Atcoder Grand Contest 013 解説

writer : maroonrk

2017 年 4 月 17 日

For International Readers: English editorial starts from page 5.

A : Sorted Arrays

A_1 と A_2 の間で数列が切られているような最適な分割があったとします。この時、A_2 と A_3 の間を切って、A_1 と A_2 をくっつければ、これもまた最適な分割となります。よって、A_1 と A_2 の間で数列が切られていないと考えても最適解は変わりません。次に、A_2 と A_3 の間について考えます。A_1,A_2,A_3 という数列が単調非減少でも単調非増加でもない場合、A_2 と A_3 の間は必ず切られています。A_1,A_2,A_3 という数列が単調非減少または単調非増加である場合は、A_1 と A_2 の時と同じ議論を用いると、A_2 と A_3 の間を切らない最適な分割があると分かります。続く要素についてもこのように考えていくと、結局、数列の先頭から条件を満たすできるだけ長い列を切り出す、という操作を続けることが最適であると分かります。この操作は $O(N)$ で行えるので、この問題を解くことが出来ました。

B : Hamiltonish Path

まず最初に、どんなに短くても良いので適当なパスを作ります。次に、そのパスが条件を満たすかどうか調べて、もし条件を満たすならば、そのまま出力します。条件を満たさない場合、パスの端点と直接結ばれているのに、パスに含まれていないような頂点があることになります。そこで、その頂点を新しいパスの端点として追加し、パスを伸ばします。パスを伸ばしたら、再び条件を満たすか確認し、条件を満たさないならば、更にパスを伸ばして・・・という風に繰り返して行きます。パスを伸ばせる回数は高々 $N-1$ 回しかないので、このアルゴリズムは条件を満たすパスをいつかは出力します。計算量は、各頂点について、パスの端点として追加される回数が高々 1 回であることと、各辺について、パスの端点とパス内を結んでいると分かること、パスの端点とパス外を結んでいると分かること、がそれぞれ 1 回ずつであることから、このアルゴリズムは $O(N+M)$ で動き、この問題を解くことが出来ました。

C : Ants on a Circle

問題の設定を次のように言い換えます。

N 匹の蟻がいて、それぞれの蟻は 1 から N まで番号の書かれたゼッケンをつけている。二つの蟻が同じ場所についた瞬間、その二つの蟻はゼッケンを交換し、そのまますれ違って同じ方向に歩き続ける。T 秒後のゼッケンの番号と座標の対応を答えよ。

どの瞬間においても、1 のゼッケン、2 のゼッケン、…N のゼッケンは、この順に時計回りに分布することになります。すると、二つの蟻がゼッケンを交換する時、時計回りに歩いてる蟻のゼッケンの番号は 1 増え、反時計回りに歩いている蟻のゼッケンの番号は 1 減ります。ある二つの蟻が T 秒間にすれ違う回数は、$O(1)$ で求めることができます。よって、最初に 1 のゼッケンをつけていた蟻が、T 秒間に他の蟻とすれ違う回数の合計は、$O(N)$ で分かります。他の蟻とすれ違う回数が分かるので、最初に 1 のゼッケンをつけていた蟻が T 秒後に付けているゼッケンの番号も分かります。これで、T 秒後にどの番号のゼッケンがどの座標にあるかが一つ分かります。ところで、T 秒後にゼッケンのある座標の集合は、当然、T 秒後に蟻のいる座標の集合と同じなので、T 秒後にそれぞれの蟻がどこにいるかを求めると、番号は分からないが、どの座標にゼッケンがあるのかが分かります。ゼッケンの番号が時計回りに $1, 2, ...N$ となっているので、一つでもゼッケンの番号がわかれば他の番号も分かりますが、先ほど一つゼッケンの番号と位置の情報を得たので、これで求められます。ゼッケンのある座標の集合を昇順にソートする部分がボトルネックで、$O(NlogN)$ でこの問題は解けました。

D : Piling Up

ある実行可能な積み方 X があったとします。そして、最初に赤い積み木を A 個、青い積み木を $N - A$ 個箱に入れた時に、X が実行可能であるような最小の A がとれます。この時、最初に赤い積み木を A 個、青い積み木を $N - A$ 個箱に入れた状態から X を実行すると、箱の中の赤い積み木が 0 個である瞬間が存在します。もし存在しないとすれば、$A = A - 1$ としても X が実行可能であることから、これは分かります。よって、初期状態と積み方の組み合わせとして、箱の中の赤い積み木が 0 個である瞬間が存在するようなものだけを考えるようにします。すると、ある積み方は一度しか数えられない事になり、また全ての積み方は数えられる事になります。そしてこれは、次のような DP で数えることができます。

$DP[i][j][k] = i$ 回目の操作の直後において、箱に j 個の赤い積み木があり、箱の中の赤い積み木が 0 個になったことが $(k = 0 \rightarrow$ない$)(k = 1 \rightarrow$ある$)$ 時の場合の数

この DP の初期値は、$DP[0][0][1] = 1, DP[0][1][0] = 1, DP[0][2][0] = 1, ...DP[0][N][0] = 1$ となります。そして、最終的に求める答えは $DP[M][0][1] + DP[M][1][1] + ...DP[M][N][1]$ となります。この DP は遷移が $O(1)$ で求まるため、全体で $O(NM)$ で計算でき、この問題は解けました。

E : Placing Squares

たくさんの長方形を置く代わりに、問題を次のように言い換えます。

長さ N の棒の上に、いくつかの仕切りを立てる。仕切りは棒の左端からの距離が整数の場所にのみ立てる。棒の両端には必ず仕切りを立て、逆に印のついている部分の真上には必ず仕切りを立てない。二つの仕切りの間には、赤いボールと青いボールを置く。ボールが置かれる座標は、棒の左端から距離が $i.5(i$ は整数$)$ という形で表されるものだけを考える。二つのボールが同じ座標にあっても構わない。この時、仕切りとボールの置き方は全部で何通りあるか。

この問題が元の問題と同値であることは、仕切りを置く位置を正方形の置き方に、ボールの置く位置の自由度を正方形の面積に対応させればすぐに分かります。この問題は、次のような DP で解けます。

$DP[i][j] =$ 棒の左端から距離 i の地点までの仕切りとボールの設置を終えており、最も右にある仕切りの右側に既に置いてあるボールが j 個であるような場合の数

この DP の遷移は、印のある位置でも無い位置でも $O(1)$ で行えるので、$O(N)$ で解くことができます。し

かし、DP の遷移の形に注目すると、これは 3×3 行列で表せるので、行列累乗が行えます。これで、印の無い連続する区間での DP の遷移が $O(logN)$ で行えるようになるので、全体で $O(MlogN)$ でこの問題は解けました。

F : Two Faced Cards

まず C_i の値を昇順にソートしておきます。また、$B_i = min(A_i, B_i)$ としておきます。この問題を解くためには、

$ans_v = C_v$の値の書かれたカードを X に加えてゲームを行った時のスコアの最大値

が 1 から $N+1$ までのすべての v について前計算できればよいですが、まずは最初に、v の値がひとつ与えられた時に、ans_v を計算する方法を考えます。

この問題は、二部マッチングを作る問題とも考えられます。そこで、$cnt[i] = C_i$以下の値を使っている Z のカードの数 $-i$ とすると、cnt の全要素が非負になっていれば条件を満たしていることになります。なので、配列 cnt を保持して、スコアを最大化しつつ、cnt の全要素を非負にすることを目標にアルゴリズムを実行します。まず、$cnt[i] = -i$ で初期化します。次に、全ての $i(1 \leq i \leq N)$ について、$cnt[A_i], cnt[A_i+1], ...cnt[N+1]$ の値を全て 1 増やします。これは、とりあえず X のカードを表面のままにしておくことに対応します。ここからできる操作は、次の二つです。

操作 1 $v \leq i \leq N+1$ なる全ての i について、$cnt[i]$ の値を 1 増やす。(新しいカードを使うことに対応)

操作 2 $B_j \leq C_i < A_j$ なる全ての i について、$cnt[i]$ の値を 1 増やす。(X のカード j をひっくり返すことに対応)

あとは、上記 2 種類の操作を行って cnt を非負にして、そのなかで操作 2 を行う回数を最小化すればよいです。これは、次のような貪欲法で計算できます。

まず、上記の操作は区間加算と見ることができるので、操作の行える区間の集合を S とする。また、区間の集合 T を空集合としておく。整数 k を、$1, 2, ...N+1$ と動かしながら、以下の操作をする。$0 \leq cnt[k]$ なら、次の k に移る。そうでない場合、k を含む区間を S から T へ移す。$cnt[k] < 0$ である限り、T から右端が最も右にある区間を取り出し、区間加算をする、という操作を繰り返す。$cnt[k] < 0$ であるのに操作が行えなくなれば、割り当てが存在しないと報告する。

T から右端が最も右にある区間を取り出す操作は、他の操作よりも損しないため、この貪欲法は最適になります。区間の集合を優先度付きキューで管理すれば、この貪欲法は $O(NlogN)$ で動くので、v がひとつ与えられた場合は解けます。

v が決まっていない場合について考えます。ここで、v を考えずに、操作 2 だけを考えて、先ほどの貪欲法を実行してみます。すると、不完全な解が得られます。この時の解で使用した区間の集合を U とします。すると、v を決めて先ほどの貪欲法を実行した時の解で使う操作 2 の区間の集合は、U の部分集合となることが証明できます。これは、アルゴリズムの動き方から分かります。よって、全ての v について、U から除いても問題ない区間を選んで、その個数を最大化する問題になります。まず、$cnt[i] < 0, i < v$ を満たす i が存在する時、その v については割り当てが存在しません。次に、U 内の区間全てについて、

区間の左端 = その区間を T から取り出した時の k の値

とします。この時、$cnt[i] < 0$ なる i を含むような区間は、v の値によらず取り除けません。そして、それ以外の区間からなる U の部分集合 W について、それを U から取り除いても問題ないことの必要十分条件は、W 内の区間の左端が全て v 以上で、かつどの二つの区間も重ならないことになります。

これは、後ろの方から見る区間スケジューリング問題と考えることができ、v を大きい方から求めていくと、効率的に全ての v について答えが求められます。このアルゴリズムは適切に実装すれば $O(NlogN)$ で動くので、この問題を解くことが出来ました。

Atcoder Grand Contest 013 Editorial

writer : maroonrk

2017 年 4 月 15 日

A : Sorted Arrays

Suppose that $A_1, A_2, \ldots, A_i$ is an (either non-increasing or non-decreasing) monotonous sequence. In this case, for each $j < i$, it never makes sense to cut the sequence between A_j and A_{j+1}. For example, in some solution, if the first two subarrays are $A_1, \ldots, A_j$ and $A_{j+1}, \ldots, A_k$ $(k > i)$, you can change them to $A_1, \ldots, A_i$ and $A_{i+1}, \ldots, A_k$ without making the solution worse.

Therefore, you can repeat the following: find the longest monotonous prefix from the sequence, delete it, again find the longest monotonous prefix from the remaining sequence, delete it, and so on.

Note that you should carefully handle equal numbers when you check whether a sequence is monotonous. First you should check whether there are two adjacent terms x, y such that $x < y$, and also check whether there are two adjacent terms x, y such that $x > y$. When both of these are found, the sequence is not monotonous.

This solution works in $O(N)$.

B : Hamiltonish Path

First, consider an arbitrary path. Check if this path satisfies the conditions. If yes, we are done. Otherwise, we can find a vertex that is adjacent to one of the endpoints of the path. We can extend the path using this vertex and we get a longer path. We repeat this process, and this process finishes after at most $N-1$ steps since the length of any path is at most $N-1$.

For example, we can implement this idea as follows:

1. Create a deque with single vertex: $\{1\}$.
2. Let x be the first element of the deque, and y be the last element of the deque. Repeat the following:
   (a) For each vertex z that is adjacent to x, check if z is contained in the path. If not, append it to the beginning of the deque and go to step 2.
   (b) For each vertex z that is adjacent to y, check if z is contained in the path. If not, append it to the end of the deque and go to step 2.
3. End the process. We found a solution.

In step 2(a) and step 2(b), each vertex is processed at most once. Therefore, this solution works in $O(M)$.

C : Ants on a Circle

We will restate the problem as follows:

Let's assign a card to each ant. Initially, the ant i has a card labelled with i. When two ants meet, instead of changing their directions, they swap their cards (and their directions won't be changed). After T seconds, we want to know the positions of each card.

Now, since the ants don't change directions, we can easily compute the positions of ants after T seconds. The main difficulty is to find the correspondence between ants and cards.

- The relative positions of the cards never changes. That is, the cards $1, 2, \ldots, N$ are aligned clockwise in this order.
- We can determine the card assigned to ant 1 after T seconds in the following way. If this ant is moving clockwise, each time it meets with another ant, the number on the card assigned to this ant increases by 1. Similarly, if this ant is moving counter-clockwise, each time the number decreases by 1. Thus, we can determine the number by counting the total number of meetings with other ants (which can be done in $O(N)$).

If we combine the two observations above, we can get the correspondence between ants and cards. This solution works in $O(NlogN)$.

D : Piling Up

Suppose that the box currently contains x red bricks and $N - x$ blue bricks. You will perform M operations from now, and each operation is one of the following four types:

- Operation 'RR': Take a red brick from the box, put a red brick and a blue brick into the box, and take a red brick from the box. This is possible only when $x > 0$, and x decreases by 1 after the operation.
- Operation 'RB': Take a red brick from the box, put a red brick and a blue brick into the box, and take a blue brick from the box. This is possible only when $x > 0$, and x remains unchanged after the operation.
- Operation 'BR': Take a blue brick from the box, put a red brick and a blue brick into the box, and take a red brick from the box. This is possible only when $x < N$, and x remains unchanged after the operation.
- Operation 'BB': Take a blue brick from the box, put a red brick and a blue brick into the box, and take a blue brick from the box. This is possible only when $x < N$, and x increases by 1 after the operation.

The problem asks the number of possible sequences of these operations.

Now, it is natural to define $dp[i][j]$: the number of possible sequences of the first i operations that end with j red bricks after the operations. The problem is that, we may count the same sequence multiple times this way.

For example, the following two sequences should not be distinguished because they have the same sequence of operations:

- Start with one red brick, perform an 'RR' operation, and end with two red bricks.
- Start with two red bricks, perform an 'RR' operation, and end with three red bricks.

In order to avoid double-countings, we add another restriction: you must perform at least one special operation. We call an operation "special" if it has the smallest possible value of x when the operation is performed. That is, a 'RR' or 'RB' operation when $x = 0$, or a 'BB' or 'BR' operation when $x = 1$. It is easy to see that any sequence of operations has exactly one way to perform with at least one special operation.

Now, define $dp[i][j][k]$ as the number of possible sequences of the first i operations that end with j red bricks after the operations, and additionally k is a boolean value that shows whether we have performed special operations. This solution works in $O(NM)$.

E : Placing Squares

The key observation in this problem is to rephrase the statement in a combinatorial way. For example, when we put k squares of sizes $a_1, \ldots, a_k$ from left to right (in a valid way), we want to count this configuration $a_1^2 a_2^2 \ldots a_k^2$ times. How can we do that?

It turns out that the problem is equivalent to the following:

> There are N cells. Some of the borders between two adjacent cells may be marked. How many ways are there to put some separators and red/blue balls such that:
>
> - There must be a separator at the left border of the leftmost cell and the right border of the rightmost cell.
> - You may put separators between two adjacent cells that are not marked.
> - You may put balls into cells (a cell may contain both red and blue balls).
> - Between each pair of adjacent separators, there must be exactly one red ball and exactly one blue ball.

The relation between this problem and the original problem is clear: the separators correspond to squares, and balls are added to add the factor of $a_1^2 a_2^2 \ldots a_k^2$.

Now the problem suddenly becomes very easy!

Define $dp[x][k]$ as the number of ways to determine the placement of separators and balls to the first x cells such that currently we put k balls after the last separator. This solution works in $O(N)$, and it is straightforward to improve this solution using matrix exponentiation. The solution with matrix exponentiation works in $O(MlogN)$.

F : Two Faced Cards

This problem is very challenging, and this is the main reason of the extended contest duration. Before tackling the original problem, let's prepare several things.

First, without loss of generality, we can assume that $C = \{0, 1, \ldots, N\}$. We should first sort the given C, and change the smallest number to 0, the second smallest number to 1, and so on. We should also change the values of A_i, B_i, D_i, E_i accordingly to preserve the relative relation between these numbers and C_i. More specifically, when $C_{i-1} < x \leq C_i$, the number x should be converted into $i - 1$.

Next, suppose that we have two arrays $p_1, p_2, \ldots$ and $q_1, q_2, \ldots$ (here each element is up to N). How can we check if we can permute the elements in p, q such that $p_i \leq q_i$ is satisfied for each i? One obvious solution is to sort each of them. However, in this solution, we use the following.

Consider an array of integers that is initially filled with zeroes. For each p_i, we add 1 to the interval $[p_i, N]$. For each q_i, we add -1 to the interval $[q_i, N]$. The condition is satisfied if and only if the array obtained this way consists of non-negative integers.

Now, let's return to the original problem. First, we create an array with zeroes, and for each i from 0 to N, add -1 to the interval $[i, N]$. This corresponds to the cards in deck Y. Then, for each i, we add 1 to the interval $[a_i, N]$. This corresponds to the cards in deck Z when we use front sides of all cards, except for the single added card.

This array may still contain negative numbers. We want to change all elements of this array to non-negative numbers by the following operations:

- Flip some cards in deck Z. This makes sense only when $b_i < a_i$, and it adds 1 to the interval $[b_i, a_i)$.
- Add a single card (depending on the query). It adds 1 to the interval $[x, N]$ for some integer x.

And our objective is to minimize the number of operations of the first type.

How can we handle queries? Suppose that the given query is (d, e). We consider two cases independently: whether we use the front side (d) or the back side (e) of the given card. Thus, in the problem above, it is sufficient if we can compute the minimum number of operations of the first type for each x from 0 to N.

Let's summarize what we have to do from now. (Note that the variable names in the problem below has nothing to do with the original problem, but the correspondence should be clear.)

> You are given an array of integers $a_0, \ldots, a_{N-1}$. This array may contain positive or negative numbers. You are also given a (multi)set of intervals $[L_i, R_i)$. In one operation, you can choose one of the intervals and add 1 to all elements of the array corresponding to this interval. Each interval can be used at most once. For each x between 0 and $N - 1$, solve the following problem:
>
> - How many operations are required to convert the array such that $a_0 \geq 0, a_1 \geq 0, \ldots, a_x \geq 0, a_{x+1} \geq -1, \ldots, a_{N-1} \geq -1$?

In the remaining part of this editorial, we will describe how to solve this problem.

First, find the rightmost element in the array that is smaller than -1. Regardless of the value of x, we must choose at least one of the intervals that contain this element. Among them, consider the interval whose left end is the leftmost (let's call it I). We will claim that, regardless of the value of x, we should use this interval.

Proof. Let p be the rightmost position such that $a_p < -1$. Let c_i be the distance from the current value of the i-th element to the target value of this element. For example, if this element is currently -4 and this element must be at least -1 after choosing some intervals, the distance is defined as 3. Now, regardless of the value of x, it is easy to see that $c_p > 0$ and for each $q > p$, $c_p > c_q$ is satisfied. In the optimal solution, there are at least c_p chosen intervals that contain the position p. Among them, consider the one with the smallest right end. It does no harm even if we change this interval to I. Thus, we can assume that we always use the interval I.

Now, we can greedily repeat this process. Traverse the elements of the array from right to left. When we find an element that is smaller than -1, we find an unused interval that contains this element and use it. In case there are multiple such intervals, choose the one with the smallest left end. This way, we get an array whose elements are at least -1 in all positions.

In the remaining part, it's easier to find a correct strategy. Since each element is now at least -1, we are now only interested in the leftmost occurrence of -1. Thus, we can repeat the following process. This time, traverse the elements of the array from left to right. When we find an element that is smaller than 0, we find an unused interval that contains this element and use it. In case there are multiple such intervals, choose the one with the largest right end.

Finally, we solved the problem. This solution works in $O(NlogN)$ (we need priority queues to simulate the solution above).