.
  - If $u = 4$ (this implies $2 \le s \le u-2$), $v > 1$, n is jimp and q is the least prime factor of v, then $\frac{n}{4q} = 2^{u-2}\frac{v}{q} \le \sqrt{\frac{n}{3}} < 2^{u-2}v = \frac{n}{4}$ holds. Hence $3n \le 16q^2$. Let α be the highest degree of q dividing n. Then $48q^\alpha \le 3n \le 16q^2$, and $\alpha = 1$.

Since $n = 16qz$ for some $z \geq 1$ not divisible by 2 or q, $3n \leq 4q^2 \Rightarrow z \leq \frac{q}{12} \Rightarrow z = 1$. That means in this case n is jimp if and only if $\frac{n}{16}$ is an odd prime.

– If $u \geq 5$ (this implies $2 \leq s \leq u-2$) and n is jimp, then $\frac{n}{8} = 2^{u-3}v \leq \sqrt{\frac{n}{3}} < 2^{u-2}v = \frac{n}{4}$ holds. Hence $n \leq \frac{64}{3}$ and n is divisible by 32, which is impossible.

Denote the number of primes not exceeding n with remainder 1 and 3 modulo 4 by $f_1(n)$ and $f_3(n)$, respectively. Also consider function $f(n) = f_1(n) + f_3(n)$. Then the answer is $f_3(n) + f_1\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + f_3\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + f_1\left(\left\lfloor\frac{n}{16}\right\rfloor\right) + f_3\left(\left\lfloor\frac{n}{16}\right\rfloor\right) + C$ where $C = |\{4, 16\} \cap \{x : x \leq n\}|$.

How to calculate $f_1(n)$ or $f_3(n)$ for fixed n fast enough? There are two intended approaches.

The first (and easier) one is to precalculate the number of primes with remainder 1 and 3 modulo 4 on each block of size $blockSize \approx 2.5 \cdot 10^6$ (there will be $\left\lceil\frac{10^{11}}{blockSize}\right\rceil$ blocks). The precalculated values will easily fit into the code size limit (which is 256 KB in Yandex.Contest). If we need to calculate $f_i(n)$, we can sum up the precalculated values on $\left\lfloor\frac{n}{blockSize}\right\rfloor$ blocks and then use the segmented sieve on a part of the block containing n. However, the precalculation takes some time (about 20 minutes on the authors' PC).

The second approach uses dynamic programming and the sieve of Eratosthenes. It can also be used to solve this problem when $n \leq 10^{12}$; however, in this case its running time and memory consumption increase significantly. The approach is described below.

Consider the process of obtaining all odd primes via the sieve of Eratosthenes. Let p_i be the i-th odd prime. After step i, number 1 and all odd numbers divisible by p_1, p_2, $\ldots$, p_{i-1} or p_i and not equal to p_j for all $1 < j \leq i$ are considered composite. We will call all other odd numbers *semi-prime after step i*. Denote the number of semi-primes not exceeding k with remainder 1 and 3 modulo 4 by $dp_1[k][i]$ and $dp_3[k][i]$, respectively. It's clear that $dp_j[n][i]$ doesn't change if $i \geq \lfloor f\left(\sqrt{n}\right)\rfloor$ and increases, and also that $f_1(n) = dp_1[n][f(\lfloor\sqrt{n}\rfloor)]$ and $f_3(n) = dp_3[n][f(\lfloor\sqrt{n}\rfloor)]$. Also, $dp_j[k][0]$ is equal to the number of numbers of form $4z + j$ not exceeding k except 1, that is, $dp_1[0][0] = dp_3[0][0] = 0$, $dp_1[k][0] = \left\lfloor\frac{k-1}{4}\right\rfloor$ for $k \geq 1$, $dp_3[k][0] = \left\lfloor\frac{k+1}{4}\right\rfloor$ for $k \geq 1$.

How does $dp_1[k][i]$ differ from $dp_1[k][i-1]$? Consider the set S_{1i} of numbers not exceeding k of form $4z+1$ that are semi-prime after step $i-1$ but are not semi-prime after step i. All numbers from S_{1i} have form αp_i, where $\alpha \equiv p_i \mod 4$, $\alpha \leq \left\lfloor\frac{k}{p_i}\right\rfloor$, $\alpha \neq 1$ and α is coprime with all numbers p_1, p_2, $\ldots$, p_{i-1}. Then α is semi-prime after step $i-1$ that is not equal to any of $p_1, p_2, \ldots, p_{i-1}$, so the number of distinct values of α is $dp_x[k'][i-1]$ minus the number of primes p_1, $\ldots$, p_i of form $4z+x$ not exceeding k', where x is the remainder of p_i modulo 4 and $k' = \left\lfloor\frac{k}{p_i}\right\rfloor$. Then $dp_1[k][i] = dp_1[k][i-1] - dp_x[k'][i-1] + f_x(\min\{p_{i-1}, k'\})$.

Considering the case of $dp_3[k][i]$ in the same manner, we can obtain that for $p_i \equiv 1 \mod 4$

$$\begin{cases} dp_1[k][i] = dp_1[k][i-1] - dp_1[k/p_i][i-1] + f_1(\min\{p_{i-1}, k/p_i\}), \\ dp_3[k][i] = dp_3[k][i-1] - dp_3[k/p_i][i-1] + f_3(\min\{p_{i-1}, k/p_i\}), \end{cases}$$

and for $p_i \equiv 3 \mod 4$

$$\begin{cases} dp_1[k][i] = dp_1[k][i-1] - dp_3[k/p_i][i-1] + f_3(\min\{p_{i-1}, k/p_i\}), \\ dp_3[k][i] = dp_3[k][i-1] - dp_1[k/p_i][i-1] + f_1(\min\{p_{i-1}, k/p_i\}), \end{cases}$$

where / is integer division. Note that $\min\{p_{i-1}, k/p_i\} \leq \min\{p_i, k/p_i\} \leq \sqrt{k}$, so to calculate all dp values we only need values $f_j(k)$ for $k \leq \sqrt{n}$, which can be precomputed.

We can calculate $dp_j[n][f(\lfloor\sqrt{n}\rfloor)]$ for any fixed n as follows. We will go from $i = f(\lfloor\sqrt{n}\rfloor)$ to $i = 0$ maintaining the expression for the answer in the form $ans = C + \sum_k (c_{1k} dp_1[k][i] + c_{3k} dp_3[k][i])$, where i is fixed and C is some constant, and we will express terms with $i = q$ through terms with $i = q - 1$ using the formulas above. For example, if we want to calculate $dp_1[k][i]$, initially the expression for the answer is $dp_1[k][i]$, after the next step it will be $1 \cdot dp_1[k][i-1] - 1 \cdot dp_1[k/p_i][i-1] + f_1(\min\{p_{i-1}, k/p_i\})$, and so on. Note that we will only go through such states which the final answer depends on. Eventually we will get an expression of the form $ans = C + \sum_k (c_{1k} dp_1[k][0] + c_{3k} dp_3[k][0])$, where all values $dp_j[k][0]$ are already known.

As all k parameters of visited states have form $\left\lfloor \frac{n}{x} \right\rfloor$, there are at most $\mathcal{O}(\sqrt{n})$ possible values of k: values from 0 to $\lfloor\sqrt{n}\rfloor$ and values $\left\lfloor \frac{n}{x} \right\rfloor$, where $1 \le x \le \sqrt{n}$. So there are at most $\mathcal{O}(\sqrt{n} f(\lfloor\sqrt{n}\rfloor)) = \mathcal{O}\left(\frac{n}{\log n}\right)$, since it is known that $f(x) = \mathcal{O}\left(\frac{x}{\log x}\right)$. Hence the current complexity is $\mathcal{O}\left(\frac{n}{\log n}\right)$.

Note that if we need values $dp_j[k][i]$ for all $j \in \{1, 3\}$ for s states with $k \le t$, we can calculate these values using Fenwick tree and sieve of Eratosthenes in time $\mathcal{O}(s \log t + t \log\log t + t \log t) = \mathcal{O}((s + t) \log t)$.

Let's choose a constant t and modify the above process of getting the answer as follows: we will express values of $dp_j[k][i]$ through dp values with lesser i only if $k \ge t$, otherwise we will save this state and calculate its values later using Fenwick tree and sieve of Eratosthenes (which will be used for all such states with $k < t$).

How many states with $k \ge t$ will we visit? If $p_i > \frac{n}{t}$ (or, equivalently, $i > f\left(\frac{n}{t}\right)$), there can be at most one state with this fixed i and $k \ge t$, since there is initially one, and it creates one state with the same k (equal to n) and one state with $k = n/p_i < t$. For all other fixed i, there can be at most $\frac{n}{t}$ states with $k \ge t$, since k always has the form $\left\lfloor \frac{n}{x} \right\rfloor$. In total, we can visit $\mathcal{O}\left(f(\sqrt{n}) - f\left(\frac{n}{t}\right) + f\left(\frac{n}{t}\right) \cdot \frac{n}{t}\right) = \mathcal{O}\left(\frac{\sqrt{n}}{\log n} + \frac{n^2}{t^2 \log \frac{n}{t}}\right)$ states with $k \ge t$. The same estimation applies for the number of states with $k < t$, since we can go to such state only from a state with $k \ge t$, and from a state with $k \ge t$ we can go to at most one state with $k < t$.

As the states with $k \ge t$ are processed in time $\mathcal{O}(1)$, the total time complexity is

$$\mathcal{O}\left(\left(\frac{\sqrt{n}}{\log n} + \frac{n^2}{t^2 \log \frac{n}{t}} + t\right) \log t\right)$$

If we take $t = n^{0.64}$, the complexity becomes

$$\mathcal{O}\left(n^{0.5} + n^{0.72} + n^{0.64} \log n\right) = \mathcal{O}\left(n^{0.72} + n^{0.64} \log n\right).$$

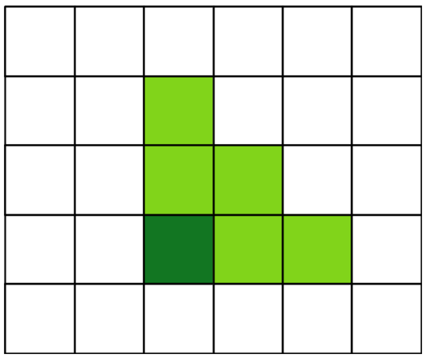
However, in practice the lowest time is attained when $t = n^{0.6}$.

Also note that all five terms $f_3(n)$, $f_1\left(\left\lfloor \frac{n}{4} \right\rfloor\right)$, $f_3\left(\left\lfloor \frac{n}{4} \right\rfloor\right)$, $f_1\left(\left\lfloor \frac{n}{16} \right\rfloor\right)$, $f_3\left(\left\lfloor \frac{n}{16} \right\rfloor\right)$ needed for the answer can be calculated using the same dynamic programming and not the five different.

Note that for states $dp_j[k][i]$ which satisfy $k < t$ and $f\left(\sqrt{k}\right) < i$, $dp_j[k][i] = f_j(k)$, so if we compute these dp values without Fenwick tree and memorizing we can reduce the memory consumption significantly. However, the optimal t values for minimization of running time and for minimization of memory differ.

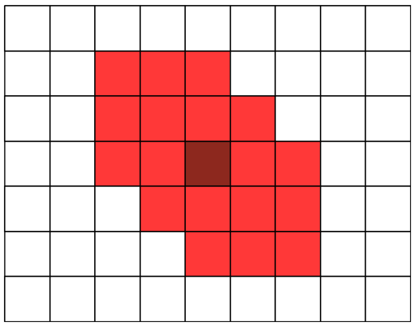
Problem Tutorial: "K-Triangles"

There are several solutions to this problem, we describe only one of them. We need to consider all the 16 cases of the rotation of the triangles. Here, we consider both triangles rotated as shown on the picture below, other cases can be handled in a similar way.



Let's call the dark green cell a *pivot cell* for the given triangle. For the *pivot cell*, we need to calculate the sum on the corresponding triangle. To do this, we notice that, when a triangle shifts one cell right, a vertical line is removed and a diagonal line is added. So, if we precalculate prefix sums on vertical, horizontal and diagonal lines, we can "shift" a triangle in $\mathcal{O}(1)$. And when the triangle shifts one cell down, a diagonal line is removed and a horizontal line is added.

So, for each pivot cell, we know the sum on the corresponding triangle. Now iterate over all the possible positions of the first triangle. From here, some of the cells are forbidden as pivot cells for the second triangle. The configuration of such cells looks like this:



It is easy to notice that the following subset of cells is allowed:

- Some prefix and suffix of columns.
- Some prefix and suffix of rows.
- Some prefix and suffix of diagonals.

So we can precalculate prefix and suffix maxima on the vertical, horizontal and diagonal lines (there are two kinds of diagonal lines, by the way). Now we can take some of the prefix and suffix maximums and calculate for each triangle the feasible triangle with the biggest sum, in $\mathcal{O}(1)$. Thus, the time complexity is $\mathcal{O}(nm)$, with large constant factor.

Problem Tutorial: "Long Game"

If the permutation is equal to $(1, 2, \ldots, n)$, then Alice immediately loses. Otherwise let's look at the game in the reversed order. We can see that a losing state of strips looks like this: p_1, p_2, …, $p_i p_{i+1}$, …, p_n, where $p_i > p_{i+1}$. If it does not match this pattern, then the current player has a correct move. So, we have a fixed number of moves in the game which is equal to $n - 2$, thus the answer depends only on parity of n: if n is odd, Alice wins, otherwise Bob wins.