The 2019 Southeastern Europe Regional Contest Editorial

Lorina Negreanu (lorina.negreanu@cs.pub.ro) & Anton Tsypko (tsypko@oi.in.ua)

Problem Editorial: "Max or Min"

Author: **Roman Bilyi**
Developer: **Roman Bilyi**
Editorialist: **Stanislav Bezkorovainyi**

Before solving the problem for all k from 1 to m, let's try to solve a problem for a fixed number k.

It's obvious, that if there is no element of array a that is equal to k, then the answer is -1.

Otherwise, the answer always exists. Let's prove it.

At first, let's replace all the elements in the array by the following rule:

- If $a_i < k$, then $a_i := -1$ (replace a_i with -1)
- If $a_i = k$, then $a_i := 0$ (replace a_i with 0).
- If $a_i > k$, then $a_i := 1$ (replace a_i with 1)

Now, the problem is equivalent to the previous one, we just have to make elements of the array equal to 0.

Let's call *zebra* a subarray of the array a that consists of altering numbers -1 and 1. Examples of zebras are: $[-1]$, $[1]$, $[-1, 1, -1]$, $[1, -1]$, $[1, -1, 1, -1, 1, -1, 1]$, $[-1, 1, -1, 1, -1, 1, -1, 1, -1]$. Note, that since the array is arranged in a circle, arrays like, for example, $[1, -1, -1, -1]$ have zebra $[-1, 1, -1]$ that starts from the last element.

Since the elements are arranged in a circle, then cyclical shifts do not change it. Let's make necessary number of cyclical shifts to left, so that the first element of the array is equal to 0. Now we know for sure, that there are no zebras that come through index 1. It helps, because it ensures that finding all zebras in our cyclical string is now equivalent to finding all zebras in the string after the shifts. Why do we even need to find these zebras? We'll come to that later.

Let's define $Z(a)$ as a multiset of lengths of the longest sequential zebras. How we build it:

At first we stick the pointer p at position 1. Then the following algorithm works until $p \le n$:

- If $a_p = 0$, then increase p by 1. Repeat the algorithm.
- Otherwise, let len be the length of the longest possible zebra that starts from position p. Add len to $Z(a)$ and increase p by len. Repeat the algorithm.

For example, if a was $[k, -1, 1, -1, -1, 1, k, -1, -1]$, then $Z(a) = \{3, 2, 1, 1\}$ (the longest zebra that starts on position 2 has length 3, the longest zebra that starts on position 5 has length 2, the longest zebras that start on positions 8 and 9 have length 1).

Let's define function $cnt(a, c)$ that returns number of occurrences of c in the array a. Let's define $f(a)$ as a function that takes the array a as an argument and returns the following value:

$$f(a) = cnt(a, -1) + cnt(a, 1) + \sum_{z \in Z(a)} \left\lfloor \frac{z}{2} \right\rfloor$$

In other words, $f(a)$ equals to number of elements that are not equal to 0 plus the sum of floored lengths of zebras divided by two.

Claim. The answer is equal to $f(a)$.

Those, who don't want to read lengthy proofs can skip to **How to implement it?** section. For those, who want to read the proof, here it is:

Proof.

- **Claim**. $f(a) = 0$ if and only if all the elements of the array are equal to k.

  Proof. If all the elements are equal to 0, then $cnt(a, -1) = cnt(a, 1) = 0$ and there are definitely no zebras in the array, so $f(a) = 0$. It works in the other direction too: if $f(a) = 0$, then $cnt(a, -1) + cnt(a, 1) = 0$, so there are no elements that are not equal to 0, i.e. all of the elements of the array are equal to 0.

  What it gives us? It means that our problem is equivalent to problem of reducing value of $f(a)$ of our initial array to 0 in as few operations as possible.

- **Claim**. If we can transform our array a into array b in one operation, then $f(b) \geq f(a) - 1$. In other words, we can not reduce the value of the function more than by one in an operation.

  Proof. At first, we should notice, that an operation that changes value of the element that is equal to 0 only increases value of $cnt(a, -1) + cnt(a, 1)$, thus $f(b) > f(a)$ if b was obtained by changing value of an element in the array that is equal to 0.

  Now it is left to prove that changing value of an element that is equal to either -1 or 1 can not reduce the value of the function more than by one. I'll show you why changing an element that is equal to 1 can not do this. The proof for -1 is very similar.

  Let's suppose that $a_i = 1$ for some i. Now let's look at all possible triples $[a_{i-1}, a_i, a_{i+1}]$ (since the array is cyclical $a_0 = a_n$, $a_{n+1} = a_1$). There are 9 possible triples:

  - $[1, 1, 1]$. We can not change a_i at all.
  - $[-1, 1, 1]$. We can make a_i equal to -1. Even though a_i might belong to a zebra, it is easy to proof that changing it to -1 won't affect $f(a)$ by more, than 1.
  - $[1, 1, -1]$. The same as the previous one.
  - $[-1, 1, -1]$. We can make a_i equal to -1. It doesn't change $cnt(a, -1) + cnt(a, 1)$, but it is obvious that a_i belongs to some zebra that starts on position $p < i$. By setting value of a_i as -1 we "split" this zebra into two zebras which start at positions p and $i+1$. Let's say that their lengths are len_1 and len_2 respectively. The length of the zebra before the split is $len_1 + len_2 + 1$ (it included a_i). Now, $f(b) = f(a) - \left\lfloor \frac{len_1+len_2+1}{2} \right\rfloor + \left\lfloor \frac{len_1}{2} \right\rfloor + \left\lfloor \frac{len_2}{2} \right\rfloor$. It is easy to prove, that $\left\lfloor \frac{len_1+len_2+1}{2} \right\rfloor - \left\lfloor \frac{len_1}{2} \right\rfloor - \left\lfloor \frac{len_2}{2} \right\rfloor$ is either equal to 0 or 1. In order to prove this, you should test how expanding flooring brackets will work for cases, when: len_1 is even and len_2 is even, len_1 is even and len_2 is odd, len_1 is odd and len_2 is even, and both len_1 and len_2 are odd.
  - $[0, 1, 1]$. We can make a_i equal to 0. It will only affect $cnt(a, 1)$, because of that $f(b) = f(a) - 1$.
  - $[1, 1, 0]$. The same as the previous one.
  - $[0, 1, 0]$. We can make a_i equal to 0. $f(b) = f(a) - 1$ as well.
  - $[-1, 1, 0]$. We can make a_i equal to -1. It will not affect $cnt(a, -1) + cnt(a, 1)$. It is also obvious that a_i belongs to some zebra. Let's suppose that it's length is len. $f(b) = f(a) - \left\lfloor \frac{len}{2} \right\rfloor + \left\lfloor \frac{len-1}{2} \right\rfloor$. It is obvious, that $\left\lfloor \frac{len}{2} \right\rfloor - \left\lfloor \frac{len-1}{2} \right\rfloor$ is equal to either 0 or 1.
  - $[0, 1, -1]$. The same as the previous one.

  What it gives us? Now we know that $f(a)$ is the lower bound of number of operations needed to make all the elements of the array equal to 0.

- **Claim**. It is always possible to solve the problem in $f(a)$ operations.

  Proof. We do it using the following algorithm:

  1. Look through all positions i such that $a_i = 0$ and $a_{i-1} \neq 0$. If there are no such positions, then all the elements of the array are equal to 0 and thus the algorithm should stop. If such position i exists, then proceed to step 2.
  2. If $[a_{i-2}, a_{i-1}, a_i]$ is equal to either $[0, -1, 0]$, $[0, 1, 0]$, $[-1, -1, 0]$ or $[1, 1, 0]$, then we set value a_{i-1} as k, which will obviously decrease the value of $f(a)$. After that, return to step 1. If the array is not equal to neither of the mentioned above, then don't do anything and go to step 3.

3. Now we know, that $[a_{i-2}, a_{i-1}, a_i]$ is equal to either $[1, -1, 0]$ or $[-1, 1, 0]$. It is obvious, that a_{i-2} belongs to some zebra of length greater than 1. By applying min or max operation respectively we can make a_{i-2} have same value as a_{i-1}.

   If the zebra's length is greater, then 2, then after making a_{i-1} and a_{i-2} equal, $a_{i-3}, a_{i-2}, a_{i-1}$ will have same values, thus, the zebra will no longer contain elements a_{i-2} and a_{i-1}, so it's length will be decreased by 2, so it's length divided by two will be decreased by one. This is why $f(a)$ will be decreased by 1.

   If the zebra's length is equal to 2, then, obviously, the zebra contains only elements a_{i-1} and a_{i-2}. If they are equal, then the zebra no longer exists at all. Thus, $f(a)$ will be decreased by 1.

   After making a_{i-2} equal to a_{i-1}, return to step 1.

What it gives us? Now we know, that $f(a)$ is the lower bound of our answer and it is always achievable. Thus, $f(a)$ is the answer.

How to implement it? It is obvious how to find $f(a)$ in $O(n)$ for a fixed k. But we need to solve the problem for all k from 1 to m.

Let's remember for each value from 1 to m on which positions they occur. And set all $a_i := 1$.

Let's define arrays p, len and value $sumz$:

- Let's say that an element on a position i belongs to some zebra, then the value of p_i is the position from which the zebra starts. If $a_i = 0$, then $p_i = i$.
- If there is a zebra that starts from position i, then the value of len_i is equal to length of that zebra. If no zebra starts from position i, then $len_i = 0$.
- $sumz$ is equal to $\sum_{z \in Z(a)} \lfloor \frac{z}{2} \rfloor$, assuming that our array is not cyclical (i.e we do not count zebras that contain both positions 1 and n).

Let's also define procedure $upd(pos, val)$. $upd(pos, val)$ changes value of a_{pos} to val in a way that sustains all the arrays p and len and value $sumz$ valid.

Now, let's make all $p_i := 1$ and thus all $len_i := 1$, $sumz := 0$. Now, we iterate k from 1 to m, solving the problem for every possible k in the following way:

- If the value k has no occurences, then the answer for this k is -1.
- Otherwise, proceed:
  - Now, we should iterate over all positions i, where k occurred and call $upd(i, 0)$.
  - Now, answer ans_k for k is equal to number of positions, where k doesn't occur plus $sumz$. But we need to make sure the zebra that contains both elements on positions n and 1 (if it exists) is counted too. If $a_1 = 0$ or $a_n = 0$ or $a_n = a_1$, then the zebra doesn't exist at all, so we don't need to modify ans_k. Otherwise, the zebra exists and we need to modify ans_k in the following way: $ans_k := ans_k - \lfloor \frac{len_1}{2} \rfloor - \lfloor \frac{len_{p_n}}{2} \rfloor + \lfloor \frac{len_1 + len_{p_n}}{2} \rfloor$. This modifying is equivalent to deleting zebras that start on positions 1 and p_n (this is the zebra that contains n) and adding zebra of length $len_1 + len_{p_n}$.
  - Now, we should iterate over all positions i, where k occurred and call $upd(i, -1)$.

  This algorithm will always work, because, since we iterate k in ascending order, once position is assigned -1 we will never have to change it's value again.

The only question is how to implement $upd(pos, val)$. In order to simplify it's implementation, I will use the fact if $val = 0$, then a_{pos} previously was 1, and if $val = -1$, then a_{pos} previously was 0.

- If $val = 0$, then do the following:
  - If $p_{pos} + len_{p_{pos}} - 1 = pos$ (i.e the zebra, that contains position pos ends on it), then we should decrease $sumz$ by $\lfloor \frac{len_{p_{pos}}}{2} \rfloor - \lfloor \frac{len_{p_{pos}-1}}{2} \rfloor$ and decrease $len_{p_{pos}}$ by one.
  - Otherwise, the zebra doen't end on position pos, so transforming a_{pos} into k will split it. Decrease $sumz$ by $\lfloor \frac{len_{p_{pos}}}{2} \rfloor - \lfloor \frac{pos-p_{pos}}{2} \rfloor - \lfloor \frac{p_{pos}+len_{p_{pos}}-1-pos}{2} \rfloor$ (basicaly, we just split the zebra). Assign $p_i := pos + 1$ for all i from $pos + 1$ to $p_{pos} + len_{p_{pos}} - 1$. So now, we set $len_{pos+1} := p_{pos} + len_{p_{pos}} - 1 - pos, len_{p_{pos}} := pos - p_{pos}$.
  - In the end, in both cases, set $p_{pos} := pos, len_{pos} := 0$, $a_{pos} := 0$.
- If $val = -1$, then do the following:
  - At first, set $a_{pos} := -1, len_{pos} := 1, p_{pos} := pos$.
  - If $pos < n$ and $a_{pos+1} = 1$, then there is a zebra that starts form position pos that contains zebra that starts from position $pos + 1$ as a suffix. Increase $sumz$ by $\lfloor \frac{len_{pos+1}+1}{2} \rfloor - \lfloor \frac{len_{pos+1}}{2} \rfloor$. Set $p_i := pos$ for all i from $pos + 1$ to $pos + len_{pos+1}$. Set $len_{pos} := len_{pos+1} + 1, len_{pos+1} := 0$.
  - If $pos > 1$ and $a_{pos-1} = 1$, then there is a zebra that starts from position p_{pos-1} that has zebra that starts from position pos as a suffix. Increase $sumz$ by $\lfloor \frac{len_{p_{pos-1}}+len_{pos}}{2} \rfloor - \lfloor \frac{len_{p_{pos-1}}}{2} \rfloor - \lfloor \frac{len_{pos}}{2} \rfloor$. Set $p_i := p_{pos-1}$ for all i from pos to $pos + len_{pos} - 1$. Set $len_{p_{pos-1}} := len_{p_{pos-1}} + len_{pos}, len_{pos} := 0$.

In order to assign some value on a segment in $O(\log n)$ you can use Segment Tree (you can read more about this data structure here `https://cp-algorithms.com/data_structures/segment_tree.html`). Since $upd(pos, val)$ is called exactly 2 times for each pos, then the procedure is called $O(n)$ times. Every call is done in $O(\log n)$, thus, the final time complexity is $O(n \log n)$.

Problem Editorial: "Level Up"

Author: **Stefan Ruseti**
Developer: **Adrian Budau**
Editorialist: **Adrian Budau**

The first idea would be to go through the quests in order and pick whether or not we solve that quest at the first level or at the second level.

We can use dynamic programming for this: $dp[i][j][k]$ to be at quest i and have accumulated j experience at the first level and k experience at the second level.

When we are at a state $dp[i][j][k]$ and are deciding what to do with quest $i+1$ we might run into this special case: we try to solve quest $i+1$ in the first level but we overflow the experience into the second level ($j + x_i > S1$). We must remember to use the overflow experience at the second level. (so $dp[i][j][k] -> dp[i+1][S1][k + (j + x_i - S1)]$).

Unfortunately, we don't exactly try all possible orders to do the quests, and especially the quest that triggers the overflow. We solve quests independently at each level in the order we process the quests. But, there is a simple observation to be made: if in a correct solution we solve the quests $(i_1, i_2, i_3, ..., i_K)$ at the first level in this particular order, we could reorder quests $i_1, i_2, ..., i_{K-1}$ and get a different solution. We can not exactly pick any quest to be the last at level 1.

For example if we have 4 quests we want to do at the first level, 3 giving 30 experience each and 1 giving 90 experience and $S1 = 100$ the only way to solve them at the first level is to first solve the 3 quests giving 30 each (therefore accumulating 90 experience) and then solving the 90 experience quest. We can see that if we can overflow using a quest giving U experience from the list $i_1, i_2, \ldots, i_K$ than we can overflow with any quest giving $V \geq U$ experience.

So that means we could set the order of solving the quests: ascendant order by their first level experience (x_i). Now we can use the dynamic programming solution describe above with no other modification.

Total complexity: $\mathcal{O}(n \cdot s1 \cdot s2)$ time and $\mathcal{O}(n \cdot s1 \cdot s2)$ or $\mathcal{O}(s1 \cdot s2)$ (depending on implementation) memory.

Problem Editorial: "Find the Array"

Author: **Anton Trygub**
Developer: **Stanislav Bezkorovainyi**
Editorialist: **Stanislav Bezkorovainyi**

If $n \leq 30$, that we can simply find the array using 30 queries of type 1. Otherwise we should do the following:

At first, we should find the position of maximum or minimum element of the array. Let's define those positions as w and u respectively. Now, use query of type 2 on the whole array. Let's denote the maximum element of the resulting array as M. It is obvious that $M = a_w - a_u$.

What does it mean if the maximum element of the resulting array on query of type 2 for some subset of indexes S is equal to M? It means that both w and u belong to S. Now, we can using binary search find the leftmost position pos such that the maximum element of the resulting array on the query of type 2 for set of indexes $\{1, 2, \ldots, pos\}$ is equal to M.

Since both u and w belong to $\{1, 2, \ldots, pos\}$ and one of them does not belong to $\{1, 2, \ldots, pos - 1\}$, u or w is equal to pos. For now, it does not really matter whether $u = pos$ or $w = pos$. What matters is that, since all a_i are distinct and that the value of a_{pos} is whether smaller or greater than value of any a_i, there is no such pair of indexes $(j, i), i \neq j$ that $|a_{pos} - a_i| = |a_{pos} - a_j|$.

Let's denote $b_i = |a_{pos} - a_i|$. It's obvious, that $b_{pos} = 0$. Now we should find b_i for every $1 \leq i \leq n, i \neq pos$. All b_i will be distinct.

At first, let's define function $F(Q) = B$ that takes some set of indexes Q, such that $pos \notin Q$ as an argument and returns set B that contains all b_i for any $i \in Q$. How to implement such function in your program? That's quite simple. At first, let's denote set A as the result of query 2 on set of indexes Q and set A' as the result of query 2 on set $Q \cup \{pos\}$. Then, $F(Q) = A' \setminus A$ (All elements, that belong to A', but do not belong to A). Since all b_i are different, size of set Q will always be equal to size of $F(Q)$.

Now, let's use divide and conquer approach.

Let's say that we have some set of positions Q and set $B = F(Q)$. We want to know b_i for every $i \in Q$. There are two possible scenarios:

- Size of Q is equal to 1. Then b_i is the only element of set B, while i is the only element of set Q. Thus, we know b_i without any additional queries.
- We split Q into two sets S_1, S_2 such that first $\lfloor \frac{|Q|}{2} \rfloor$ elements of set Q go into set S_1 and the rest of them go into S_2. We can find $B_1 = F(S_1)$, $B_2 = F(S_2)$ in one call of function F: $B_1 = F(S_1)$, $B_2 = B \setminus B_1$. Now, we should solve the problem independently for sets S_1, B_1 and S_2, B_2.

At the beginning of the algorithm, Q contains all indexes except pos and B is calculated in a straightforward way $B := F(Q)$.

A split of a set takes 1 call of F, but wait a second... There are $\approx n$ splits. It is no better than simply asking value for every index with query 1! Yeah, I know. But we can optimize it easily into $O(log_2 n)$ calls of F.

It is easy to notice that splits on the same depth of our "divide and conquer binary tree" can be done simultaneously with only only call of function F. Let's say that there are k sets $Q_1, Q_2, \ldots, Q_k$ about to be split on the same depth. Their first halfs are $S_{1_1}, S_{2_1}, \ldots, S_{k_1}$. Let's define $A = F(S_{1_1} \cup S_{2_1} \cup \ldots \cup S_{k_1})$ (Function of union of all S_{i_1}). Now, for every $1 \leq i \leq k$ $B_{i_1} = B_i \cap A, B_{i_2} = B_i \setminus B_{i_1}$.

Depth of the tree is at most $\lceil log_2 n \rceil$. So that we will make at most $\lceil log_2 n \rceil$ calls of function F + one call in order to get $F(Q)$ for the initial set. Every call of F takes exactly 2 queries, so we will spend $2 \cdot \lceil log_2 n + 1 \rceil$ queries on this part of the program.

Now, we know b_i for every $1 \leq i \leq n$. Let's define pos_2 as such index that $b_{pos_2} = \max(b_1, b_2, \ldots, b_n)$. Since the minimal element has the greatest difference with the maximal element and vice versa, if a_{pos} is the minimal element, then a_{pos_2} is the maximal one and vice versa. Let's use query 1 to ask the value of a_{pos} and the value of a_{pos_2}. There are two possible scenarios:

- $a_{pos} > a_{pos_2}$. That means a_{pos} is the maximal element of the array a and thus $a_i = a_{pos} - b_i$ for every $1 \leq i \leq n$.
- $a_{pos} < a_{pos_2}$. That means a_{pos} is the minimal element of the array a and thus $a_i = a_{pos} + b_i$ for every $1 \leq i \leq n$.

Overall, we used $1 + \lceil log_2 n \rceil$ queries to find pos, $2 \cdot \lceil log_2 n \rceil + 2$ queries to find b_i for every i and 2 queries to find values of a_{pos} and a_{pos_2}.

$$1 + \lceil log_2 n \rceil + 2 \cdot \lceil log_2 n \rceil + 2 + 2 = 5 + 3 \cdot \lceil log_2 n \rceil = 5 + 3 \cdot \lceil log_2 250 \rceil = 29$$

Note. Be careful while implemeting binary search and function F. Otherwise, you may find yourself getting `Wrong Answer` due to very strict constraints.

Problem Editorial: "Cycle String?"

Author: **Anton Trygub**
Developer: **Nazarii Denha**
Editorialist: **Stanislav Bezkorovainyi**

Let's say, that cnt_{ch} is the number of occurrences of character ch in the string. Let's say that cm is such a character, that cnt_{cm} is maximum among all cnt_{ch}.

Now, there are several possible scenarios.

Scenario 1. $cnt_{cm} = 2 \cdot n$. In other words, the string consists of the same characters. Obviously, the answer in that case is "NO".

Scenario 2. $cnt_{cm} = 2 \cdot n - 1$ and $n \geq 2$. In other words, the string consists of the same characters, except for one position. The answer in that case is "NO".

Proof. Let's say that $cm =$ "a" and the only symbol not equal to cm is equal to "c". Our string will look like this:

$$aaaaaaaaaa \ldots aaaaacaaaaa \ldots aaaaaaaaaa$$

Since the string is cyclical, then cyclical shiftings of the string do not change it. Let's shift it the way that the only character that differs from the rest is placed on the right side of the string. Now the string looks like this:

$$\overbrace{a\underbrace{aaaaa \ldots aaaaaaa}_{n \text{ characters}}}^{n \text{ characters}} aaaa \ldots aaaaac$$

Scenario 3. $cnt_{cm} = 2 \cdot n - 2, n \geq 3$ and the characters besides cm are the same. The answer in that case is "NO".

Proof. Let's say that $cm =$ "a" and the rest of the characters are equal to "c". Now we shift the string with cyclic shifts in such a way that the first character of the sting is "c". Let's denote the other position on the string, where character "c" is placed as p. Now, our string looks like this:

$$\overbrace{caaaaa \ldots aaac}^{p \text{ characters}} aaa \ldots aaaaaa$$

It is obvious that $p - 2 < n + 1$. Otherwise, there would have been at least two strings of length n containing only characters "a" between 1 and p. For similar reason $2 \cdot n - p < n + 1$.

$$p - 2 < n + 1 \iff p < n + 3$$

$$2 \cdot n - p < n + 1 \iff n - p < 1 \iff p > n - 1$$

Now we know that p is either equal to n or $n + 1$ or $n + 2$.

Why not n? The substring t is equal to concatenation of string g_1 and g_2 (remember, our string is cyclical, so everything is fine):

$$\underbrace{ca}_{g_2}\overbrace{aaaa \ldots aaaca}^{t}\underbrace{aaaaaa \ldots aaaaaa}_{g_1 = n-2 \text{ same symbols}}$$

Why not $n + 1$? The substring t is equal to substring g:

$$\overbrace{caaaaa\ldots aaa}^{t}\overbrace{caaaaa\ldots aaa}^{g}$$

Why not $n+2$? We can make $n+1$ cyclic shifts to left and after that we'll get the same situation as when $p = n$.

So there is no valid way to place p, so the answer is "NO".

Scenario 4. $cnt_{cm} = 2 \cdot n - 2, n \geq 3$ and the characters besides cm are distinct. In this case the answer is "YES".

Proof + the way to build such string. Let's say that $cm =$ "a" and the other two characters are equal to "c" and "d". We can build the answer this way:

$$\overbrace{caaa\ldots aaa}^{n \text{ symbols}}\overbrace{daaa\ldots aaa}^{n \text{ symbols}}$$

It is obvious that every substring will contain either "c" or "d" but not both of them. Since there is only one character "c", any two different substrings that contain "c" will have it on different positions and thus will be distinct. The same goes for any two substrings that contain "d".

Scenario 5. $cnt_{cm} \leq n$. In this case the answer is "YES".

Proof + the way to build such string. We can simply sort the string.

Let's suppose that it does not work and there are two identical substrings s_i and s_j of length n that start on positions i and j respectively, where $i < j$. Let's denote the first character of s_j as c. Let's say that at the beginning of the string s_j there are $p \leq n$ characters c. Since $s_i = s_j$ same should go for s_i. If character at position $j + p - 1$ is equal to c too, then, since the string is sorted, there are at least $j + p - i > p$ characters c at the beginning of the string s_i. We face contradiction and thus, the statement is false, i.e there are indeed no same substrings that start on different positions.

Scenario 6. $n < cnt_{cm} \leq 2 \cdot n - 3$. In this case the answer is "YES".

Proof + the way to build such string. Let's say that $cm =$ "a" and the rest of the symbols are "b", "c", "d", ..., "z". We can put an arbitrary symbol besides cm on the first position, then n symbols cm (let's call it $block_1$), then all the remaining symbols besides cm in any order (let's call it $block_2$), then $cnt_{cm} - n$ symbols cm (let's call it $block_3$):

$$b\overbrace{aaaaaaaaaaa\ldots aaaaaaaaaaa}^{block_1 \text{ — } n \text{ symbols } cm}\overbrace{zbtbcydcceecgy\ldots vwqxheyqyui}^{block_2 \text{ — the rest of the symbols}}\overbrace{aaaaaaaaaaa\ldots aaaaaaaaaaa}^{block_3 \text{ — } cnt_{cm}-n \text{ symbols } cm}$$

We should notice, that since $cnt_{cm} \leq 2 \cdot n - 3 \iff cnt_{cm} - n \leq n - 3$. And thus length of $block_3$ is smaller than $n - 2$.

Why all of the substrings of length n will be unique?

- Substring that starts on position 1 is the only substring that starts from a symbol that is not equal to cm and is followed by $n - 1$ characters cm.
- Substring that starts on position 2 is the only substring that consists only of characters cm.
- Substring that starts on position 3 is the only substring that has $n - 1$ symbols cm at the beginning and one symbol that is not equal to cm in the end.
- Substrings that start on positions from 4 to $n + 1$ are the only substrings that start from cm and they have a substring of length ≥ 2 that consists of characters that are not equal to cm. They will differ among themselves because they all have different number of cm-s at their beginning.

- Substrings that start on positions in $block_2$ are the only substrings that start from symbol that is not equal to cm and have $\leq n-2$ characters cm in them. They will differ among themselves because lengths of their prefixes that do not contain symbol cm are distinct.
- Substrings that start from positions in $block_3$ are the only substrings that start with symbol cm and have a symbol surrounded by cm-s that is not equal to cm (I'm talking about the first character of the string). They will differ among themselves by positions of the character that is not equal to cm.

Problem Editorial: "Life Transfer"

Author: **Eugenie Daniel Posdărascu**
Developer: **Radu Vișan**
Editorialist: **Adrian Budău**

The easy approach to this problem is to fix the number of motorcycles to rent. The number of cars needed will be uniquely determined by this.

Now let's say we want C cars and M motorcycles. We need the oldest C people to drive the cars, and the next oldest M people to drive the motorcycles. For both of these groups we can divide the people into 4 categories:

- People that have at least d years more than the minimum age required to drive the vehicle. From these people, we can afford to take d years from each of them to give to someone else.
- People that have the minimum age required to drive the vehicle, but they are less than d years older than the minimum age required. From these people we can take years equal to their age out of which we subtract l_c or l_m respectively. This can be rewritten as the sum of their ages - l_c or l_m multiplied by their number.
- People that do not have the minimum age required to drive the vehicle, but need less than or equal to d years to reach it. This is symmetrical to the case above: they need years equal to l_p or l_m out of which we subtract the sum of their ages.
- People that are more than d years younger than the minimum required age to drive their vehicle. In this case, it means C and M can not represent the number of cars and motorcycles.

We can also divide the rest of the people in two categories:

- People with age at least $d + 1$ out of which we can take d years to give to someone else (treated as the first category for cars/motorcycles)
- People with age less than or equal to d, out of which we can take years equal to their age minus 1 (treated as the second category for cars/motorcycles).

It can be seen that if we order all people by age all these categories of people become intervals which we can precompute for the whole array and then restrict depending on C and M. And the number of years we can take / we need to give become partial sums on this ordered array of years.

Total complexity: $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

Problem Editorial: "Game on a Tree"

Author: **Anton Trygub**
Developer: **Maksym Zub**
Editorialist: **Stanislav Bezkorovainyi**

It is easy to notice that if a vertex u is an ancestor of vertex v, then vertex v is a descendant of vertex u. And thus if we can move the chip from u to vertex v, then we can also move the chip from vertex v to vertex u.

Now let's build a new undirected graph G, that consists of n vertices. In graph G two vertices v and u are connected with an edge if and only if v is an ancestor of u or u is an ancestor v in the original tree.

Obviously, playing the game on the tree is equivalent to playing the game on this graph, just with a small change. In the tree we could move the chip from vertex u to vertex v if and only if u was either ancestor of v or descendant of v. In graph G we can move the chip from vertex u to vertex v iff u and v are connected with an edge.

Now let's find a *maximum matching* of the graph. More information about matchings of graphs and their properties can be read on page `http://bit.ly/seerc-2019-matching`. But I'll provide a few definitions here:

A *matching* in a graph is a set of edges without common vertices.

A *maximum matching* is a matching that contains the largest possible number of edges.

A *perfect matching* is a matching which matches all vertices of the graph.

Claim. If a maximum matching of graph G is a perfect matching, then the second player wins. Otherwise, the first one wins.

Proof.

- At first, let's proof that if the matching is perfect, then the second player wins. Let's suppose that set of edges E is a perfect matching of graph G. As we all know, every edge in an undirected graph can be defined by a pair a numbers — pair of vertices that the edge connects. So, actually, by building a perfect matching we divide n vertices into $\frac{n}{2}$ pairs of vertices $(v_1, u_1), (v_2, u_2), \ldots, (v_{\frac{n}{2}}, u_{\frac{n}{2}})$, such that every vertex from graph G belongs to exactly one pair.

  Let's see how Bob has to play in order to win this game. Let's say that on the first turn Alice chooses to put the chip on vertex a. Since the matching is perfect, vertex a belongs to some pair. Let it be (a, b). Then Bob chooses to move the chip to vertex b. So now, all the vertices from the pair are used, so the players won't be able to the chip on neither vertex a nor on vertex b. Now Alice has to put the chip in a vertex that belongs to another pair. And so on.

  In general, if on turn k Alice decides to put the chip on vertex a_k, Bob finds the pair (a_k, b_k) to which a_k belongs and moves the chip to vertex b_k. Since we know that wherever Alice decides to put the chip Bob will still have the next vertex to go, it is obvious that Alice will loose, and thus Bob will win.

- Now let's proof that if the maximum matching is not perfect, then Alice can always win.

  Let E be a maximum matching of graph G. E is a set of edges and thus, it can be also represented as a set of pairs of vertices $(v_1, u_1), (v_2, u_2), \ldots, (v_m, u_m)$, where m is the size of E. Since E is not a perfect matching there is at least one vertex t that does not belong to any of the pairs.

  At the first turn, Alice should choose any vertex that does not belong to any of the pairs. At the second turn, Bob will either have no vertices to go (and thus lose the game) or the only vertex he can go will belong to one of the pairs in matching E. Now the game is just the same as in the previous case, but now Bob is the first one to put the chip on a vertex that belongs to one of the pairs of the matching and because of that Bob will definitely lose.

But wait a second... In the previous case every vertex belonged to exactly one pair. Since E in this case is not perfect, there can be lots of vertices that do not belong to any of the pairs. Why can't Bob move to a vertex that does not belong to any of the pairs, just like Alice did in the first turn?

Let's suppose that Bob can do so. There are two possible cases:

- Bob decides to move the chip on a vertex that does not belong to any of the pairs on the second turn. If he can do so, the previous vertex and the new one can be matched, and thus E is not a maximum matching. Contradiction.
- Bob decides to move the chip on a vertex that does not belong to any of the pair on the k-th turn. Let vertices $v_1, v_2, v_3, \ldots, v_{k-1}, v_k$ be vertices on which the chip was placed on turns 1, 2, 3, $\ldots$, $k-1$ and k respectively. Since Alice plays the game just like Bob in the case when E was perfect, $k-1$ unordered pairs $(v_2, v_3), (v_4, v_5), \ldots, (v_{k-2}, v_{k-1})$ belong to E. Whats more, there are edges between k pairs of vertices $(v_1, v_2), (v_3, v_4), \ldots (v_{k-1}, v_k)$. We can delete the $k-1$ pairs from E mentioned earlier and insert k pairs $(v_1, v_2), (v_3, v_4), \ldots, (v_{k-1}, v_k)$. By doing that we create a new matching E_2 that has size greater than that of matching E. That means that E is not a maximum matching. Contradiction.

So now we know how to solve the problem if know the size of maximum matching of the tree. But how to find it? You may try to use standard approaches for finding maximum matching, but these are very likely to get "`Time limit exceeded`" verdict. Here is author's approach that works in $O(n)$ that capitalizes on the way how graph G was built.

We will use recursion + dynamic programming on the initial tree. Let's define dp_v as the minimal number of vertices that are left unmatched if we try to match all the vertices in the subtree of vertex v.

Let's say that we want to calculate dp_v for some vertex v. Now there are two possible senarios:

- Vertex v is a leaf. Obviously, $dp_v = 1$.
- Vertex v is not a leaf, so it has one or more direct descendants. Let's denote the direct descendants of vertex v as $d_1, d_2, \ldots, d_k$. At first, let's calculate values of $dp_{d_1}, dp_{d_2}, \ldots, dp_{d_k}$. Let's define $D = \sum_{i=1}^{k} dp_{d_i}$. If D is greater than 0, then we can match vertex v with one of it's unmatched descendants and get $dp_v = D - 1$. Otherwise, all of the descendants of vertex v were matched, so obviously vertex v has no vertex to match with, so $dp_v = 1$. We get the formula:

$$dp_v = \begin{cases} D - 1, & \text{if } D > 0 \\ 1, & \text{if } D = 0 \end{cases}, \text{ where } D = \sum_{i=1}^{k} dp_{d_i}$$

If $dp_1 = 0$, then we can match all of the vertices and thus the maximum matching is perfect, which means Bob will win. Otherwise, Alice will win.

Problem Editorial: "Projection"

Author: **Virgil Palanciuc**
Developer: **Adrian Budău**
Editorialist: **Adrian Budău**

Let's call the 2 projections the NM and NH projections.

First of all, we can notice that a shadow on row X in one of the projections can only be generated from a cube that is at coordinates $(X, someY, someZ)$. This means that the problem can be solved independently for each X from 1 to N.

First let's discuss the case of no solution: If at some row X one of the projections has at least one shadow (implying there is a cube at this row), but the other one doesn't (implying that there isn't any cube at this row) then obviously we can not construct a solution. If this doesn't happen there is always a solution and it will be obvious from the construction.

For the maximum number of cubes, the solution is unique. If the NM projection has a shadow (X, Y) and the NH solution has a shadow at (X, Z) then we can put a cube at (X, Y, Z). There is no other spot we can place a cube, thus the solution is unique. Since it's unique we just have to print these cubes ordered by X, then by Y, then by Z to get the minimal lexicographically solution.

For the minimum number of cubes let's look at some row X. If the NM projection has P shadows it means there are at least P cubes at this row. Similarly, if the NH projection has Q shadows there are at least Q cubes at this row.

If $P = Q$ and we have shadows in NM at $(x, y_1), (x, y_2), \ldots, (x, y_P)$ and shadows in NH at $(x, z_1), (x, z_2), \ldots, (x, z_Q)$ with $y_1 < y_2 < \cdots < y_P$ and $z_1 < z_2 < \cdots < z_Q$. then we can place the P cubes at $(x, y_1, z_1), (x, y_2, z_2), \ldots, (x, y_P, z_P)$ and there is no solution using less cubes at this row (since we need at least $P = Q$) and no smaller lexicographically one using only P cubes.

If $P < Q$ we obviously need to use the same y multiple times (since we need at least Q cubes at this row). Because we require the minimum lexicographically solution we should use the smallest y multiple times. We can duplicate it until we have as many y as z and then apply the same idea as in the $P = Q$ case.

If $P > Q$ we do as in $P < Q$ except we duplicate the smallest z and then apply the same idea as in the $P = Q$ case.

Total complexity: $(N \cdot M \cdot H)$ time and space.

Problem Editorial: "Tree Permutations"

Author: **Anton Trygub**
Developer: **Stanislav Bezkorovainyi**
Editorialists: **Anton Trygub and Stanislav Bezkorovainyi**

Sort the array first. Say that $a_1 \le a_2 \le \cdots \le a_{2n-2}$.

- **Claim 1:** If $a_i > i$ for some i, there is no valid tree at all.

  Proof: Suppose that $a_i > i$. Obviously, this means that at most $i-1$ numbers don't exceed i, but, from the other side, parents of nodes $2, 3, 4, 5, \ldots, i+1$ all have to be at most i. Contradiction.

- **Claim 2:** If $a_i = i$ for some i, the path from n to 1 has to include i.

  Proof: Suppose that $a_i = i$. Obviously, this means that at most $i-1$ numbers don't exceed $i-1$. From the other side, parents of nodes $2, 3, 4, 5, \ldots, i$ all have to be at most $i-1$. Therefore, all these $i-1$ numbers will be used as parents for $2, 3, 4, 5, \ldots, i$, and therefore parents of all numbers from $i+1$ to n will be at least i. This means, that we can't get from n to 1 without coming through i (suppose that j is the last vertex on the path from n to 1 which is larger than i, then it's parent has to be i).

- Let $t_1, t_2, \ldots, t_m$ be the set of indexes for which $a_x = x$, with $t_1 = 1$. Let S be the set of all numbers which appear among a_i, excluding t_j for $1 \le j \le m$ — those are potential other vertices on the path. Obviously, there is no k-path for $k < m$ (as every path from n to 1 has to include at least $m+1$ nodes $t_1, \ldots, t_m, n$). Also, there is no k-path for $k > m + |S|$, as the total possible number of different vertices in the path doesn't exceed $m + |S| + 1$ (corresponding to n).

- **Claim 3:** For $m \le k \le m + |S|$, the k-long trees exist. Sum of weights for k-perfect tree can then be computed as following: mark $t_1, t_2, \ldots, t_m$ as used, and $k - m$ smallest elements from S as used. Answer is the sum of k largest of $2n - 2 - k$ not marked numbers.

  Proof: Obviously, this is the upper bound, it's left to prove that it's achievable. Denote $k - m$ smallest elements of S as $s_1 < s_2 < \cdots < s_{k-m}$. So our path will contain vertices $path_1 < path_2 < \cdots < path_k < path_{k+1}$, where $path_1 = 1$, $path_{k+1} = n$, and $path_1, path_2, \ldots, path_k$ is a sorted union of $t_1, t_2, \ldots, t_m, s_1, \ldots, s_{k-m}$. We now set parent of $path_{i+1}$ to be $path_i$ for each i.

  Mark first occurrences of $t_1, t_2, \ldots, t_m, s_1, \ldots, s_{k-m}$ in a as used. Also mark all the vertices that we've given a parent this way. There are $2n - 2 - k$ indexes which are yet not used, let them be $x_1 \le x_2 \le \cdots \le x_{2n-2-k}$. Now look at all the $n - k - 1$ vertices (except for 1) which aren't assigned a parent yet, assign a_{x_i} to the i-th of them (in increasing order). After that, distribute all $n - 1$ numbers which are left can be used as weights, use k largest of them on the $path$, and others arbitrarily. This gives the answer from the claim, we just have to prove that we have actually produced a tree.

  Let $v_1, v_2, \ldots, v_{n-k-1}$ be a sorted array of vertices which were not assigned a parent yet. Note, that vertex 1 can not have a parent, so $v_1 \ne 1$. Let's iterate over array v. Let's say that we want to assign a parent for vertex v_i . As said before, we want to assign parent of vertex v_i as a_{x_i}. This is valid if and only if $a_{x_i} < v_i$. Now we have to prove that $a_{x_i} < v_i$ for any i from 1 to $n - k - 1$.

  Let's suppose that it is not true and for some i, $a_{x_i} \ge v_i$. Without loss of generality, $path_j \le a_{x_i} < path_{j+1}$ for some j. It means, that a_{x_i} is greater than or equal to the numbers of first j vertices of the $path$ (i.e marked vertices) and $a_{x_i} \ge v_i > v_{i-1} > \ldots > v_1$ (i.e i unmarked vertices). Since a_{x_i} is greater than or equal to the numbers of the first $i + j$ vertices it means, that $a_{x_i} \ge i + j$.

  Now let's look at value x_i. What does it mean that an element has position x_i in array a? It means that there are exactly $x_i - 1$ elements before it in the array.

How many unmarked positions are there before x_i? These are $x_1, x_2, \ldots, x_{i-1}$. And thus, there are exactly $i-1$ unmarked positions before x_i.

How many marked positions are there before x_i? As we already said, $path_j \leq a_{x_i} < path_{j+1}$. Since we marked the first occurence of $path_j$ in the array, so $path_j > path_{j-1} > \ldots > path_1$ these are all before of the position x_i. $x_i < path_{j+1} < \ldots < path_k$ these all, obviously go after position x_i since array a is sorted. So there are exactly j marked positions before x_i.

Since every position is either marked or not marked, there are exacly $i-1+j$ positions before x_i. So, $x_i = i-1+j+1 = i+j$.

Quick observation. From first two claims we know that if for some i $a_i > i$, then there is no answer and if for some i $a_i = i$, then we have already added i to out path. So, now we know that on every position i that has not been marked yet $a_i < i$. Thus, $a_{x_i} < x_i$. But we have already discovered, that $a_{x_i} \geq i+j$ and that $x_i = i+j$, so we get $a_{x_i} \geq x_i$. Contradiction.

- Now let's move to the algorithmic side. We have to be able to calculate the sum of k largest numbers among $x_1, x_2, \ldots, x_{2n-k}$ where $k-m$ smallest elements of S are discarded. It's the easy part: for $k = m$, the set of m largest numbers will be just m largest numbers in x. Now, maintain the set of values of S that are prohibited, and we haven't discarded yet. At every step, add next element of S to this set, and the next (in decreasing order) x_i in the set of the largest numbers. Now, while set of largest numbers contains some c that is in set **to be discarded**, delete one instance of c from both **to be discarded** and largest numbers, and add a next number into the largest instead. Sum recalculation is obvious.

Problem Editorial: "Absolute Game"

Author: **Roman Bilyi**
Developer: **Roman Bilyi**
Editorialist: **Stanislav Bezkorovainyi**

Claim. This game is equivalent to the game where Alice has to choose x from her array and then Bob has to choose y from his array.

Proof. Since Alice can finish the game with whatever x she wants, the result of the game for Bob can't be better than in the game mentioned above. Now let's prove that it is always possible for Bob to achieve this.

At first, let's define an array c of n elements. Each c_i has such value, that the value of $|a_i - b_{c_i}|$ is minimal possible. If for some position i there are more than one possible c_i, then choose the smallest one. In other words, we build an array such that if Alice finishes the game with $x = a_i$, then the best outcome for Bob is if $y = b_{c_i}$.

Let's say that in the first turn, Alice removes some element a_i. Then we should also remove the element c_i from array c. Now, since there are $n - 1$ elements of the array c, there is at least one such position $1 \le p \le n$, that $c_i \neq p$ for every i. Bob should remove the element b_p. It is optimal because whatever x we end up with, he will still be able to choose the best possible y from the remaining elements of array b.

In general, if after some turns there are k elements in Alice's array and Bob's array, now it's Alice's turn and if she removes an element a_i from her array, then Bob should remove an element b_p on such position p that there are no $c_i = p$ among all possible i (i.e. among all such i that a_i have not been removed yet).

If Bob plays the way mentioned above, then after each his turn, for every x from the remaining array a, he will be able to choose the best possible y from the remaining array b.

Ok, I got it. But how to implement it? Implementation of the "light version" of the game is rather straightforward. Let's define an array ans such that ans_i is the answer if Alice decides to choose x to be equal to a_i. If x is equal to a_i, then Bob will obviously choose such y that $|x - y|$ is minimal possible. Thus, we get the formula:

$$ans_i = \min_{j=1}^{n} |a_i - b_j|$$

Since Alice wants to maximize $|x - y|$, the final answer is equal to $\max_{i=1}^{n} ans_i$.

Problem Editorial: "Graph and Cycles"

Author: **Anton Trygub**
Developer: **Stanislav Bezkorovainyi**
Editorialist: **Stanislav Bezkorovainyi**

Since n is always odd, then the degree of each vertex (i.e number of vertices adjacent to it) is always even, so there are even number of edges adjacent to each vertex.

Let's look at an edge e that connects two vertices (v_1, v_2). In a cycle-array to which e will belong, this edge will be adjacent (i.e compared when calculating the price of the cycle-array) with exactly one edge, that is connected to v_1 and exactly one edge that is connected to v_2.

So, if we look at a vertex v, then all the edges that are connected to it can be divided into $\frac{n-1}{2}$ pairs of edges (e_1, e_2), where e_1 and e_2 belong to same cycle-array and e_1 is adjacent with e_2. So now, we can define the *price of a vertex* as the sum of maximums of weights in such pairs. In other words, if all the edges that are adjacent to a vertex v are arranged in pairs like that $(e_1, e_2), (e_3, e_4), \ldots, (e_{n-2}, e_{n-1})$, then:

$$\text{Price of } v = \sum_{i=1}^{\frac{n-1}{2}} \max(\text{weight of } e_{2 \cdot i-1}, \text{weight of } e_{2 \cdot i})$$

It is easy to see that sum of prices for all of the vertices is equal to price of the cycle-split.

Let's say that edges $e_1, e_2, \ldots, e_{n-1}$ with weights $w_1, w_2, \ldots, w_{n-1}$ are connected to v. Let's say, that w' is a sorted array w. It is easy to prove that the smallest possible price of a vertex can be obtained if we split the wieghts into pairs in the following way: $(w'_1, w'_2), (w'_3, w'_4), \ldots, (w'_{n-2}, w'_{n-1})$.

What is left to prove is that it is achievable. Let's say that for each vertex v we arranged edges adjacent to it into pairs, such that the price of the vertex is smallest possible. Let's denote this arrangement as $(e'_1, e'_2), (e'_3, e'_4), \ldots, (e'_{n-2}, e'_{n-1})$. Let's call a pair (e'_i, e'_{i+1}) `unfulfilled` if e'_i was used and e'_{i+1} weren't or vice versa.

Now, we can build the necessary cycle-split the following way:

- If all of the edges were used, then stop the algorithm.
- Otherwise, we should start forming a new cycle-array that will be added to the split. Find any vertex v that has an unused edge adjacent to it. We add this edge to the current cycle-array and mark it as used. Let's say that it is edge that connects vertex v with vertex u. Let's set our current vertex as u and repeat the following algorithm:
  - Let's suppose that our current vertex is some vertex u. If there are no unfulfilled pairs of it, then the cycle-array has been finally formed, thus we stop the algorithm.
  - Otherwise, there is exactly one unfulfilled pair. Wlog that it is pair (e'_i, e'_{i+1}).
    If e'_i has not been used yet, then add it to our cycle-array and set the current vertex as a vertex that the edge e'_i connects u with. Now there are no unfulfilled pairs of the vertex u. It can be seen that e'_{i+1} was added just before we set u as our current vertex. Thus, e'_i and e'_{i+1} will be compared, when calculating price of the array.
    The similar works for the case when e'_{i+1} has not been fulfilled.

  The cycle array that is obtained this way is always valid, because it is an Euler's tour of the edges that belong to it.

So now, we have proven the solution. How to find the answer?

For any vertex v create an array w' that has all the weights of the edges that are connected to it in an ascending order. The price of this vertex is the sum of the values on even positions in the array w'. The price of the split is the sum of prices of all the vertices.

Total time complexity $O(n^2 \log n)$.

Problem Editorial: "Stranded Robot"

Author: **Vladimir Olteanu**
Developer: **Vladimir Olteanu**
Editorialists: **Vladimir Olteanu and Roman Bilyi**

For the sake of simplicity, we will be assuming that $M = N = P$.

The key insight is that the robot can only attach itself to blocks that can be lit. For each of the 6 directions from which the sun can shine, there are at most N^2 blocks that can be lit. The rest of the blocks make no difference whatsoever, and can be ignored.

Let us consider what happens when the sun is shining along the positive direction of the z axis. In order to determine which blocks are lit, we will need to compute the depth buffer $zMin$, where $zMin[x, y]$ is the minimal z coordinate of all the blocks that share the same x and y, or $-\infty$ if no such block exists.

From this perspective, the moves available to a robot located at (x, y, z) are:

- If $z + 1 = zMin[x, y]$, the robot is resting on top of a block, and can move to $(x \pm 1, y, zMin[x \pm 1, y] - 1)$ or $(x, y \pm 1, zMin[x, y \pm 1] - 1)$ if the $zMin$ at the the destination is greater or equal to $z + 1$.
- If $z + 1 < zMin[x, y]$, the robot can fall to $(x, y, zMin[x, y] - 1)$.

When the sun is shining along the negative direction of the z axis, we compute the depth buffer $zMax$ instead, where $zMax[x, y]$ is the maximal z coordinate of all the blocks that share the same x and y, or ∞ if no such block exists. The moves available are:

- If $z-1 = zMax[x, y]$, the robot can move to $(x\pm1, y, zMax[x\pm1, y]+1)$ or $(x, y\pm1, zMax[x, y\pm1]+1)$ if the $zMax$ at the the destination is less or equal to $z - 1$.
- If $z - 1 > zMax[x, y]$, the robot can fall to $(x, y, zMax[x, y] + 1)$.

These statements can be generalized for the other two axes.

There are at most $6N^2$ cells that the robot can visit, and the shortest path to the teleporter can be found using BFS. This solution works in $O(N^2)$ after reading the input. The total complexity is $O(N^3)$ and the memory usage is $O(N^2)$.