AGC 10 解説

yutaka1999

2017 年 2 月 4 日

For International Readers: English editorial starts on page 4.

A : Addition

偶奇が等しい 2 つの数を足すと、その和は偶数になる。よって、$N \geq 2$ より、最後に残る数は偶数でなくてはならない。

逆に、和が偶数ならば、もとの数の中に奇数は偶数個あることになるので、奇数を 2 つずつのペアにしてその和に置き換えることで、全ての数を偶数にできる。このとき、どのような順番でも数は 1 つにできるので十分である。

よって、合計が偶数かどうかの判定ができればいいので、$O(N)$ で解くことができる。

B : Boxes

一回の操作で取り除かれる石の個数はちょうど $\frac{n(n+1)}{2}$ 個であるので、はじめにある石の個数から、操作をする回数が分かる。これを k とすると、ちょうど k 回操作を行って、全ての数が等しくなるようにできるかを判定すればよい。

ここで、$d_i = A_{i+1} - A_i$ の変化を考えると、一回の操作で $\{d_i\}$ の内、ちょうど一つが $+(n-1)$ され、その他が -1 される。よって、$\{d_i - k\}$ のうち毎回一つを $+n$ するとして、全て 0 にできるか判定すればよい。

これは、各 d_i が負であり、かつ n の倍数であることが必要十分であるから、$O(N)$ で判定可能である。($\sum d_i = 0$ であることを用いれば、必要な $+n$ の回数が必ず k 回であることが分かる。)

C : Cleaning

適当な頂点 r を根として、根付き木として考える。さらに、根以外の各頂点に対して、その頂点とその親を結ぶ辺を通る操作の回数を通る回数と呼ぶことにする。このとき、明らかに各葉に対しては通る回数が分かる。ここで、葉でない頂点 v について、v の子での通る回数が分かっているときに、v の石を全て取り除けるかどうかと、その頂点の通る回数を調べる。

各子の通る回数を $c_1, c_2, ..., c_k$ と置くと、子を通る操作は v と v の親と通るパスを通るか、v で他の子を通る操作とつながるかの 2 通りで、どちらも v から石を 1 つ取り除くから、前者のパスの個数、つまり、v の通る回数を T と置くと、$T + \frac{\sum c_i - T}{2} = A_v$ となる。よって、この情報から、T の値が分かるので、後は異なる子を通るパス 2 つをペアにしたものを $\frac{\sum c_i - T}{2}$ 個作れるかどうかを判定すればよい。

つまり、$L = \frac{\sum c_i - T}{2}$ として、1 から k までの各数 i が c_i 回まで使えるときに、ペアを L 個作れるかとい

う判定ができればよく、そのようなペアは最大で $\min\left(\max c_i, \frac{\sum c_i}{2}\right)$ 個であることが示せるので、この判定は容易にできる。

以上より、各頂点に対して、その子の情報から今の頂点の情報を処理することができたので、根付き木の下の頂点から順に処理していくことで $O(N)$ で解くことができる。

D : Decrementing

毎回の操作で後者の操作を行わない、つまり、GCD で割らないときのことを考える。このとき、$\sum(A_i - 1)$ の偶奇で勝敗が決まるので、同様にして、偶奇に注目して勝敗を調べる。

まず、A_1 から A_n の中に奇数があれば、その GCD も奇数なので、GCD で割った後も各数の偶奇は入れ替わらない。さらに、A_1 から A_n の GCD が 1 のとき、少なくとも一つ奇数がある。この性質を用いると、以下のように場合分けできる。

i)　A_1 から A_n の中に偶数が奇数個あるとき
偶数が一つ以上あるため、偶数を一つ選んで -1 し、相手に渡す。このとき、操作前に奇数が 1 つ以上あり、操作後には奇数が 2 つ以上あるため、相手の操作時の GCD も奇数となる。よって、相手から戻ってくるならば、その時も偶数が奇数個あるため、先手はこれを繰り返すことで必勝である。

ii)　A_1 から A_n の中に偶数が偶数個あるとき
奇数が 2 つ以上あるならば、どうやっても相手に渡したときに偶数が奇数個あるため、i) より、後手必勝になる。ただし、奇数がちょうど 1 つあるときは、その奇数を消すことで、GCD が偶数になるため、後手必勝とは限らない。よって、この場合は、後手必勝になるか操作が一意に定まるかのいずれかなので、これを繰り返すことでどちらが必勝かわかり、GCD が偶数になる回数は高々 $O(\log(\max A_i))$ 回なので、十分高速に動く。

以上より、$O(N \log(\max A_i))$ でこの問題を解くことができる。

E : Rearranging

まず、後手の最適手を考える。$A_1, A_2, ..., A_N$ の順に並んでいるとする。互いに素な数のみが順序を入れ替えられるため、$\gcd(A_i, A_j) \neq 1$ であれば、これらの相対的な順序は変化しない。

よって、$i < j, \gcd(A_i, A_j) \neq 1$ となる (i, j) に対して、$i \to j$ と有向辺を張ることで DAG ができ、このトポロジカル順序で辞書順最大のものが求めるものとなる。これは簡単に求められるので、後手の最適手は簡単に求められる。

よって、先手の最適な順序を求めればよい。自然数からなる集合 S に対して、各要素間に GCD が 1 でないなら辺を張る、というようにして作ったグラフを $g(S)$ とおき、$g(S)$ が連結グラフであるような集合を連結集合と呼ぶことにする。このとき、先の後手の動きからもわかるように、各連結集合ごとの順序が定まれば後手の差す手は決まるので、それぞれ独立に考えてよい。

ここで一般化して、「連結集合 S に全順序を定めて後手の操作の後に辞書順最小になるようにする。ただし、後手がどのように操作しても先頭が X と互いに素となってはならない。」という問題を考える。この答えとなる順序の一つを $f(S, X)$ と置く。

まず、X との GCD が 1 でない元の中で最小の元を x とする。このとき、明らかに先頭は x 以上でなくてはならないが、$g(S)$ が連結なため、x を始点とする DFS 順序にそってほかの要素を並べることで、後手が最適に行っても、先頭を x のままにすることができる。よって、$f(S, X)$ の先頭は x としてよい。このとき、$S \backslash \{x\}$ の各連結集合 T での順序が求まれば、それを適当につなぎ合わせることで $f(S, X)$ が得られるため、その順序を考えることにする。

まず、T を並べたとき、その先頭が y であるとする。$(x, y) = 1$ ならば、x と y を入れ替えることができるため、明らかに不適である。逆に、$(x, y) \neq 1$ ならば、どのようにしても x と y の順序は入れ替えらないため、T の並びとしては、先頭が x と互いに素ではない並びで最適な並び、つまり、$f(T, x)$ が最適であることが分かる。

以上をまとめると、$f(S, X)$ は X との GCD が 1 でない元で最小の元 x を先頭にし、S から x を取り除いたときの各連結成分 T に対する $f(T, x)$ を適当に連結したものとすればよい。

求める先手の順序は $S = \{A_1, A_2, ..., A_n\}$ （ただし、S は multiset として考える）として、$f(S, 0)$ であるから、先頭の元を取り除く操作が N 回、各連結成分を求める操作が $O(N \log N)$ 程度で可能なので、全体として $O(N^2 \log N)$ で解くことができる。

ただし、連結成分を求める際は、あらかじめすべての数の素因数を計算し、数と素因数の間に辺を張ってグラフを構築するものであるから、その各数の素因数の計算に $O(\sqrt{\max A_i})$ かかる。よって、全体としては、$O(N(N \log N + \sqrt{\max A_i}))$ となる。

F : Tree Game

最初に高橋君が駒を置く頂点を一つ固定する。このとき、各頂点について $O(N)$ で解ければよい。ここで、木は二部グラフであるから、どの頂点に対してもそこから石をとるプレイヤーは一意に定まることに注意する。

その選んだ頂点を根として、木を根付き木として考える。このとき、以下のように再帰的に各頂点 v に対して $f(v)$ を定める。

- v が葉であるとき、$f(v) = A_v$
- v が葉でないとき、v の直接の子での $f(v)$ が全て A_v 未満なら、$f(v) = \inf$
  そうでないなら、$f(v) = A_v$

このとき、葉でない頂点 v に対して、v を根とする部分木で v に駒を置いて始めたゲームの勝敗と $f(v) = \inf$ かどうかが一致することを帰納的に示す。

まず、v の直接の子がすべて葉の時、$f(v) = \inf$ ならば、ある子 c で $A_c < A_v$ となる。このとき、先手は v から常に c に動かすことで勝つことができる。逆に、$f(v) = A_v$ の時は、どの頂点 c に対しても $A_c \geq A_v$ となるため、先手はどのようにしても勝つことができない。よって、この場合は示された。

次に、v のすべての子で f の値が定まっているときを考える。このとき、$f(u) = \inf$ である頂点 u に動かすと後手が勝ちになるため、このような頂点 u に先手は動かすことができない。さらに、そうでない頂点 u に対しては、先手が u に動かしたときに後手は v に動かすのが最適である。なぜならば、後手が u の子に動かすとき、u を根とする部分木だけ動かす場合は先手勝ちであるから、後手が勝つには途中で u から v へ戻る必要がある。しかし、結局 u を根とする部分木において、後手陣地である u の石が減るだけであるから、また v から u へ戻ったとしても、その部分木内だけ動かした場合は先手勝ちであることに変わりない。よって、u からその子へ行ったとしても、u の石の個数を減らすだけで、結局 v に戻る必要があるため、意味がない。よって、後手は u から v に動かすのが最適である。

以上より、v のすべての子で f の値が定まっている場合も、v の子がすべて葉である場合と同じように処理することができるため、正当性が示された。

さて、先のような f の値は根付き木を根から再帰的に調べることで $O(N)$ で求めることができるため、根をすべて試すことで、$O(N^2)$ で解くことができる。

AGC 10 Editorial

yutaka1999

February 4th, 2017

A : Addition

Suppose that after the operations you have only one integer. This integer is obtained as a sum of two integers of the same parity, so it must be even. This means that the sum of A_i is even.

On the other hand, if the sum of A_i is even, we can prove that the answer is always "YES". In this case initially there are even number of odd integers. Divide them into pairs and perform the operation for each pair. This way all integers will be even and after that we can add any pair of integers.

Thus, we just need to check whether the sum of A_i is even, and it can be done in $O(N)$.

B : Boxes

In each operation we remove $\frac{n(n+1)}{2}$ stones. Let s be the total number of stones. Obviously, s must be a multiple of $\frac{n(n+1)}{2}$. Let $k = s/\frac{n(n+1)}{2}$. We want to check whether after exactly k operations, the number of stones in each box can be the same.

Let $d_i = A_{i+1} - A_i$, the difference of number of stones in adjacent boxes. Usually an operations reduces this number by 1, but one of the values of $\{d_i\}$ is incremented by $n-1$. It means that for some non-negative integer x, $d_i - (k-x) + (n-1)x = 0$ or $k - d_i = nx$ (here x is the number of "unusual" operations happened for d_i). Thus, for each i, $k - d_i$ must be non-negative and it must be a multiple of n. We can also prove that this condition is sufficient: the sum of $(k-d_i)/n$ is always k because the sum of d_i is zero. We can solve this problem in $O(N)$ time.

C : Cleaning

Let's assign labels to the edges. Initially all the edges are labeled with zero. When an operation is performed on a path connecting two different leaves, we increment the labels of all edges on the path by one.

When can we remove all stones? Each leaf vertex on a path is incident to an edge on the path, and each non-leaf vertex on a path is incident to two edges on the path. Thus, we can see that this is equivalent to the following:

- For each leaf v, the label assigned to the only edge incident to v is A_v.
- For each non-leaf vertex v, the sum of labels assigned to the edges incident to v is $2A_v$.

Now, consider the tree as a rooted tree. From the conditions above, we can uniquely determine the labels of all edges in the order from leaves to the root. Note that this is not always possible - make sure to check that all labels are non-negative and the condition is satisfied at the root.

In case we can successfuly assign labels, how can we check whether we can decompose it into a set of paths connecting leaves? Consider a certain non-leaf vertex, and let $c_1, \ldots, c_k$ be the labels assigned to these edges. These $\sum c_i$ things must be separated into pairs, and each pair must consist of two things from different edges. Thus, the condition is that, $\sum c_i$ must be even and for each i, $c_i \leq \sum c_i/2$ must be satisfied.

The solution above works in $O(N)$ time.

D : Decrementing

First consider an easier version of problem: we don't divide the integers by their GCD. In this case, obviously the answer only depends on the parity of the total number of stones. The original problem also has something to do with parities.

Suppose that in your turn all integers are odd. If all integers are one, you can't perform any operations and you lose. If not, you can perform an operation, and after the operation exactly one integer is even and the others are odd (here the GCD doesn't matter - since GCD is odd in this case, it doesn't change the parities). However, if your opponent perform an operation on the only even number, you will again face the situation where all numbers are odd, and you will lose eventually.

If exactly one integer in your turn is even, you can always win. By performing an operation in the only even number, you can force the opponent to lose (as described above).

By similar observations, we get the following:

1. In case there are odd number of even integers, you win. Your strategy is as follows. In your turn you choose one of even numbers and perform an operation on it. After the operation there will be two or more odd integers, so the GCD operation doesn't change the parities. Thus, in the opponent's turn there will be even number of even integers, and in your next turn you will again have odd number of even integers. By repeating this strategy, you always win.
2. In case there are even number of even integers and two or more odd integers, you lose. No matter what you do, after your operation the opponent will be given a winning state, as described above.
3. In case there are even number of even integers and exactly one odd integer. In this case, it's clear that if you perform an operation on one of even integers, you lose (your opponent will face a winning state). Thus, you must perform an operation on the odd integer, and after that divide all integers by their GCD. It's not clear what happens in this case - just simulate the game and solve the problem recursively.

When the simulation happens, all the integers are divided by at least two. Thus, the number of simulations is $O(\log(\max A_i))$. The time complexity of the algorithm above is $O(N \log(\max A_i))$.

E : Rearranging

Let's think about the second player's optimal strategy first. He wants to maximize the permutation by swapping two adjacent coprime elements. If two numbers x and y are in the permutation in this order, and if they are not coprime, he can never change the relative order of these elements. We can also prove that this condition is sufficient: he can get any permutation that satisfies the condition above by repeating operations.

For convenience, assume that the input is sorted ($A_1 \leq \cdots \leq A_N$). Construct a graph with N vertices. There is an edge between vertices i and j if A_i and A_j are not coprime. After the first player decides his arrangement, each edge in this graph is directed. The second player wants to find the greatest topological sort of this directed graph.

Next, let's think about the first player's optimal strategy. From the observations above, we can handle each connected component independently. When we merge two connected components, we first get two sequences from each connected component, and find the greatest sequence we can get by merging the two sequences. From now, we assume that the graph is connected.

Let x be the smallest element in the permutation. The first player can force that, even after the second player's operations, the first element of the permutation becomes x. For example, take an arbitrary spanning tree of the graph. Arrange the numbers according to the DFS-order of this spanning tree. Then, each element is topologically greater than x in the directed graph, so the second player can't change the first element. Threrefore the first player should put x first.

Then, remove x from the graph and again compute connected components. For the same reason as above, we can consider each connected component independently. The only difference is that, if y comes first in a connected component, y must be non-coprime to x (i.e., x and y must be adjacent in the graph). Otherwise the second player can swap x and y and make the permutation greater. We do this recursively in the same manner.

How can we implement it efficiently? Run dfs in the graph and construct DFS-trees of the graph. When there are multiple adjacent vertices, visit the smallest vertex first. This way, we get a DFS-tree with the following properties:

- Each edge connects an ancestor and a descendant (like all DFS-trees).
- x is the smallest element in the subtree rooted at x.
- After we remove x, we can get new connected components as the subtrees rooted at x's children.

This is exactly what we need to compute. After computing this DFS tree, we assign a

sequence to each vertex. In order to compute the sequence assigned to vertex x, first we find the lexicographically greatest sequence that can be obtained by merging sequences assigned to x's childrem. Then, append x to the front of this sequence, and this is the sequence we want to assign for x. We compute this from leaves to the root, and the sequence assigned to the root is the answer of the problem.

The complexity of this algorithm is $O(N^2 \log \max A_i)$.

F : Tree Game

Suppose that Takahashi initially puts the piece on vertex r. Let r be the root of the tree. For each vertex v, we define $state(v)$ as follows:

Consider a subtree rooted at v. The two players play the game using this subtree (they are not allowed to move the piece out of the subtree), and initially the piece is at vertex v. If the first player can win in this game, define $state(v) = W$. Otherwise $state(v) = L$.

Note that $state(r)$ gives the result of the entire game when the piece is initially at r.

We claim the following:

1. If there exists a child c of v such that $A_c < A_v$ and $state(c) = L$, then $state(v) = W$.
2. Otherwise, $state(v) = L$. (In particular, if v is a leaf, $state(v) = L$.)

The proof of 1. Suppose that this is your turn and the piece is currently at v. First you take a stone from v (this is possible because A_v is not zero) and move the piece to c. Whenever the opponent tries to move the piece from c to v, you can refuse to do that by moveing it back to c (this is possible because $A_v > A_c$). This way, your opponent is forced to play the game within the subtree rooted at c. Since $state(c) = L$, your opponent loses and you win. (Strictly speaking, your opponent can also reduce the value of A_c by moving the piece between c and v back and forth, but this doesn't change the state of vertex c.)

The proof of 2. If v is a leaf, you can't move the piece and you lose. Otherwise you can move the token from v to one of v's children, w. There are two cases: $state(w) = W$ or $A_w \geq A_v$. If $state(w) = W$, your opponent never moves the piece back to v and play the rest of the game completely within the subtree rooted at w. Since $state(w) = W$, this way your opponent wins and you lose. If $A_w \geq A_v$, your opponent refuses to move the piece from v to c by moving it back to v (this is possible because $A_w \geq A_v$). Thus, you lose in this case.

This way, for a fixed position of the initial piece, we can solve the problem in $O(N)$. In total the complexity of this solution is $O(N^2)$. (Exercise: can you improve it to $O(N)$?)