# Problem Tutorial: "A. Belarusian State University"

Let's define a convolution in some generalized form similar to FFT or XOR-transform.

We'll use 0-based indexing below. Denote the set of functions that accept two integer arguments between 0 and $k-1$ and return an integer between 0 and $k-1$ as $F_k$. Let's call a triple of square matrices $(A, B, C)$ with dimensions $k \times k$ a *convolution triple* for a function $f \in F_k$ if $C$ is nondegenerate and for any two vectors $x$ and $y$ of size $k$, the following process:

- Define vectors $p = Ax$ and $q = By$;
- Define $r$ as component-wise product of $p$ and $q$ (i. e. $r_i = p_i q_i$ for $0 \le i < k$);
- Return $z = C^{-1} r$.

returns the $f$-convolution, i. e. $z_s = \sum\limits_{f(i,j)=s} x_i y_j$.

Now we want to find a convolution triple for a given bitwise-independent function such that the convolution process is fast enough.

How to check whether three matrices form a convolution triple for a certain function $f \in F_k$? The necessary condition for that is: the convolution should be correct for all vectors with all elements but one being zeroes. If $x$ has the only nonzero element $x_i = 1$, and $y$ has the only nonzero element $y_j = 1$, then $z$ should have only one nonzero element $z_{f(i,j)} = 1$. These conditions are also sufficient because of the linearity of the convolution.

If $A = \{a_{ij}\}$, $B = \{b_{ij}\}$, $C = \{c_{ij}\}$, the above condition can be rewritten as: $C$ must be nondegenerate, and for all $i, j, s \in \{0, 1, \ldots, k-1\}$,

$$a_{si} b_{sj} = c_{s,f(i,j)}$$

should be satisfied.

Now we can try to solve the case $n = 1$. One of the ways to find the convolution triples for any binary function is to try all triples of matrices with elements in $\{-1, 0, 1\}$. It turns out we can find matrices for all 16 functions among those possibilities. (Note that we cannot assume $A = B = C$ as we usually do for several specific convolutions, as for the function $f(x, y) = 0$ there is no such triple.)

Suppose there are square matrices $X = \{x_{ij}\}$ of size $a$ and $Y = \{y_{ij}\}$ of size $b$. Let's denote as a composition $X \circ Y$ of matrices $X$ and $Y$ a square matrix $Z = \{z_{ij}\}$ of size $ab$ such that $z_{bi+j,bk+l} = x_{ik} y_{jl}$ for $0 \le i, k < a$, $0 \le j, l < b$ (in other words, it's just $a^2$ matrices $Y$ put into a matrix $X$ and multiplied by corresponding coefficients from $X$).

We can show (by checking the above condition) that if $(A_1, B_1, C_1)$ is a convolution triple for $f \in F_a$, and $(A_2, B_2, C_2)$ is a convolution triple for $g \in F_b$, then $(A_1 \circ A_2, B_1 \circ B_2, C_1 \circ C_2)$ is a convolution triple for a function $h \in F_{ab}$ such that $h(bx_1 + x_2, by_1 + y_2) = bf(x_1, y_1) + g(x_2, y_2)$. We can also show that $C_1 \circ C_2$ is nondegenerate if both $C_1$ and $C_2$ are nondegenerate.

Thus, we only need to obtain the convolution triples $(A_i, B_i, C_i)$ for all given functions $f_i$, and then $(A, B, C) = (A_{n-1} \circ \ldots \circ A_0, B_{n-1} \circ \ldots \circ B_0, C_{n-1} \circ \ldots \circ C_0)$ is a convolution triple for the original function.

The only thing left for the whole solution is fast multiplication by $A$, $B$, and $C^{-1}$. As all of those matrices are compositions of several $2 \times 2$ matrices, the process of multiplication of $A$ (or $B$) by a vector $v$ can be done in the same manner as for FFT, with the following difference: when we change indices $v[i]$ and $v\left[i + 2^k\right]$, we should apply the matrix $A_k$ to them instead of butterfly transform, i. e.

$$\begin{pmatrix} v_{\text{new}}[i] \\ v_{\text{new}}\left[i + 2^k\right] \end{pmatrix} = A_k \cdot \begin{pmatrix} v_{\text{old}}[i] \\ v_{\text{old}}\left[i + 2^k\right] \end{pmatrix}.$$

While multiplying $C^{-1}$ by a vector, we can just reverse the process similar to above.

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

One could also solve the problem using only convolutions for XOR and AND/OR, as other binary functions can be either handled separately or reduced to those three using negations. There also exist $\mathcal{O}(3^n)$ solutions which are not intended to pass.

## Problem Tutorial: "B. Beautiful Sequence Unraveling"

Let $dp[x][y]$ be the number of beautiful sequences of $x$ elements consisting of integers between 1 and $y$.

Let's calculate the number of all beautiful sequences (which is obviously equal to $y^x$) and subtract all sequences which are not beautiful. To calculate the total number of bad sequences, we calculate the number of bad sequences for which the largest $i$ such that $\max\{a_1, \ldots, a_i\} = \min\{a_{i+1}, \ldots a_n\}$ is equal to $t$ and $\min\{a_{i+1}, \ldots a_n\}$ is equal to $m$.

Note that it means that basically sequences to the left and to the right of $t$ are independent as $a_u \geq a_v$ for all $u \geq t+1$, $t \geq v$. But since $t$ is the largest position with this property, the sequence to the right should also be beautiful.

That's why the answer to this subproblem is equal to $(dp[x-t][y-m+1] - dp[x-t][y-m]) \cdot (m^t - (m-1)^t)$. Here we subtract some numbers because we also have $\max\{a_1, \ldots, a_i\} = \min\{a_{i+1}, \ldots a_n\} = m$.

So, we can already calculate $dp$ in $\mathcal{O}(n^4)$ for all $1 \leq x, y \leq n$. To optimize it, we can notice that, for example:

$$dp[x-t][y-m+1] \cdot m^t = m^x \cdot \left(\frac{dp[x-t][y-m+1]}{m^{x-t}}\right),$$

so after fixing $m, y$, we can calculate all transitions for $x, t$ in $\mathcal{O}(n)$ time by maintaining the sum of values $\frac{dp[i][y-m+1]}{m^i}$ for the prefix. The same is true for all other summands, so now we can calculate the $dp$ table in $\mathcal{O}(n^3)$.

To solve the initial problem, we should calculate $f[i]$ which is the number of beautiful sequences of length $n$ with exactly $i$ different numbers. We can easily calculate it using $dp[n][i]$ and previous values $f[1], \ldots, f[i-1]$. To calculate the final answer, we just iterate over the number of distinct values $t$ in the beautiful sequence and multiply $f[t]$ by some binomial coefficient.

Some $n^3 \cdot \log$ FFT and $n^2 \cdot \log^2$ solutions using FFT and D&C are also possible, but we think they are not supposed to pass.

## Problem Tutorial: "C. Brave Seekers of Unicorns"

Let $dp[i]$ be the number of unicorn sequences with the last element equal to $i$.

Consider the element $x$ of the sequence before $i$ (if it exists). We should add $dp[x]$ to $dp[i]$, but it can happen that the sequence will become "bad" after adding $i$. This happens only if the element before $x$ was equal to $x \oplus i$. So, we should subtract $dp[x \oplus i]$ if $(x \oplus i) < x$.

Now, we should probably add something again as we usually do in the inclusion-exclusion principle. This is the number of unicorn sequences ending with $x \oplus i$ which became "bad" after adding $x$. But it is easy to see that the number of such sequences is equal to zero because a unicorn sequence should be strictly increasing and $(x \oplus i) \oplus x = i > x \oplus i$.

In conclusion, we just need to calculate the sum of $dp[y]$ for all $y$ such that $y < (y \oplus i) < i$. To do this, we can maintain prefix sums of $dp$ array and try all positions of the leading bit of $y$.

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

## Problem Tutorial: "D. Bank Security Unification"

Let $dp_i$ be the maximum possible security of the network over all subsequences ending at the element $i$. This $dp$ can be computed in $\mathcal{O}(n^2)$ by iterating over all possible previous elements of a subsequence. Now, we will reduce the number of possible states to make transitions from.

Denote $H(x)$ as the highest power of two that is less than or equal to $x$. The key observation here is that if subsequence $f_{j_1}, f_{j_2}, \ldots, f_{j_k}, f_i$ is optimal for some $i$, then there is no $j'$ ($j_k < j' < i$) such that $H(f_{j_k} \& f_i) = H(f_{j'} \& f_i)$ (since we can add this element to the subsequence which will lead to greater security of the network). It means that we can only consider positions of the last occurrence of each bit in $f_j$ as a state to make a transition from. Therefore, we can maintain the array $last$, where $last_p$ denotes the position of the rightmost occurrence of the $p$-th bit among all numbers $f_j$ considered earlier, and we will make transitions to the state $dp_i$ only from states $dp_{last_p}$ for all $p$.

For each $i$, we have $\mathcal{O}(\log C)$ states to make transitions from, so the overall complexity is $\mathcal{O}(n \log C)$.

## Problem Tutorial: "E. Brief Statements Union"

We will solve the problem for each bit independently, so we can suppose that for all conditions, $x$ is equal either to 0 or to 1.

We have conditions of the two types for some segments:

1. a segment contains only 1's (the first type);
2. there is at least one 0 on a segment (the second type).

We will call a condition *good* if, after its removal, we can find at least one array which satisfies all other conditions. We can easily find all conditions of the second type which are not satisfied initially. If there is no such condition, then all the conditions are good. If there is only one of them, then it is the only condition of the second type which is good. Otherwise, there are no good conditions of the second type.

Now, let's find good conditions of the first type. First, calculate for each position $i$ of the array how many segments of the first type contain $i$. If there is only one such segment, let's find its index. To do this in linear time, we can use the following classical counting trick:

`For each segment of the first type` $l$, $r$ `with index` $i$ `do: f[l] += i; f[r + 1] -= i.`

After that, if the position $pos$ is contained in only one segment, its index will be $f[1] + \ldots + f[pos]$.

Then, if the position is contained in only one segment, let's write down the index of this segment into this position. It is easy to see that written indices will form contiguous groups (there will be no situations like $a..b..a$). After that, iterate over all conditions of the second type: if it's not violated initially (there is at least one zero on the corresponding segment), then we can skip it. Otherwise, the index of the good condition of the first type should be written at least once on this segment. Because of the previous observation, we can see that good conditions of the first type will form some segment of indices, so to find all good conditions, we should just intersect the segments of indices for all violated conditions of the second type.

The time complexity is $\mathcal{O}(n \log C)$, though it requires careful implementation to make it work fast.

## Problem Tutorial: "F. Border Similarity Undertaking"

We will use a divide-and-conquer approach. Without loss of generality, suppose that $n \le m$ (rotate the matrix and the answer will stay the same). If we divide the matrix by central column, we need to calculate the number of rectangles passing through this column. Let's fix two cells on the central column. After that, we can see that there are two independent problems in the left and right parts of the matrix: count the number of rectangles that have the fixed cells as corners and have the same letter on the border except for the side that lies on the central column. This can be easily done using prefix sums.

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

## Problem Tutorial: "G. Biological Software Utilities"

If $n$ is odd, then the answer is obviously 0. Assume that $n$ is even.

Note that if a tree has a perfect matching, then this matching unique. So, the number of trees which have perfect matching is a product of the following values:

1. the number of ways to split vertices into pairs;
2. the number of ways to connect $\frac{n}{2}$ pairs into a tree.

One can see that the first number is $(n-1)!! = (n-1)\cdot \ldots \cdot 3 \cdot 1$. To calculate the second number, we can notice that we need to add $\frac{n}{2}-1$ edges. When we add each edge between vertices in particular pairs, there are 4 options of choosing which vertices in pairs will be connected by this edge. So, the second number is (number of trees with $\frac{n}{2}$ vertices) $\cdot 4^{\frac{n}{2}-1} = (\frac{n}{2})^{\frac{n}{2}-2}\cdot 4^{\frac{n}{2}-1}$.

In conclusion, the answer is equal to $(n-1)!! \cdot (\frac{n}{2})^{\frac{n}{2}-2}\cdot 4^{\frac{n}{2}-1}$.

## Problem Tutorial: "H. Bytelandia States Union"

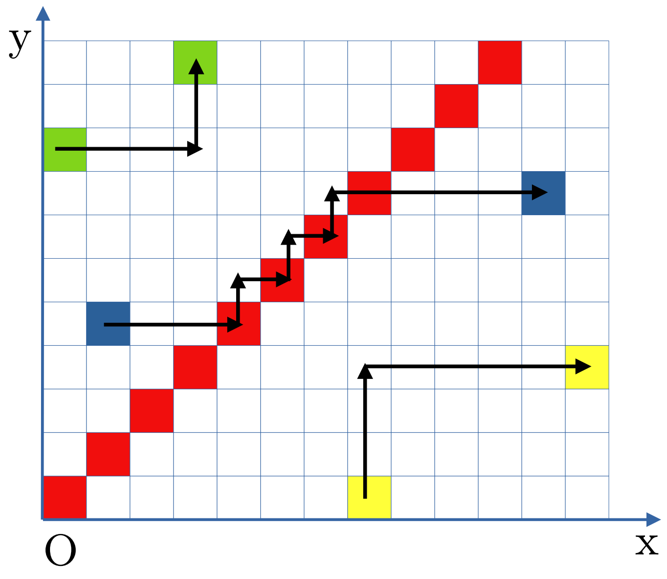
Consider a path that passes through cells $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots (x_k, y_k)$. One can observe that the amount of time for this path is equal to

$$T = x_k^2 y_k^2 - x_1^2 y_1^2 + \sum_{i=1}^{k-1} x_i^2 + y_i^2.$$

It means that we can immediately add $x_k^2 y_k^2 - x_1^2 y_1^2 - x_k^2 - y_k^2$ to the answer (since this part depends only on endpoints, not on the path itself) and focus on minimizing $\sum\limits_{i=1}^{k} x_i^2 + y_i^2$, i. e. the sum of $x_i^2 + y_i^2$ over all the visited squares.

To minimize the sum over $x_i^2 + y_i^2$, one can try various paths and notice the pattern. First, we focus on the case where $x_1 \le x_2$ and $y_1 \le y_2$. In the same way, we consider a similar case $x_2 \le x_1$ and $y_2 \le y_1$. Handle these two cases as one by flipping the endpoints.

The picture below shows an optimal strategy for different endpoints on the rectangle:



The diagonal $x = y$ is marked red on the picture. As one can notice, we do the following: move from the first endpoint to the diagonal, then go alongside the diagonal, and then move to the second endpoint. We go only up and right, i. e. always increasing one of the coordinates. If we cannot reach the diagonal by always increasing one of the coordinates, we just go as near as possible to it.

The other major case is $x_1 \le x_2$ and $y_1 \ge y_2$. In this case, we go down to decrease $y$, and then go right to increase $x$.

One should implement these strategies carefully and calculate all the segments of the path in constant time, using the formula

$$1^2 + 2^2 + \dots + k^2 = \frac{k(k+1)(2k+1)}{6}.$$

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

## Problem Tutorial: "I. Binary Supersonic Utahraptors"

$|a_y - b_r| = |a_y - (m - b_y)| = |(a_y + b_y) - m|$, so the score does not depend on moves of the players.

## Problem Tutorial: "J. Burnished Security Updates"

It is clear from the definition that we need to find a set of vertices of a minimum size such that each edge of the graph has exactly one of its endpoints from the set. It means that each edge connects a chosen vertex with one that is not chosen. Therefore, the graph is bipartite, and its left part consists of all chosen vertices.

From this observation, we check that all connected components of the graph are bipartite and take the smallest part from each component to the final set. This solution runs in $\mathcal{O}(n + m)$.

## Problem Tutorial: "K. Bookcase Solidity United"

First, one should notice that it is not always optimal to break shelves from top to bottom. The second example in the statements clearly shows it.

Notice that the balls do not always fall one by one. When a shelf breaks, many balls fall simultaneously. The pattern of the falling balls matters. If too many balls fall at the same time, they all collide with the next shelf and get lost. If the same amount of balls falls one by one, then some of them don't collide with the next shelf, so more balls remain.

Let's fix the shelves on which we put the balls. After that, the optimal strategy is to put the balls onto the fixed shelves from bottom to top. Why so? Suppose we want to put the balls onto some shelf, and we have already put some balls above. If we don't want to destroy this shelf now, nothing changes if we put these balls before the balls above. If we want to destroy it, then we could have put these balls before we put something onto shelves above, and nothing would change either.

Now, let's use dynamic programming. Let $dp[l][r][k]$ be the minimum amount of balls to destroy all the shelves from $l$ to $r$, assuming that no balls fall from the above and $k$ balls fall below the $r$-th shelf after destruction. We assume that these $k$ balls fall simultaneously. As shown above, this assumption doesn't make the answer better, and if it doesn't hold in reality, this state of DP is just not optimal.

Now, we need to define the transitions. Initially, $dp[l][l][a_l/2] = a_l$ and all other states are assigned to infinity. To which states can we go from $dp[l][r][k]$? We consider different scenarios. First, assume that we haven't put enough balls onto the $(r+1)$-st shelf, and it's solid before we start to put balls onto the $r$-th shelf and above. In this case, we update $dp[l][r+1][\max\{k, a_{r+1}\}/2] = dp[l][r][k] + \max\{0, a_{r+1} - k\}$, i. e. put the balls onto the $(r+1)$-st shelf in such a way that it would break immediately after we destroy the segment $[l; r]$. Otherwise, we broke the $(r+1)$-st shelf before we started to break $[l; r]$. Then, the segment $[r+1; p]$ is already broken for some $p \le r+1$. So, we iterate over $p$ and $k_1$ and update $dp[l][p][k + k_1] = dp[l][r][k] + dp[r+1][p][k_1]$.

The final step is to notice that in DP, we need to consider only such $k$ for which $k \le \max a_i$. Indeed, if no more than $\max a_i$ balls can fall onto a shelf simultaneously, then no more than $\frac{a_i + \max a_i}{2} \le \max a_i$ balls can fall below simultaneously. Using this fact and knowing that there exists a shelf for which nothing falls from above, we can prove by induction that only $k \le \max a_i$ is needed.

The answer for prefix $[0; i]$ can be obtained as a minimum of $dp[0][i][k]$ over all $k$.

Let's estimate the time complexity. Assume that $m = \max a_i$. We have $\mathcal{O}(n^2 \cdot m)$ states and $\mathcal{O}(n^3 \cdot m^2)$ transitions, so the time complexity is $\mathcal{O}(n^3 \cdot m^2)$. But the constant factor is pretty low, and some DP states are never used.

## Problem Tutorial: "L. Business Semiconductor Units"

The first thing to say about this problem is that the given instruction set is Turing-complete (see the paper `http://stedolan.net/research/mov.pdf` for details).

Let's deduce more complex operations using the given simple operations:

- *Conditionals.* The most essential part is to learn how to perform conditional loads and stores. The idea here is simple. Let's assume that `false` is equal to constant $255 \cdot 256 = 65280$, and `true` is equal to $255 \cdot 256 + 2 = 65282$. So, we perform `ld a, b` if condition in the register `c` is true, in the following way:

  ```
  imm r15, 65280
  st r15, a
  ld r14, b
  imm r15, 65282
  st r15, r14
  ld a, c
  ```

  As you can see, the original value in the register `a` is loaded to the address 65280, and the new value is loaded to the address 65282. Then, we load the value to `a` according to the condition in `c`. If it's false, the same value is loaded back to `a`, otherwise, the new value is loaded.

- *Moving from register to register.* This is as simple as

  ```
  imm r15, 65284
  st r15, a
  ld r15, b
  ```

- *Getting or changing the upper part.* To achieve this, we may read or write from the address shifted by one. For example, if we want to change the upper part of `a` to 142, we may do the following:

  ```
  imm r15, 65284
  st r15, a
  imm r15, 65285
  imm r14, 142
  st r15, r14
  imm r15, 65284
  ld a, r15
  ```

  In `st r15, r14`, we overwrite the upper part of `a` with 142, and then we load the modified value back. We can use the same trick to retrieve the upper part of the number or shift the number by 8 digits right.

  We further assume that `cup a, b, c` sets the upper part of `b` to some constant `c` and stores the modified value into `a`.

- *Equality check.* First, learn how to check whether the lower parts of the numbers are equal, the upper parts are checked in the same manner. If all the memory cells from $253 \cdot 256 = 64768$ to $254 \cdot 256 = 65024$ are zeroed, then we can do the following to compare the lower parts of the registers `a` and `b`:

  ```
  cup r15, a, 253
  cup r14, b, 253
  imm r13, 2
  ```

```
st r15, r13
ld r13, r14
cup r13, r13, 255
imm r14, 0
st r15, r14
```

This works as follows. We put 2 to the address $253 \cdot 256 + low(a)$ and then read the number from $253 \cdot 256 + low(b)$. If $low(a) = low(b)$, then we get 2 in the lower part, otherwise, we get 0. If we add 255 to the upper part, then the value (stored in `r13` in the code above) will be equal to `true` if $low(a) = low(b)$, and `false` otherwise. Finally, we clean up.

Now, we move to more complex operations. The essential part is how to perform additions and multiplications. For this purpose, we will perform all the calculations in the base 16. So, we may want to compute $x \bmod 16$ and $\lfloor \frac{x}{16} \rfloor$ for the lower part of the number. This can be done with precomputed tables. Consider $\lfloor \frac{x}{16} \rfloor$, other operations can be done in the same manner. For this purpose, we may store the value of $\lfloor \frac{x}{16} \rfloor$ at the address $252 \cdot 256 + x$. To find $\lfloor \frac{a}{16} \rfloor$ for the value in the register `a` and to place the result into `b`, we do the following:

```
cup b, a, 252
ld b, b
cup b, b, 0
```

Now, consider how to calculate $a \cdot 16 + b$. To do this, we need 16 tables, each of them consists of 16 values. Suppose that the first table starts from address $234 \cdot 256$, the second table starts from address $235 \cdot 256$ and so on. The $j$-th value in the $i$-th table contains the value $i \cdot 16 + j$. There is also an *index table*, starting from the address $233 \cdot 256$. The $i$-th value of the index table contains $234 + i$, i. e. the upper part of the start address for the $i$-th table. Then, the operation looks as follows, assuming that the result is stored in `c`:

```
cup r15, a, 233
ld r15, r15
cup r15, r15, 0
cup r14, b, r15
ld c, r14
```

Using the operations described above, addition and multiplication can be performed using the standard algorithm. To store the results for $a + b$ when `a` and `b` are hexadecimal digits, one can compute $a \cdot 16 + b$ and store the result for $a + b$ in some table by index $a \cdot 16 + b$.

Now, let's take a broad look at the whole algorithm. We store the following variables:

- $sz$, the size of the array;
- $pos$, the position in the array;
- $ptr$, the pointer to the next number (basically, position multiplied by two);
- $res$, the accumulator (i. e. product of all the processed numbers);
- $c_s$, the condition whether the program has started or not;
- $c_e$, the condition whether the program has processed all the required numbers or not.

Some of these variables can be stored in registers, and some of them can be stored in memory to save valuable space in the registers.

Now, the algorithm is performed as follows:

1. fill the tables to precompute the operations;
2. if $c_s = 0$, then it's the first step, so we assign $res = 1$ and $c_s =$ `true`;
3. if $c_e = 0$ and $pos = sz$, then put the result into `r0` and set $c_e = 1$. Some operations will be done later, but they won't affect `r0` and $c_e$, so it will be the final answer;
4. read the value pointed by $ptr$ into some memory cell $tmp$, increase $pos$ by one and $ptr$ by two;
5. decompose $tmp$ and $res$ into the base 16;
6. perform multiplication;
7. store the result back in $res$.

One can notice that we process a single element on each step, so we require $n$ steps to finish the computation.

Careful implementation is required for this problem. Use registers and memory in such a way that the operations do not clobber registers for each other. Also, the registers should be used reasonably because there are only 16 registers.

Note that the constraints on the number of operations are rather loose for this problem. The model solution can achieve the result in 5000 operations. By the way, the authors of the problem have a solution that calculates everything in the base 2 instead of the base 16 and requires approximately 50 000 instructions.

## Problem Tutorial: "M. Brilliant Sequence of Umbrellas"

Let's try to obtain an infinite sequence $\{a_i\}$ such that for any $n$, the sequence has a prefix that satisfies all requirements.

Denote $g_0 = 1$, $g_i = \gcd(a_i, a_{i+1})$. Clearly, $a_i$ should be divisible by both $g_{i-1}$ and $g_i$, so let's assume $a_i = b_i \cdot \operatorname{lcm}(g_{i-1}, g_i)$. Rewriting the definition of $g_i$ using the previous formula, we can obtain that for all $i \geq 1$, $\gcd(b_i \cdot \operatorname{lcm}(g_{i-1}, g_i), b_{i+1} \cdot \operatorname{lcm}(g_i, g_{i+1})) = g_i$ should hold.

If we consider the degrees of a certain prime in the variables of the above equation, we can imply that $\gcd(b_i, b_{i+1}) = 1$, and that $g_i$ is divisible by $\gcd(g_{i-1}, g_{i+1})$.

For simplicity, we can try $b_i = 1$, $\gcd(g_i, g_{i+1}) = \gcd(g_i, g_{i+2}) = 1$ for all $i \geq 1$, and that will be enough to satisfy all requirements except the length of the sequence.

We can obtain the sequence $\{g_i\}_{i=1}^{+\infty}$ greedily by adding positive integers one by one, as long as they are coprime with two most recently added numbers. We can prove using induction that a number is present in the resulting sequence of this greedy process iff its remainder modulo 6 is in $\{1, 2, 3, 5\}$. Using these values of gcd, we can obtain the sequence $\{a_i\} = \{1 \cdot 1, 1 \cdot 2, 2 \cdot 3, 3 \cdot 5, 5 \cdot 7, 7 \cdot 8, \ldots\}$, for which we can prove that $a_i \leq \frac{9}{4}i^2$ holds for any $i$, thus for any $n$ we'll be able to pick a suitable prefix.

## Problem Tutorial: "N. Best Solution Unknown"

Let's check when the participant $i$ can win the competition. Notice that any match without the person $i$ is unprofitable for them because it boosts their opponents. That's why the person $i$ will participate in all matches.

Also, we can see that a person with the largest strength can win the competition by winning all the opponents one by one.

Let $m$ be an index of any participant with the largest strength. Notice that the participant who beats the person $m$ in their match can win the whole competition. Now, for the person $i < m$, the winning strategy is to beat all the opponents on positions from 1 to $m - 1$, then beat $m$, and then beat the rest (they can do it in such order because $a_m$ is the largest strength). Assume that the person $i < m$ can beat all the opponents on positions from 1 to $m - 1$. The person $i$ can beat $m$ iff $a_m \leq a_i + m - 2$, or $a_m - m + 2 \leq a_i$.

Now, let's solve the problem in the range of participants $[L, R]$ with an additional statement that requires the initial strength of the winner to be not less than some constant $C$. The initial problem is ($L = 1$, $R = n$, $C = 0$). Let $M$ be the index of any of the participants with the largest strength in the range $[L, R]$. Then the person $M$ can beat all the other participants in the range, and this person can be the winner iff $C \leq a_M$. Divide $[L, R]$ into two segments $[L, M-1]$, $[M+1, R]$. We need to solve a subproblem on the range $[L, M-1]$ with the additional statement $\max\{C, a_M - M + L + 1\} \leq a_i$. We also do the same for the range $[M+1, R]$ with a similar additional statement.

This way, we have $\mathcal{O}(n)$ calls of a recursive function, and for each call, we need to find the largest element on a range. This can be done using a segment tree (or sparse tables), so the overall complexity is $\mathcal{O}(n \log n)$.