Problem Tutorial: "AND"

Firstly, it is easy to see that the `AND` of all numbers in b is equal to the `AND` of all numbers in a, so it should also lie in b. Let us call this number x. If this condition holds, it is easy to construct the following answer:

$b_1, x, b_2, x, \ldots, b_{n-1}, x, b_n$

Problem Tutorial: "Bruteforce"

Note that we can rewrite $\left\lfloor \frac{b_i \cdot i^k}{w} \right\rfloor$ as $\frac{b_i \cdot i^k - (b_i \cdot i^k) \bmod w}{w}$. So in some sense we have two independent problems: finding $\sum_{i=1}^{n} b_i \cdot i^k$ and $\sum_{i=1}^{n} (b_i \cdot i^k) \bmod w$. We will solve both problems using segment tree. We will build our segment tree over values(so it will have fixed size of 10^5), soeach node will maintain information about some subsegment of values. Let's talk about second subproblem first: for each node we will maintain $cnt[x][y]$ — how many numbers c in this node have $b_c \bmod w = x$ and $c \bmod w = y$. When we compute this value, we consider only numbers stored in this node(so number c ranges from 1 to amount of numbers lying in this node). As usual, the hardest thing that we need to do is to be able to merge information from two sons. In this subproblem, it's easy to do this, because the only thing that happens is change of indexation for right son(each index will be shifted by amount of numbers in left son). So we are able to do single merge in $O(w^2)$ time.

To solve first subproblem, we will maintain $f[t]$ — $\sum_{i=1}^{n} (b_i \cdot i^t)$ for $0 \le t \le k$. Then, to do merge we should be able to compute $\sum_{i=1}^{n} (b_i \cdot (i + shift)^t)$ for some integer value of $shift$. It's easy to see, that after opening brackets, it will be equal to $\sum_{i=1}^{n} \sum_{x=0}^{t} (b_i \cdot i^x \cdot \binom{t}{x} \cdot shift^{t-x}) = \sum_{x=0}^{t} f[x] \cdot \binom{t}{x} \cdot shift^{t-x}$. So, we can do merge in $O(k^2)$.

Each query changes only one leaf node in segment tree, so we can do single update in $O((k^2+w^2) \cdot log(10^5)$ time. So complexity is $O((q+n) \cdot (k^2 + w^2) \cdot log(10^5))$

One small note — in terms of implementation, it's better to add value $b_0 = 0$ and solve problem in 0-indexation.

Problem Tutorial: "Crab's Cannon"

This problem has short and beautiful solution. I have discovered a truly marvelous explanation of this, which this document is too small to contain.

That was a joke, so let's start the problem analysis. First, we will solve a simpler problem: we are given a set of integers, and we need to tell if there exists a string with its PPS equal to the given set. After solving this simpler problem, we use the observations from there to solve the harder version. I will try to provide a complete proof, so the analysis will be quite long. It's also possible that the solution can be proved in a shorter way, but I didn't think of it.

Solving a simpler problem

Convention

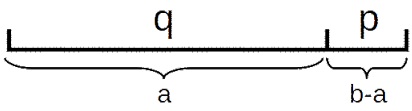
- s' denotes reversed s. So, if $s =$ "`crab`", then $s' =$ "`barc`"
- $a + b$ denotes concatenation of strings a and b
- Palindromic prefix of length x is called just "prefix x". It also means that x is in the PPS.

For simplification, we assume that 0 is present in the PPS. So, we add 0 to the given set. Also, we sort the set beforehands.

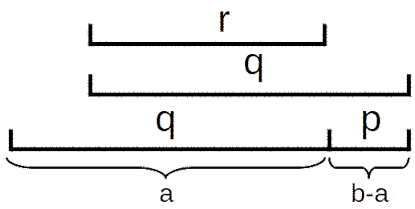
We need to define some criteria which are necessary and sufficient to determine if the set can be PPS of some string.

First of all, 1 must be present in the set (string of length 1 is always a palindrome), otherwise the answer is "NO".

Then consider two numbers from the set, a and b ($a < b$). What information we can obtain from this fact?

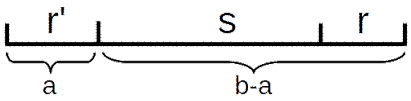


It's easy to see that, since $q + p$ is a palindrome, $q + p$ ends with q'. But, since q' is also a palindrome, $q' = q$:



Then consider the string r. It's the prefix of q and it the same time it's the suffix of q, thus r' is the prefix of $q' = q$. So, we get $r' = r$. r has length $a - (b - a) = 2a - b$. So, we get the palindrome prefix of length $2a - b$.

What if $2a - b < 0$? This is also a valid case, because the string grows more than twice:



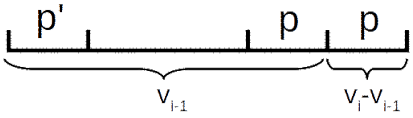
s can be any palindromic string.

So, the first condition (denote it C_1) is as follows: if a and b are in the set, then $2a - b$ is also in the set, or $2a - b < 0$.

Now we can try the following algorithm: for each a and b in the set ($a < b$), check whether $2a - b$ is negative or is present in the set.

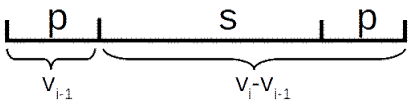
It can be shown that it's enough to check only the pairs a and b that are neighbors in the set. So, sort the set and for each i ($2 \le i \le n$), check if $2v_i - v_{i-1}$ is in the set or it's negative.

The idea above also gives us a way to construct such a string in $O(\ell)$ time. Iterate over the set. Suppose we had prefix v_{i-1} and now we consider prefix v_i:



Look at the picture above. The prefix v_i ends with p. It's a palindrome, thus the string starts with p'. But since the prefix v_{i-1} is also a palindrome, the string ends with p. So, we copy the suffix of length $v_i - v_{i-1}$ to the end of the string.

If $2v_i - v_{i-1}$ is negative, we don't have the suffix of length $v_i - v_{i-1}$:



So, we copy p to the end of the new string and fill s with unused characters. One can easily show that such situation happens no more than $\lceil \log_2 \ell \rceil + 1$ times, so we can use the alphabet of size no more than $\lceil \log_2 \ell \rceil + 1$.

But C_1 is not sufficient. Consider the set $\{1, 2, 4\}$. C_1 is still held, but the required string doesn't exist (constructing the string gives "aaaa", which has 3 in its PPS). What are we missing?

Using the constructive method above, we can just build the string and check if the given set and the PPS of the constructed string match. But it's $O(\ell + n)$, which doesn't pass if $\ell = 10^{18}$. So, it's a better idea to search for more conditions.

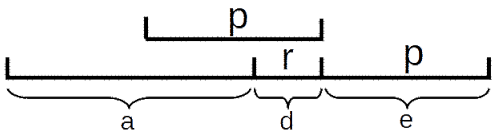
We can notice that while moving from prefix v_{i-1} to prefix v_i, we can skip some palindromic prefixes. It happens, for example, when we copy the string p in the constructive method, but this string is periodic. In this case, if we add only the period, we still get a valid palindromic prefix. So, we need to cut out such cases.

More formally, the following theorem is true:

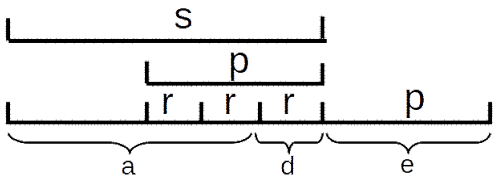
Theorem 1. If we have prefixes a, b, c $(a < b < c)$ such that $c - b \neq b - a$, $2b - c \geq 0$ and $c - b$ is divisible by $b - a$, we have a prefix $b + (b - a) = 2b - a$ between b and c.

Remark. The second condition is required, because if $2b - c < 0$, some characters will be filled with the symbols never met before, so we won't get a periodic string.

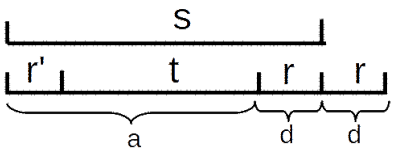
Proof. Denote $d = b - a$, $e = c - b$. Since e is divisible by d there exists such k that $e = kd$. Look at the picture below:



Since prefixes a and b are palindromic, $a - d$, $a - 2d$, $\ldots$, $a - kd$ are also palindromic prefixes. Using the same idea as in the contructive algorithm, it can be shown that p is equal to r repeated k times:



It's left to prove that $s + r$ is a palindrome, so there is a prefix $b + d = b + (b - a) = 2b - a$.



Apparently $s = r' + t + r$ is a palindrome, so t is also a palindrome. Also $r' + t$ is a palindrome, so $r' + t = t + r$. We have

$$s + r = r' + (t + r) + r = r' + (r' + t) + r = r' + (r' + t + r)' = r' + s' = (s + r)'.$$

So, $s + r$ is also a palindrome.

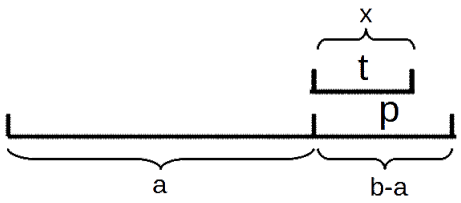
So, we proved that the condition in *Theorem 1* (denote it C_2) is necessary. C_2 can be used to check that we didn't skip a prefix in our set. If we have two prefixes v_i and v_{i-1}, we need to ensure that there's no prefix k such that $v_i - v_{i-1}$ is divisible by $v_{i-1} - k$, $2v_i - v_{i-1} \geq 0$ and $v_i - v_{i-1} \neq v_{i-1} - k$. It will be proved later that $k = v_{i-2}$ is enough to check.

It appears that these conditions are sufficient and there are no other prefixes we missed in the set. Let's prove it.

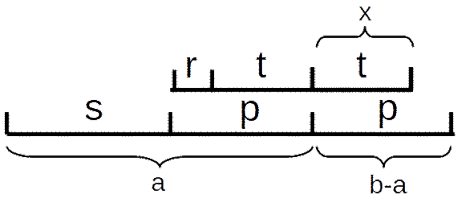
Proof. Proof by contradiction. Suppose we have $a = v_{i-1}$, $b = v_i$, the conditions above (C_1 and C_2) are met, and there's a palindromic prefix $b_1 = a + x$ $(0 < x < b - a)$ between a and b. If there multiple such x, we use the minimal x possible.

First, if $2a - b < 0$, then using the constructive algorithm above, we build a string in which $b - 2a$ positions in the middle are filled with new characters, so there cannot be any palindromic prefixes between a and b.

Now consider the case when $2a - b \geq 0$:



Using C_1, we can see that $2a - b$ and $a - x$ are also palindromic prefixes:



As both $s + p + p$ and $s + r$ (prefixes $a - x$ and b respectively) are palindromes, $s + p + p$ ends with r. So, we get $r + t = t + r$, or similarly, the string is equal to its cyclic shift. It means that the string p is periodic. So, we get the contradiction with C_2.

So, C_1 and C_2 are necessary and sufficient. Now we can construct the algorithm in $O(n \cdot \log n)$ time (or even in $O(n)$ time if you use a hash set) to check both conditions. Unlike the analysis, the code is very neat and short :)

```
bool isValidPalindomicPrefixSet(vector<int64_t> v) {
  if (v[0] != 1) return false;
  v.insert(begin(v), 0);
  set<int64_t> s(begin(v), end(v));
  for (size_t i = 2; i < v.size(); ++i) {
    if (v[i] - v[i-1] == v[i-1] - v[i-2]) continue;
    if (2 * v[i-1] - v[i] < 0) continue;
    if (!s.count(2 * v[i-1] - v[i])) return false;
    if ((v[i] - v[i-1]) % (v[i-1] - v[i-2]) == 0) return false;
  }
  return true;
}
```

Now the only thing we need to prove is that it's enough to take only neighbors while checking C_1 and C_2.

To simplify the reasoning, consider the PPS in terms of *segments*. A *segment* here is the distance between two neighboring prefixes. For example, if the string has palindromic prefixes of length $\{0, 1, 3, 7, 11\}$ (do not forget that we also consider prefix of length zero), then we can construct the segments of length $\{1, 2, 4, 4\}$. A *group of segments* is a set consisting of one or more consecutive segments. The total length of the group of segments is the total length of all the segments in the group. Each segment and group of segments have their starting and ending prefixes, i. e. the numbers each prefix starts and ends with.

Later we will use the length of a segment or a group of segments. Length of the segment s is denoted $|s|$. Length of the group G is denoted $|G|$.

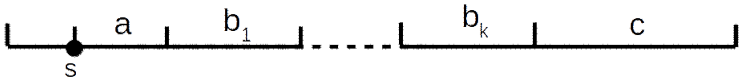
It's easy to notice that the lengths of the segments are non-decreasing, otherwise C_1 isn't held.

We can also explain C_1 and C_2 in terms of segments. C_1 means that if we have a group of segments G starting at prefix p, then we also have a group of segments H ($|H| = |G|$) ending at p, or $p < |G|$. And C_2 means that for each segment s starting at p and each group G ending at p, one of the following conditions is held: $|s| = |G|$, or $|s|$ is not divisible by $|G|$, or $p < |s|$.

We also need to prove the following statement:

Theorem 2. Suppose we have the segment b ending in p and the segment c starting in p, and $|b| < |c|$ applies. Then, $|b| + |c| > p$.

Proof. Consider the neighboring pairs of segments b and c such that $|b| < |c|$ from left to right and check if $|b| + |c| > p$ for all of them. For first such pair we obviously have $|b| = 1$, so $|c| > p$ must hold (otherwise we get a contradiction with C_2 because $|c|$ is always divisible by $|b| = 1$). Otherwise, the situation is as follows:

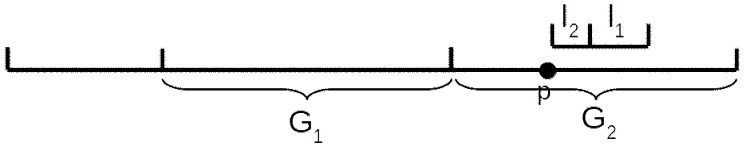


Here, a, $b_1, \ldots, b_k$ and c are segments, and s is some prefix. Also we have $|b_1| = \cdots = |b_k|$ and $b_k = b$. Check if the theorem is true for segment c. By considering all the smaller prefixes, it's already proved that $|a| + |b| > s + |a|$. Now we need to prove that $|b| + |c| > s + |a| + k \cdot |b|$, so the entire theorem will be proved. It's easy to see that $|c| \geq k \cdot |b| + |a|$ (otherwise either C_1 isn't held or $|c|$ is divisible by $|b|$ and C_2 isn't held). So, we have $|b| + |c| \geq |b| + k \cdot |b| + |a| > k \cdot |b| + s + |a|$.

Using the results above, we can prove another theorem:

Theorem 3. If we have two neighboring groups of segments G_1 and G_2 (G_2 is to the right of G_1) such that $|G_1| = |G_2|$, then G_2 consists of segments of equal length.

Proof. Proof by contradiction. Suppose we have such neighboring groups G_1 and G_2 such that G_2 contains at least two segments (l_2 and l_1) of different lengths. It's easy to notice that we can pick l_2 and l_1 in such a way that they are neighbors:



It's safe to assume that $|l_1| > |l_2|$. Also, both l_1 and l_2 are in G_2, and $|G_1| = |G_2|$, so $p \geq |l_2| + |l_1|$. Contradiction with *Theorem 2*.

Finally, we show that taking neighbors while checking C_1 and C_2 is sufficient.

First, prove it for C_1 by contradiction. Then there's a group of segments G starting at p, which violates C_1 and consists of more than one segment. If there are two segments of different lengths, then $|G| > p$ by *Theorem 2*, so C_1 is held. Now suppose G consists of k segments of the same length l. In this case we need to check for the prefix $p - kl$. But we can achieve this only using one segment of length l. So, we don't miss anything.

To show that checking only neighbors is valid for C_2, we use *Theorem 3*. Consider adding a segment s with starting point p. If $|s| > p$, then C_2 is held. Otherwise there is a group G ending in p such that $k \cdot |G| = |s|$ and $k > 1$ (if $k = 1$ or $|s|$ is not divisible by $|G|$, C_2 is held either). But as $k \cdot |G| = |s| \le p$, we can show using C_1 that there is a group H right before G such that $|G| = |H|$. Now *Theorem 3* shows that G consists of segments of the same length l. But in this case checking only the segment of length l ending in p is OK.

Good, we finally dealt with the simpler version. So, let's move on and solve the harder one.

Solving the harder version

In the harder version, we need to add minimum amount of numbers in the set in such a way that it becomes a PPS or some string. When we know the solution for the simpler version, it's easy to construct a naive solution for the harder one:

- If 1 is not in set, then add it.
- Check if C_1 holds, i. e. for each neighboring prefixes a and b ($a < b$) either $2a - b < 0$ must hold or $2a - b$ must be present in the set. If not, add $2a - b$ to the set.
- Check if C_2 holds, i. e. for segment s starting at prefix p that has previous segment t, if $|s|$ is divisible by $|t|$ and $|s| \le p$, then the segment s must be split with a new prefix into smaller segments that have length $|t|$.

Unfortunately, such solution works in $\Theta(\ell)$ time in the worst case (i. e. not better than in linear time), so we need something better.

First, forget about the prefixes itself. Now we consider only segments. And, what's more, we compress the information about the segments, keeping pairs of the form (l, c) instead of original segments, where l is the length of one segment, and c is the amount of segments of such length coming sequentially. For example, the set of prefixes $\{0, 1, 3, 7, 11, 15\}$ will be transformed to the sequence of segments $\{1, 2, 4, 4, 4\}$ and then transformed to the sequence of pairs $\{(1, 1), (2, 1), (4, 3)\}$.

Call the segment s starting at prefix p *large* if $|s| > p$. (By the way, p is the sum of previous segments' lengths.)

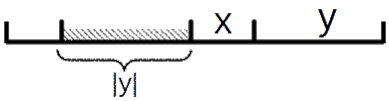
Now, let's take a different look at C_1 and C_2. C_1 says that a segment s must be either large, or $|s|$ must be equal to the sum of lengths of sequential segments coming right before s. Similarly, C_2 says that a segment s with previous segment t must be either large, or $|s| = |t|$ holds, or $|s|$ is not divisible by $|t|$.

Recall that the segments' lengths are non-decreasing, otherwise a C_1 doesn't hold. Another useful observation comes from *Theorem 3*: if the segment s starting from prefix p is longer than the previous one, t, then $s \ge \frac{p}{2}$. So, adding a new segment with different length increases the total length at least by half, which means that the number of pairs in compressed representation is $O(\log \ell)$, which is much smaller than the original sequence of segments.

Before going further, we prove the following theorem:

Theorem 4. Consider a segment s, which is not large. This segment is split into two smaller segments, x and y ($|x| + |y| = |s|$). If the split these segments to the segments of equal length $g = \gcd(|x|, |y|)$, the answer to the problem won't change.

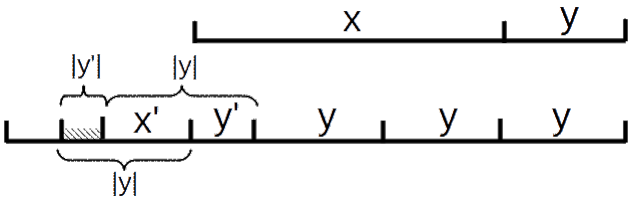
Proof. If s is not large, then there exists a group of segments G such that $|G| = |s|$. For the sake of clarity, we merge the group G into one segment and use the last $|y|$ ($|y| \le |G|$) positions of it as a scratch space:



Now we need to prove the possibility of such replacement if we have $|y|$ positions of scratch space before x. We do it by induction on $|x|$ and $|y|$.

If $|y|$ is divisible by $|x|$, then we use C_2 to split $|y|$ to the segments of size $|x| = g$. If $|x|$ is divisible by $|y|$, then then we apply C_1 to segments of length y sequentially, splitting x by segments of size $|y| = g$. In both cases, the theorem is proved.

Now consider the case where y is not divisible by x. Then, we apply C_1 to segments of length y sequentially, cutting the segments of size $|y|$ from x. The length of the remaining segment y' is equal to x modulo y. Then, we apply C_1 once more, getting the segment x' (see the figure below):



We used $|x'|$ positions of scratch space, so $y - |x'| = y - (y - |y'|) = |y'|$ positions remain. So we perform an induction step and split x' and y' into segments of size $g' = \gcd(|x'|, |y'|)$. By Euclid's theorem, $g' = g$. Then, as $|y|$ is divisible by $|g'|$, we apply C_2 to all the upcoming segments of length $|y|$ and split them into the segments of length $|g'|$. The theorem is proved.

What basically happened in the proof is that we used Euclid's algorithm to split the segments. Neat.

Now, we come up with an optimal algorithm. We maintain the sequence of pairs (l, c) for all the added prefixes. We add the prefixes from the input one by one, in sorted order. Do not forget to add 1 to the input set beforehands if it's not already present.

When we add a new prefix, we push a new segment s and try to apply C_1 and C_2 until we get a valid PPS. So, we have $O(\log \ell)$ pairs after each addition.

To perform addition fast, we do it in the following way:

1. If s is a large segment, we can just add it.
2. Suppose t is a segment previous to s. If $|s|$ is divisible by $|t|$, then we split s into the segments of size $|t|$, add them and finish the addition.
3. Let $|t| > |s|$. Then, we try to apply C_1 to s once. If no new segments are added (i. e. $|s|$ is equals to the size of some neighboring group G), then we just add it and return. Otherwise, some segment is split. We termorarily pop all the segments after the split, applying them later. Now consider the segment being split:

   - If this segment is large, we just try to add the two subsegments recursively.
   - Otherwise, we use the result from *Theorem 4* and split those two segments x and y into smaller segments of size $g = \gcd(|x|, |y|)$.

   When returning the popped segments back, we use the same algorithm recursively. But it won't take long, as the algorithm either just adds a segment, or splits it to equal subsegments on step *2)*, so it's $O(1)$ for each readded segment.

4. Otherwise, $|t| < |s|$. We apply C_1 many times as in the proof of *Theorem 4*, getting many segments of length s and one segment t' of length t modulo s. Now, we have two cases:

   - If t was a large segment, we add the segment t' and one of the segments of length $|s|$ recursively. Then, we add the rest of the segments of length $|s|$ in $O(1)$ time, as they are either added untouched, or all of them pass through step *2)* of the algorithm and are split on the subsegments of the equal size.
   - Otherwise, we use the result from *Theorem 4* and split all the segments into the subsegments of size $g = \gcd(|s|, |t|)$.

It's easy to see that this algorithm works. All we have to do now is to estimate its time complexity. Consider the cases with two recursive calls. They only happen if we split a large segment. If at least one of the subsegments is large, then the corresponding recursive calls stop at step *1)*, so such call is basically $O(1)$. The recursive call which is not stopped proceeds with a smaller number of pairs, so these cases have $O(\log \ell)$ time complexity.

Otherwise, we split a large segment into two non-large segments. Such event destroys a large segment completely. It's not hard to observe that there are at most $O(\log \ell)$ large segments created during all the additions, since each large segment doubles the total length of the segments. Even if we obtain a large segment by splitting two large segments, the original segment had quadrupled the total segment length when it was added. So, there will be no more than $O(\log \ell)$ such splits, and $O(\log \ell)$ extra recursive branches, respectively.

The considerations above give time complexity $O(\log^2 \ell + n \log \ell)$. Though, I don't know how to generate a test for which $O(\log^2 \ell)$ part runs much slower than $O(n \log \ell)$ part.

So, finally, after all those pages of theorems and explanations, we obtain a fast and beautiful soltuion for this problem :)

Problem Tutorial: "Deleting"

Let's consider(hopefully) slow solution first — $dp[l][r]$ is the smallest value for deleting numbers from l to r. If number l was deleted in pair with number i then we have two cases: 1. $r = i$, in this case minimum value is $max(cost[l][r], dp[l+1][r-1])$.

2. $i < r$, then segments $[l, i]$, $[i+1, r]$ are independent and value is $max(dp[l][i], dp[i+1][r])$.

So slow solution which works in $O(n^3)$ is to compute this dp table, doing transitions in $O(n)$ time.

To speed it up, we will compute dp values in increasing order. Also, we will make forward transitions(so we will try to do transitions from already computed values to uncomputed ones). Suppose that now we consider segment $[l, r]$ and we are trying to make transitions from it. It's easy to do transition of the first type — we can set $dp[l-1][r+1]$ to $dp[l][r]$ if $cost[l-1][r+1] < dp[l][r]$ and $dp[l-1][r+1]$ is still not computed.

Transitions of the second type are harder. Let's assume that segment l, r is left one in transition. Then, we need to find numbers $k > r$ such that:

1. Value of $dp[l][k]$ is still not computed.

2. Value of $dp[r+1][k]$ is computed.

Then we are able to set $dp[l][k]$ to $dp[l][r]$.

To do this, we will use the best structure in the world — bitset. So, for each l we will maintain bitset of already computed values and uncomputed values. Then, finding k is just operation of AND. After this we will iterate over all k(for C++ users you can use Find_first() and Find_next(p) in std::bitset), and update the value of dp for them. Note that we will update the value of dp for each segment at most once, so this operations take $O(\frac{n^3}{64})$ in total.

There are some implementation notes:

1. Input is rather large, so using fast reading methods(for example, fread) can help a lot.

2. You can write solution without priority queue, because costs are small. Moreover, since they are disticnt, you can iterate over dp values in increasing order and, if current value is equal to z, then the only place where new value using first transition type can appear is l, r such that $dp[l][r] = z$. After that, you can do transitions of the second type, starting from this segment, you can do them in query(simple one, not priority) like structure, since all updated values will also have value of dp equal to z.

This optimizations are not needed to fit you solution in time(we have solution with priority queue and cin reading method), but are certainly useful in case you solution is a little bit unoptimal.

Problem Tutorial: "Eulerian?"

We need to check if all degrees are even.

First, let's find the total number m of edges in the first query, asking about the entire graph.

Then, do 29 iterations of the following process:

- Divide all vertices into 2 parts A and B randomly (each vertex goes to A or B with equal probability)
- Ask how many edges are in A, and how many in B, and by this deduce how many edges are there between A and B.
- If at any iteration this number is odd, report that there is no Eulerian cycle. Otherwise, there is one.

This works because the parity of the number of edges between A and B is equal to the parity of sum of degrees of vertices in A (because each edge between A and B contributes 1 to this sum, and each edge inside A contributes 2). So, if all degrees are even, the number of edges between A and B is also even. If, however, degree of some vertex X is odd, this number has probability $\frac{1}{2}$ of being even: if we move X to another group, this parity changes. Therefore, if on any of 29 runs this value is odd, there is no Eulerian cycle, else there is one (probability of failing is $\frac{1}{2^{29}}$).

Problem Tutorial: "Fancy Formulas"

Firstly, note that the sum of the numbers stays the same modulo p. If this condition doesn't hold, just output -1.

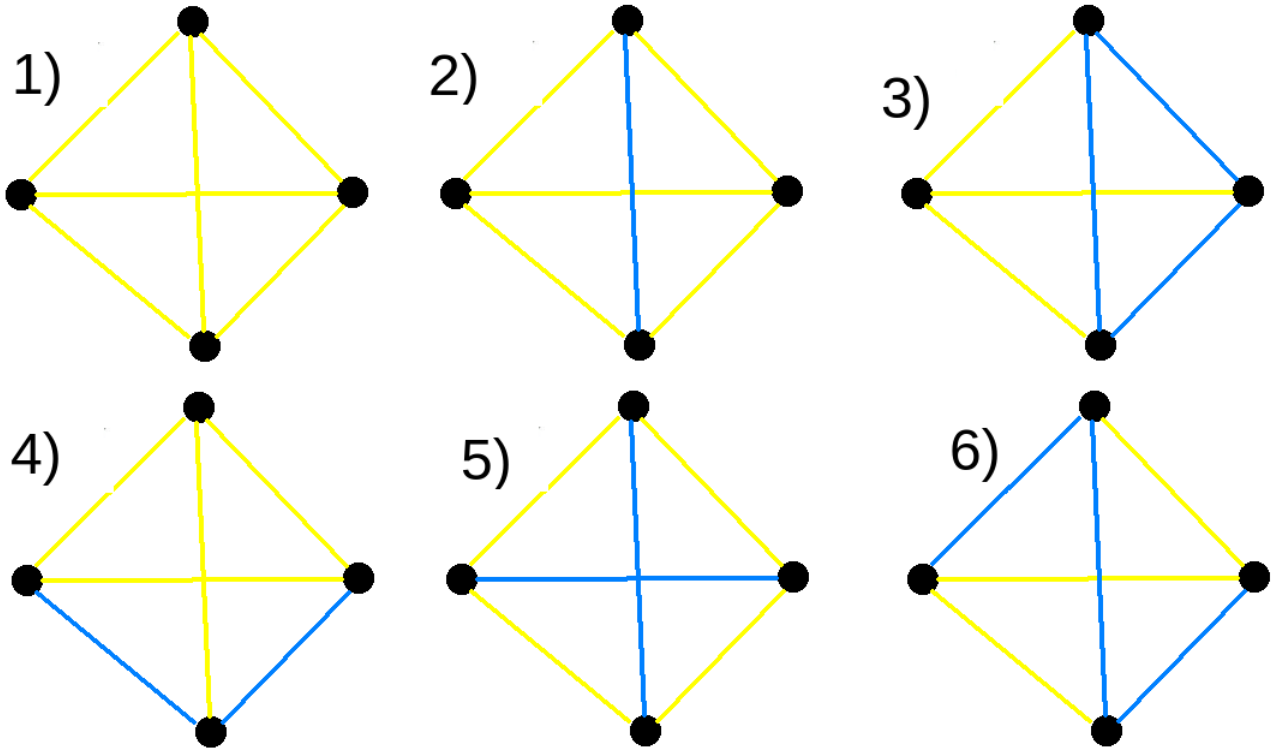
Now, let $s = (a + b) \bmod p$. Multiply all numbers a, b, c, d by s^{-1}, and the smallest number of operations clearly won't change. Note that from pair $(a, 1 - a)$ we can go to one of $(2a \bmod p, (1 - 2a) \bmod p)$, and $((2a - 1) \bmod p, 2(1 - a) \bmod p)$. Now, we can reformulate our problem in the following way:

- You are given integer a. In one operation, you can change a to $2a \bmod p$ or $2a - 1 \bmod p$. Find the smallest number of operations needed to make a equal b.

This problem is easier to handle. Note that after the set of reachable numbers after k operations is the set of remainders modulo p of numbers from segment $[2^k a - (2^k - 1), 2^k a]$. For $k \geq 30$, the length of this segment will exceed p, so all remainders will be available. So, we just need to check k one by one, until we find the first one for which the segment $[2^k a - (2^k - 1), 2^k a]$ contains a number which equals to b modulo b. Asymptotics $O(\log p)$ per query.

Problem Tutorial: "Glory Graph"

There are 6 non-isomoprhic types of graphs on 4 vertices (here we put complete blue and complete yellow graphs into the same type). Here they are. Let's denote the numbers of times they appear as subgraphs of G as $x_1, x_2, x_3, x_4, x_5, x_6$ correspondently.



We will now try to find some conditions on these numbers which would help us to dedude $Y - A = x_6 - x_2$. From now on, we will do some double counting.

- There are $\frac{n(n-1)(n-2)(n-3)}{24}$ subgraphs of size 4 in total. So $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = \frac{n(n-1)(n-2)(n-3)}{24} = C_1$.
- Let's now double count the number of quadruples (A, B, C, D) of pairwise distinct vertices, where the edges AB and BC have different colors. From one side, it's equal to $0 \cdot x_1 + 8 \cdot x_2 + 12 \cdot x_3 + 12 \cdot x_4 + 16 \cdot x_5 + 16 \cdot x_6$. From other side, it's equal to $(n-3)$ by the sum of $2b_i y_i$ over all i, where b_i and y_i are the numbers of blue and yellow edges incident to vertex i respectively, and this sum can be calculated in $O(n^2)$. So, we can find $2x_2 + 3x_3 + 3x_4 + 4x_5 + 4x_6 = C_2$ in $O(n^2)$.
- Let's now double count the number of quadruples (A, B, C, D) of pairwise distinct vertices, where edges AB, BC, CD, DA, AC all have the same color. From one side, it's equal to $24 \cdot x_1 + 4 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 0 \cdot x_5 + 0 \cdot x_6$. From other side, it's equal to sum of $2cnt^1_{ij}(cnt^1_{ij} - 1)$ over all edges (i, j), where cnt^1_{ij} is the number of vertices k such that edges $(i, k), (j, k), (i, j)$ have the same color. We can calculate all cnt^1_{ij} in $O(\frac{n^3}{32})$ with bitsets. So, we can find $6x_1 + x_2 = C_3$ in $O(\frac{n^3}{32})$.
- Let's now double count the number of quadruples (A, B, C, D) of pairwise distinct vertices, where edges AB, BC, CD, DA all have the same color, which is different from the color of edge AC. From one side, it's equal to $0 \cdot x_1 + 4 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 8 \cdot x_5 + 0 \cdot x_6$. From other side, it's equal to sum of $2cnt^2_{ij}(cnt^2_{ij} - 1)$ over all edges (i, j), where cnt^2_{ij} is the number of vertices k such that edges $(i, k), (j, k)$ have the same color different from the color of edge (i, j). We can calculate all cnt^2_{ij} in $O(\frac{n^3}{32})$ with bitsets. So, we can find $x_2 + 2x_5 = C_4$ in $O(\frac{n^3}{32})$.

Now, we can write:

$$-3C_1 + C_2 + \frac{C_3}{2} - \frac{C_4}{2} =$$

$$= -3(x_1 + x_2 + x_3 + x_4 + x_5 + x_6) + (2x_2 + 3x_3 + 3x_4 + 4x_5 + 4x_6) + \frac{6x_1 + x_2}{2} - \frac{x_2 + 2x_5}{2} =$$
$$= x_6 - x_2$$

Note. While this may seem like a black magic, obtaining of this equation isn't that random: just write all the formulas you can with $x_1, x_2, \ldots, x_6$, and combine them to get $x_6 - x_2$, as the problem asks for that.

Problem Tutorial: "Hamiltonian "

For $K = 1$, it's just a line, for $K = 2$ it's just a graph on 4 nodes with edges $(1, 2), (2, 3), (3, 1), (3, 4)$.

For $3 \le K \le 20$ we can take a cycle of length K.

Now, consider a clique with n vertices where $n \ge 3$, select some nodes A and B there. Also, consider some chain of length $m \ge 2$, with ends C and D, and connect A to D and B to C.

This graph has exactly $n(n-1)/2-1+(m-1)+2(n-1)$ pairs of nodes between which there is Hamiltonian path. Those are:

- All pairs from the clique except pair (A, B): $\frac{n(n-1)}{2} - 1$ pairs
- Every 2 consecutive nodes in a chain: $2(n - 1)$ pairs
- All pairs (C, X) for X from clique except $X = A$ and all pairs (D, X) for X from clique except $X = B$: $2(m - 1)$ pairs

Luckily, all numbers from 21 to 60 can be presented as $n(n - 1)/2 - 1 + (m - 1) + 2(n - 1)$ for some $n \ge 3, m \ge 2, n + m \le 20$.

Problem Tutorial: "Intellectual Implementation"

Let's make a graph where we will connect two indices if corresponding rectangles intersect. Then, we need to compute number of anti-triangles (i. e. triples of vertices that are pairwise not connected with an edge) in this graph. Instead of doing this we will do three things:

1. Compute the degree of each vertex (for each rectangle we need to know how many other rectangles does it interesect).
2. Compute the number of triangles (the number of triples of rectangles which intersect pairwise).
3. Compute the answer to the original problem from this two values.

We will do 1 using sweepline. We will maintain a structure which helps us to answer the following queries:

1. Add a segment to the set.
2. Delete a segment from the set.
3. For given segment, calculate how many segments from the set it intersects.

This structure can be easily implemented using segment tree or Fenwick tree, since we can notice that answer to the third query for segment $[l, r]$ is equal to number of segments in a set minus number of segments with right border $< l$ minus number of segments with left border $> r$. So we maintain two segment trees (or Fenwick trees) for left and right borders, then queries are equivalent to point update and range query.

After this, let's sort our rectangles by l. Then, we will solve two independent problems:

1.1) Count the number of such j that $l_i > l_j$ and rectangle i intersects with rectangle j.

1.2) Count the number of such j that $l_i < l_j$ and rectangle i intersects with rectangle j.

Sum of answers for 1.1 and 1.2 will be degree of vertex i.

Note that for 1.1, for rectangle j, to intersect with i, we must have $l_i < l_j < r_i$ and $[d_i, u_i]$ intersecting with $[d_j, u_j]$. So we can iterate in the increasing order of x coordinate, add $[d_i, u_i]$ to set at the moment l_i. Then answer for i will be the number of segments intersecting with $[d_i, u_i]$ at the moment r_i minus number of segments intersecting with $[d_i, u_i]$ at the moment l_i.

1.2 is done in a similar way, but we should add segment to the set at the moment l_i and delete it at the moment r_i (since in this case we have the condition $l_j < l_i < r_j$).

To do 3 let's compute the following number — number of triples (x, y, z) such that (x, y) and (x, z) are both connected or both not connected with an edge. Here we suppose that order of y and z doesn't matter. We can count it in two ways. If we fix x and suppose that $deg_x = d$ than we should add $\binom{d}{2} + \binom{n-1-d}{2}$. On the other hand, we can notice that triangle and anti-triangle triples contribute 3 to this value and all other triples contribute 1. Since you know total number of triples(namely, $\binom{n}{3}$), you can find the final answer.

Now we are left with 2. For this, let's also do sweepline over x coordinate. Suppose that we are able to answer the following queries:

1. Add a segment to the set.
2. Delete segment from the set.
3. Find how many triples of segments intersect.

If we will be able to do this, subproblem 2 will be also solved easily — we will just iterate over x coordinate and for rectangle with X-segment $[l, r]$ we will add its Y-segment $[d, u]$ to the set at moment l and delete it from the set at moment r. Also, we will calculate the difference of number of intersecting segments before adding it and after adding it. Sum of this differences over all the rectangles will be equal to total number of intersecting triples of rectangles(basically each intersecting triple will be taken into account for the rectangle with the biggest value of l). So we are left with the problem of processing these queries.

To do this, let's suppose that we will maintain two arrays — val and val'. $val[i]$ will be equal to number of segments $[l, r]$ in a set, such that $l \le i \le r$. $val'[i]$ is similar, but we will have $l \le i < r$. Then, answer to 3 is equal to $\sum \binom{val[i]}{3} - \binom{val'[i]}{3}$.

Then, our following is equivalent, to the following one

1. Do range add query (in our problem, it's either $+1$ or -1 but it doesn't matter).
2. Calculate $\sum \binom{val[i]}{3}$.

To solve this final problem, we will maintain a segment tree. In each node we will store $\sum \binom{val[i]}{z}$, for $0 \le z \le 3$. The only thing that we need to do, is to be able to do lazy updates, so we should be able to compute $\sum \binom{val[i]+lazy}{t}$. We can do this using Vandermonde's Identity:

$$\binom{val[i] + lazy}{t} = \sum_{k=0}^{t} \binom{val[i]}{k} \cdot \binom{t}{k}.$$

It's also nice that it works even for negative values of lazy (if we expand the definition of binomial coefficients).

So, we can do lazy propagation operation in constant time. Merge operation is also trivial.

Finally, problem is solved in $O(nlog(n))$ time (with pretty high constant though).

One last note is that actually you can solve queries with maintaining $\sum \binom{val[i]}{z}$, for $0 \le z \le 2$ (since you just need to calculate how many pairs of segments intersect with $[l, r]$). If you maintain $\sum \binom{val[i]}{3}$, be aware of integer overflow.

Problem Tutorial: "Joke"

Firstly, $f(p, q)$ clearly depends only on the order of pairs (p_i, q_i), so we can assume that $p_i = i$ initially.

Lemma. $f((1, 2, \ldots, n), q)$ = number of increasing subsequences of q.

Proof. Let's first analyze when the string s satisfies p, q. Basically, we have n nodes corresponding to upper row, n nodes corresponding to lower row, and some directed edges (u, v) between them, indicating that the number in cell u has to be smaller than the number in cell v. The necessary and sufficient condition for being able to put numbers from 1 to $2n$ into this nodes so that all relations are satisfied is: **There has to be no directed cycle.** We will show that in case of our graph, it's equivalent to the following: **There exists no directed cycle of size** 4.

Indeed, consider the directed cycle of the smallest length, suppose that its size is larger than 4. It has to contain some edge between nodes from two different rows, as there can't be any cycle inside a single row. Wlog it's an edge from cell $(1, i)$ to $(2, i)$. There has to be an edge from $(2, i)$ somewhere now, wlog to $(2, j)$. Finally, if the edge from $(2, j)$ goes to $(2, k)$, we could have obtained a shorter cycle by just removing $(2, j)$ from it, as there is an edge $((2, i), (2, k))$, so the edge from it goes to $(1, j)$. Now, if $p_i < p_j$, then we can replace the path $((1, i), (2, i), (2, j), (1, j))$ by just $((1, i), (1, j))$, otherwise we have obtained a cycle of size 4.

So, it's enough to ensure that there are no directed cycles of size 4. Let's find the number of strings s for which it's the case. Consider i for which $q_i = n$. If we set s_i to `0`, we can forget about pair (p_i, q_i), as it can't be involved in any cycle of length 4. Otherwise, we get that the number in the cell $(1, i)$ of the matrix is bigger than the largest number in the second row, so for each $j > i$, the number in cell $(1, j)$ is also bigger than in cell $(2, j)$. Therefore, if we set s_i to `1`, we also have to set all s_j with $j > i$ to `1`. After that, we can throw out all pairs (p_j, q_j) for $j \ge i$, as there wouldn't be able to get involved in any cycles.

So, we have an array q, and 2 operations:

- Delete the largest element
- Delete the largest element and all elements to the right of it.

It's easy to show that the number of ways to delete the entire q by applying these operations in some order is equal to the number of increasing subsequences of q. Indeed, each such sequence of operations corresponds to the subsequence of numbers to which we will apply 2-nd operation, when they are the largest.

Lemma is proved

Now, we have the following problem:

- We are given some elements of permutation q, and others are missing. Find sum of $f(q)$ over all valid permutations q (meaning that they have the given elements at the right places).

Under $n \le 100$, it's an easy problem. Set $q_0 = 0$ and $q_{n+1} = n + 1$, now $f(q)$ is the number of increasing subsequences starting at q_0 and ending at q_{n+1}. For every element that's already set, say q_i, calculate $dp[i][k]$ — the number of possible increasing subsequences starting at q_0 and ending at q_i, which contain exactly k **unset** elements.

Here are the transitions: for every $j < i$ such that q_j is also set and $q_j < q_i$, we calculate the number of "free"positions between jth and ith, and the number of "allowed"elements — the elements

from $[q_j + 1, q_i - 1]$, which aren't set as elements already. Then, for every *choose* not exceeding $max(free, allowed)$ and every *chosen*, add $dp[j][chosen] \times (\binom{choose}{allowed} \times \binom{choose}{free})$ to $dp[i][chosen + choose]$.

The answer to the problem is then just the sum of $dp[n+1][x] \times (n - set - x)!$ over x, where *set* is the number of already set elements.

Problem Tutorial: "K-onstruction"

Consider a set S of integers, in which there are exactly K subsets with sum 0, in which there are no zeros, and in which sum of all elements is not zero. Let P and N be the sums of all positive and of all negative elements of the array correspondently, wlog $P > -N$. Let's add some nonzero elements divisible by P to this set, denote the set of these added elements by T for now.

Let's look at $S \cup T$. How many subsets with zero sum are there in it? The part we take from T is divisible by P, so from S we also have to take part divisible by P. As $P > -N$, there are only 2 ways to do so: to take sum P by choosing all positive elements (in exactly one way), or to take sum 0 in K ways.

So, the number of subsets with sum 0 in $S \cup T$ is equal to $K \times$(number of subsets with zero sum in T) + (number of subsets with sum $-P$ in T). Note that the set $S \cup T$ also satisfies the conditions for S: all elements are nonzero, and sum of all elements is not zero (as it's not divisible by P).

Now, let's generate some small sets S and see what pairs (number of subsets with sum 0, number of subsets with sum 1) they produce. If for set of size n there are cnt_0 subsets with sum 0 and cnt_1 subsets with sum 1, we have a transition from (len, K) to $(len + n, cnt_0 K + cnt_1)$.

Based on these generated transitions, calculate dp array, where $dp[n]$ denotes the smallest length needed to get exactly n subsets with sum 0, and save the info by which transitions we should go to.

It turns out that generating all sets with integers from $\{-3, -2, -1, 1, 2, 3\}$ with size at most 10 is enough to make all values of dp up to 10^6 less or equal to 30, and this fits without any optimizations. We can also fit 29 easily, and will have to optimize quite a lot to fit into 28, so we decided to make the bound on size of the array 30.

Problem Tutorial: "Little LCS"

First let's note that LCS is always at least n. Indeed, for every i, there is some character that's present in both (s_{2i-1}, s_{2i}) and (t_{2i-1}, t_{2i}). It hints that awesome strings must have some very special structure.

And indeed, the problem is mainly about understanding the structure of awesome strings, the rest is just implementing straightforward checking.

It turns out that there are 2 classes of pairs of awesome strings:

- One string has form `ABABAB...ABA`, and the second one has `C` at all odd positions and at every even position has one of `A`, `B`. (And, obviously, same thing over all other permutations of `A`, `B`, `C`).

  It's obvious that strings of such format have LCS of length n: it can't be higher as LCS can't contain any of $n + 1$ `A`s of s.

- Both strings have `C` at all even positions, and for some k, first k odd positions of s contain `A`, last $n+1-k$ odd positions of s contain `B`, first k odd positions of t contain `B`, last $n+1-k$ odd positions of t contain `A`. (And, obviously, same thing over all other permutations of `A`, `B`, `C`).

  Here it's a bit harder to see why these strings have LCS n, but it can be proved by induction. Indeed, suppose that both s and t contain some string lcs of length $n + 1$ as a subsequence. If $lcs[1] =$`C`, then $s[3 : 2n + 1]$ and $t[3 : 2n + 1]$ both have to contain $lcs[2 : n + 1]$, but their LCS is $n - 1$ by induction assumption, so $lcs[1] \neq$`C`. Similarly, $lcs[n + 1] \neq$`C`.

  Wlog $lcs[1] =$`A`. Then $lcs[n+1]$ can't be `B`, as t doesn't contain `AB` as a subsequence, so $lcs[n+1] =$ `A`. But this means that $|lcs| \leq min(2k - 1, 2(n + 1 - k) - 1)$. As $(2k - 1) + (2(n + 1 - k) - 1) = 2n$, we get $|lcs| \leq n$.

During the contest, you can observe this pattern by just bruteforcing amazing strings for $n \le 4$, and don't have to prove it, but for the completeness of the editorial we will provide the proof here.

So, suppose that (s, t) is a pair of amazing strings of length $2n+1$. Let's show that they are of one of the types above, by induction by n. Base for $n = 1$ can be checked by hand, now we suppose that $n \ge 2$ and the statement is proved for all $n_1 < n$.

Note that $(s[1 : 2n-1], t[1 : 2n-1])$ is an amazing pair of strings for $n-1$. Therefore, they are one of the types above. Same goes for $(s[3 : 2n+1], t[3 : 2n+1])$.

Suppose that $(s[1 : 2n-1], t[1 : 2n-1])$ is an amazing pair of the first type. Wlog $s[1 : 2n-1] =$ `ABAB...ABA`, and $[1 : 2n-1]$ has `C` at all odd positions. $s[2n-1 : 2n+1]$ and $t[2n-1 : 2n+1]$ have to have LCS 1, and there are only 4 such pairs of strings: (`ABA`, `CBC`), (`ABA`, `CAC`), (`ABC`, `CBA`), (`ACA`, `CBC`). Note that the first two pairs correspond to the pattern of amazing strings of type 1. Let's look at last two cases.

Suppose that $(s[2n-1 : 2n+1], t[2n-1 : 2n+1]) =$ (`ABC`, `CBA`). Then $(s[3 : 2n+1], t[3 : 2n+1])$ can't be an amazing pair of type 1, so it's an amazing pair of type 2, and all letters at even positions at them are `B`. So, we get $s =$`ABAB...ABABC` and $t =$`C?CBC...CBCBA`. If `?` is replaced with `B`, we get an amazing pair of second type, otherwise it's replaced with `A` and there is a common subsequence of length $n+1$ `ABB...BBA`.

Now suppose that $(s[2n-1 : 2n+1], t[2n-1 : 2n+1]) =$ (`ACA`, `CBC`). Then $(s[3 : 2n+1], t[3 : 2n+1])$ can't be an amazing pair of type 2, so it's an amazing pair of type 1. As not all letters at even positions at $s[3 : 2n+1]$ are the same, all letters at even positions of $t[3 : 2n+1]$ must be the same, so $t[3 : 2n+1] =$ `CBCB...CBC`. Therefore, we get $s =$`ABAB...ABACA` and $t =$`C?CBC...CBCBC`. If `?` is replaced with `B`, we get an amazing pair of the first type, otherwise it's replaced with `A` and there is a common subsequence of length $n+1$ `ABB...BBC`.

In this case, we proved the statement. Now suppose that $(s[1 : 2n-1], t[1 : 2n-1])$ is an amazing pair of the second type. Similarly, $(s[3 : 2n+1], t[3 : 2n+1])$ is an amazing pair of the second type. Then, there are two cases: or (s, t) is also an amazing pair of the second type, or the strings have form (`ACBCB...CBCBCA`, `BCACA...CACACB`). But in this case they both contain a string `ACCC...CCB` of length $n+1$ as a subsequence.

We checked all the cases, so congrats to us.

Problem Tutorial: "Math"

Note that if $k^2 = a_i^2 + a_j$, then $(k - a_i)(k + a_i) = a_j$. We can iterate over all pairs of numbers x and y such that $x \cdot y \le max(a)$ and check if they produce valid pair of a_i, a_j. It will happen iff numbers $x \cdot y$ and $\frac{x-y}{2}$ both lie in our array. It's well known that number of such pairs is $O(max(a)log(max(a)))$.