AGC 034 解説

yosupo, sigma425

2019 年 6 月 2 日

For International Readers: English editorial starts on page 8.

A: Kenken Race

まず、すぬけ君とふぬけ君は、少なくとも他方がいない場合に目的のマスへ移動できる必要があります。これは途中に 2 マス連続で岩があると不可能です。

- $C < D$ の場合はこれだけでよいです。実際に、まずはふぬけ君を目的地に動かし、次にすぬけ君を動かせば目的が達成できます。
- $C > D$ の場合は、すぬけ君がふぬけ君をどこかで追い抜く必要があります。追い抜くためには 3 マス連続の空きが必要です。これが存在するか、つまり B から D のどこかに、そのマスを中心として 3 連続の空きがあるかを判定すればいいです。

サンプルコード (C++): https://atcoder.jp/contests/agc034/submissions/5747486

B: ABC

まず、文字列中にある 'BC' は、操作によって分断されたり、また新しく増えたりしないことがわかります。そこではじめに文字列 s 中の 'BC' をすべて 'D' に置き換えてみます。すると、可能な操作は 'AD' を 'DA' にすることと同じです。あまった 'B' や 'C' はそこを超えて操作できないというブロックになるので、'A','D' からなる連続する極大な文字列全てに対して操作回数を求め、その和を計算すれば答えが得られます。これは転倒数なので、これまでに出てきた 'A' の個数 などを持ちながら左から文字列を舐めることで線形で計算可能です。

C: Tests

高橋くんがテスト i で取る点数を a_i とします (変数です)。

$D := A - B$ とします。 $D \geq 0$ にするのが目標です。

また、テスト i の D への寄与 (すなわち、 $c_i \times (a_i - b_i)$) を D_i とおきます。

テスト i に対して a_i を固定したときに、重要度 c_i をどう設定すればいいかは簡単に判定できます: $a_i \leq b_i$ なら重要度を l_i に、$a_i > b_i$ なら重要度を u_i にすればよいです。このことから、D_i を a_i の関数としてみると、以下の形になっていることがわかります。

$$D_i(a_i) = \begin{cases} -l_i \times (b_i - a_i) & (a_i \leq b_i) \\ u_i \times (a_i - b_i) & (a_i > b_i) \end{cases}$$

a_i を 0 から 1 ずつ増やしていくことを考えると、問題は次のように言い換えられます。

> はじめ、$D = \sum_{i=1}^{N} D_i(0)$ である。長さ X の整数列が N 個与えられる。i 個目の数列は、はじめの b_i 項が l_i で、残りの $X - b_i$ 項が u_i である。あなたは数列の要素をいくつか選んで D に足していき、$D \geq 0$ となるようにしたい。ただし各数列内では要素を前から順番に選んでいく必要がある。最小で何個取れば条件を達成できますか。

これは次のようにして解くことができます。まず答えで二分探索をし、 k 個取ることで総和を最大化する問題を考えます。実はこのとき、次の条件を満たす最適解が存在することがわかります:

- 1 個以上 $X - 1$ 個以下の要素を選ぶ数列が高々 1 つ

これは、中途半端に選んでいる 2 つの数列があったとすると片方を 1 つ増やし、もう片方を 1 つ減らす という操作を限界まで続けることで悪くはならないようにできること (これには $l_i \leq u_i$ が効いています) からわかります。

なので、$q := \lfloor k/X \rfloor$, $r := k - qX$ とおくと、q 個の数列から要素をすべて (X 個) 選び、($r > 0$ なら) 1 個の数列から要素を r 個 選ぶことになります。r 個選ぶ数列 i を決め打つと、選ぶべき残り q 個の数列は、数列 i を除いたもののうち要素の和が大きい q 個なので、まず事前に (判定問題にする前に) 数列を要素の和の大きい順にソートしておけば、各 i ごとに $O(1)$ の足し引きをすることで求まります。よって判定問題が $O(N)$ で解けました。したがってもとの問題も解けて、全体の計算量は $O(N(logN + logX))$ です。

D: Manhattan Max Matching

おまけ： $O(S \log N)$ でもこの問題は解くことが可能です

ペアのスコアを $|rx - bx| + |ry - by|$ とする代わりに、ペアごとにスコアを

- $(rx - bx) + (ry - by) = (rx + ry) + (-bx - by)$
- $-(rx - bx) + (ry - by) = (-rx + ry) + (bx - by)$
- $(rx - bx) - (ry - by) = (rx - ry) + (-bx + by)$
- $-(rx - bx) - (ry - by) = (-rx - ry) + (bx + by)$

の 4 つから好きなものを選ぶ、としても答えは変わりません。

ボールごとに、この 4 つのうちどれを選ぶか、というのを先に決めておくことにします。

つまり、赤いボールを 4 つの以下のタイプに分類します。

- タイプ 0: スコアに $(rx + ry)$ を加算する
- タイプ 1: スコアに $(-rx + ry)$ を加算する
- タイプ 2: スコアに $(rx - ry)$ を加算する
- タイプ 3: スコアに $(-rx - ry)$ を加算する

同様に青いボールも以下のように分類します。

- タイプ 0: スコアに $(-bx - by)$ を加算する
- タイプ 1: スコアに $(bx - by)$ を加算する
- タイプ 2: スコアに $(-bx + by)$ を加算する
- タイプ 3: スコアに $(bx + by)$ を加算する

そして、各タイプごとに赤と青の個数が等しければ実際にペアを構成することができます。このような分類のうち、スコアの最大を求めれば良いです。

これは最小費用流で解けます。1(始点) + N(赤いボールの操作) + 4(タイプ) + N(青いボールの操作) + 1(終点) 頂点のグラフを作り、

- 始点から (赤いボールの操作) へ、それぞれ容量 RC_i、コスト 0
- (赤いボールの操作) からタイプへ、それぞれ容量 ∞、コストは $-$(先述のスコア)
- タイプから (青いボールの操作) へ、それぞれ容量 ∞、コストは $-$(先述のスコア)
- (青いボールの操作) から終点へ、それぞれ容量 BC_i、コスト 0

の辺を張り、S(ボールの個数) 流せば良いです。負の重みは、適切なオフセットを各辺に足すことで解決できます。

辺数が $O(N)$、流量が $O(S)$ なので、計算量は $O(SN \log N)$ です。

E: Complete Compress

コマを集める頂点を固定した時に，$O(N)$ で解く方法を解説します。これが可能ならば全ての頂点に対し固定してみることで合計で $O(N^2)$ で解け，十分高速です。以後，木を頂点 1 を根とする根付き木として考えることにし、すべてのコマを頂点 1 に集めたいものとします。また、頂点 v の深さを $\mathrm{dep}(v)$ とします。

コマの深さの和を S とすると、S が奇数だと不可能、また偶数のときは最低 $S/2$ 回は操作が必要です。

実は以下の 2 つが成立します。

- もしコマを集められるならば、必ず $S/2$ 回の操作でコマを集められる (lemma A)。
- i 回目の操作を頂点 a_i, b_i のコマに行うとするとき、$\mathrm{dep}(\mathrm{lca}(a_i, b_i))$ が広義単調減少の最短手数の手順が存在する (lemma B)。

とりあえずこれらを正しいものとすると、$\mathrm{dp}(v) =$ (自分の部分木の中で完結する操作のみを行う時，操作回数の最大値) を木 DP で計算し、$\mathrm{dp}(1) = S/2$ を判定すれば解けます。

lemma A の証明

無駄な操作 (=深さの和を変えない操作) をする必要がないことを示します。最初の 1 回のみが無駄な操作で，その後は無駄な操作を行わない操作列があった時，無駄な操作を行わない操作列へ変換出来ることを示します。

最初の無駄な操作で頂点 $u, v(\mathrm{dps}(u) < \mathrm{dps}(v))$ から u', v' にコマを動かしたとします (前者をコマ A, 後者をコマ B とします)。次にコマ A に対し操作を行うタイミング (*) で，コマ B が u' より上にいるかどうかに注目し，場合分けをします。

- 上にいない: 無駄な操作を行わず，その後 (*) の直前まで同じ操作列を適用します。そして (*) でコマ A の代わりにコマ B を動かします。すると，元の操作列を (*) まで適用した後と同じ状態になります。
- 上にいる: (*) より前のタイミングで，B が A の真上に移動するタイミングが必ず存在するはず (**) です。仮に無駄な操作を行わず，その後同じ操作列を適用すると，実は (**) のタイミングで盤面は全く同じになります。

lemma B の証明

ある i について，$\mathrm{dep}(\mathrm{lca}(a_i, b_i)) < \mathrm{dep}(\mathrm{lca}(a_{i+1}, b_{i+1}))$ であったとします。この 2 回の操作を swap する，つまり $i+1$ 回目で操作するコマたちを i 回目に操作し，i 回目で操作するコマたちを $i+1$ 回目に操作することを考えます。すると，swap しても $i+1$ 回目の終了後の盤面が変化しないこと，そして swap 後の操作列での $a_i, b_i, a_{i+1}, b_{i+1}$ を $a'_i, b'_i, a'_{i+1}, b'_{i+1}$ とすると

- $\mathrm{dep}(\mathrm{lca}(a'_i, b'_i)) = \mathrm{dep}(\mathrm{lca}(a_{i+1}, b_{i+1}))$
- $\mathrm{dep}(\mathrm{lca}(a'_{i+1}, b'_{i+1})) = \mathrm{dep}(\mathrm{lca}(a_i, b_i))$

であることが示せます。よってこの swap を繰り返し適用することで，有限回の操作で求める操作列が得られます。

F: RNG and XOR

i が生成される確率 を $a_i(=A_i/S)$ とおきます。まず 0 が i になるまでの期待値は i が 0 になるまでの期待値と同じです。これを x_i とおきます。連立方程式を解けば x_i が求まるのは明らかですが、計算量は $O(2^{3N})$ となってしまい駄目です。詳しく連立方程式の形を見てみましょう。$2^N \times 2^N$ 行列で書くと次のようになります。

$Mx = c$ ただし、

$$M_{i,j} = \begin{cases} 1 & (i = 0 \wedge j = 0) \\ 0 & (i = 0 \wedge j \neq 0) \\ a_0 - 1 & (i \neq 0 \wedge i = j) \\ a_{i \hat{} j} & (otherwise) \end{cases}$$

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2^N-1} \end{pmatrix}$$

$$c = \begin{pmatrix} 0 \\ -1 \\ \vdots \\ -1 \end{pmatrix}$$

うまく式を変形して高速に x を求めるのが目標です。

$a_0 = a_0 - 1$ と置き直します。$\sum_{i=0}^{2^N-1} a_i = 0$ に注意してください。$i = 1, \cdots, 2^N - 1$ に対して、式 F_i をこの連立方程式の i 行目の式とします。すなわち、$\sum_{j=0}^{2^N-1} a_{i \hat{} j} x_j = -1$ のことです。$i = 0$ に対しても同様の式が求まるでしょうか? 答えは YES です。実際、 $\sum_i a_i = 0$ より、$-(F_1 + F_2 + \cdots + F_{2^N-1})$ という式を考えると、$\sum_{j=0}^{2^N-1} a_{0 \hat{} j} x_j = 2^N - 1$ を得られます。これを F_0 としましょう。

実はこの 2^N 個の式に対してアダマール変換のようなことをすることで x が高速に求まります。$i = 1, \cdots, 2^N - 1$ に対して、式 G_i を $\sum_j (-1)^{\mathrm{popcnt(i\&j)}} F_j$ とします。$i = 0$ については自明な式が得られるので無視します。すると G_i は次のような簡潔な形になることがわかります。まず $b_i := \sum_j (-1)^{\mathrm{popcnt(i\&j)}} a_j$ とおきます。G_i は $b_i \times \sum_k (-1)^{\mathrm{popcnt(i\&k)}} x_k = 2^N$ という形になります。

右辺は $i \neq 0$ からすぐ示せます。左辺については、任意の i, k に対し G_i の x_k に関する等式 $\sum_j (-1)^{\mathrm{popcnt(i\&j)}} a_{j \hat{} k} = (-1)^{\mathrm{popcnt(i\&k)}} b_i$ を示せばよいですが、これは a_j の係数を比較すると $(-1)^{\mathrm{popcnt(i\&(j\hat{}k))}} = (-1)^{\mathrm{popcnt(i\&k)}} \times (-1)^{\mathrm{popcnt(i\&j)}}$ が各 bit ごとにみて正しいことからわかります。

よって $i = 1, \cdots, 2^N - 1$ に対して $y_i := \sum_k (-1)^{\mathrm{popcnt(i\&k)}} x_k$ が求まります。y_0 は、 $\sum_i y_i = x_0 \times 2^N$ と $x_0 = 0$ から求まります。最後にアダマール変換で y から x を復元することでこの問題は解けました。計算量は $O(N2^N)$ です。

AGC 034 Editorial

yosupo, sigma425

June 2, 2019

A: Kenken Race

First, each of Snuke and Fnuke needs at least to be able to move to the destination square if the other person does not exist. They cannot do this if there are two consecutive rock squares on the way.

- If $C < D$, that's it. We can achieve the objective by, for example, moving Fnuke to his destination first and then moving Snuke to his.
- If $C > D$, Snuke has to overtake Fnuke somewhere on the way, which requires three consecutive empty squares. Thus, we additionally need to check if there are three consecutive empty squares centered at somewhere between B and D.

B: ABC

First, we can see that the operation does not separate existing `BC`s or produce new `BC`s in the string, so let us replace each occurrence of `BC` in the string s with `D`. Then, the operation is now equivalent to change `AD` to `DA`. The remaining `B`s and `C`s are now obstacles that block the operation, so we can obtain the answer by finding the number of possible operations for each maximal contiguous substring consisting of `A` and `D` and summing them up. This count is equivalent to the inversion number, which we can compute in linear time by scanning the string from the left, maintaining the count of `A`s appeared so far, for example.

C: Tests

Let a_i be the variable representing Takahashi's score on Test i, and $D := A - B$. Our objective is to have $D \geq 0$.

Then, let D_i be the contribution of Test i in D, that is, $c_i \times (a_i - b_i)$.

When we fix a_i for Test i, we can easily determine the optimal choice of the importance c_i: we should set c_i to l_i if $a_i \leq b_i$, and set c_i to u_i if $a_i > b_i$. Thus, if we see D_i as a function of a_i, it looks as follows:

$$D_i(a_i) = \begin{cases} -l_i \times (b_i - a_i) & (a_i \leq b_i) \\ u_i \times (a_i - b_i) & (a_i > b_i) \end{cases}$$

Consider incrementing a_i by 1 at a time from 0, and we can rephrase the problem as follows:

> Initially, $D = \sum_{i=1}^{N} D_i(0)$. You are given N integer sequences, each of length X. The first b_i terms in the i-th sequence are l_i, and the remaining $X - b_i$ terms are u_i. You want to choose some terms in the sequences and add them to D so that $D \geq 0$. In each sequence, you have to choose terms in order from front to back. At least how many terms do you need to choose to achieve the objective?

We can solve it as follows. First, let us do a binary search on the answer, and consider the problem to maximize the sum by choosing k terms. Then, we can see that there exists an optimal solution that satisfies the following condition:

- There is at most one sequence in which 1 between $X - 1$ terms are chosen.

This is because, if two sequences are partially used, we can repeatedly choose one more term in one of them and choose one less term in the other as many times as possible, and the result does not get worse, because $l_i \leq u_i$.

Thus, let $q := \lfloor k/X \rfloor$, $r := k - qX$, and we will choose all the terms in q of the sequences, and choose r terms in one of the sequences if $r > 0$. If we decide to choose r terms in Sequence i, the other q sequences that should be chosen are the q sequences with the largest sums of terms excluding Sequence i, which we can find in $O(1)$ time for each i by sorting the sequences in descending order of the sum of their sums of terms in advance. Now we have solved the decision problem in $O(N)$ time, and also the original problem with the total time complexity $O(N(\log N + \log X))$.

D: Manhattan Max Matching

Bonus: This problem can also be solved in $O(S \log N)$ time.

Without affecting the score, we can assume that instead of getting the score of $|rx - bx| + |ry - by|$ for a pair, we can choose one of the following scores for a pair:

- $(rx - bx) + (ry - by) = (rx + ry) + (-bx - by)$
- $-(rx - bx) + (ry - by) = (-rx + ry) + (bx - by)$
- $(rx - bx) - (ry - by) = (rx - ry) + (-bx + by)$
- $-(rx - bx) - (ry - by) = (-rx - ry) + (bx + by)$

For each ball, let us decide in advance which of these four to use.

That is, we will classify the red balls into the following four types:

- Type 0: adds $(rx + ry)$ to the score.
- Type 1: adds $(-rx + ry)$ to the score.
- Type 2: adds $(rx - ry)$ to the score.
- Type 3: adds $(-rx - ry)$ to the score.

We will also classify the blue balls into the following four types:

- Type 0: adds $(-bx - by)$ to the score.
- Type 1: adds $(bx - by)$ to the score.
- Type 2: adds $(-bx + by)$ to the score.
- Type 3: adds $(bx + by)$ to the score.

Then, we can form the pairs if the number of red balls and that of blue balls are equal for each type. We want to find the maximum score of such a classification.

We can solve it as a minimum-cost flow problem. Let us build a graph with 1 (source) $+N$ (operations with red balls) +4 (types) $+N$ (operations with red balls) +1 (sink) vertices and the following edges:

- from the source to each "operations with red balls" vertex: an edge of capacity RC_i and cost 0
- from each "operations with red balls" vertex to each "type" vertex: an edge of capacity ∞ and cost $-$(the score above)
- from each "type" vertex to each "operations with blue balls" vertex: an edge of capacity ∞ and cost $-$(the score above)
- from each "operations with blue balls" vertex to the sink: an edge of capacity BC_i and cost 0

Then send the flow of S (the number of balls). We can resolve negative costs by adding some offset to each edge.

There are $O(N)$ edges, and the amount of flow is $O(S)$, so the time complexity is $O(SN \log N)$.

E: Complete Compress

Let's fix certain vertex as the root, and check if we can move all tokens to the root. We want to do this in $O(N)$ (then the entire solution will be $O(N^2)$).

Suppose that there is a sequence of operations that moves all tokens to the root. Then, we can prove that we can do that even with the following additional restrictions:

Lemma A. Let's call an operation "bad" if this operation moves one of the tokens downward. (In other words, one of the two chosen tokens is the descendant of the other.) There is a valid sequence of operations without any bad moves.

Proof. Let's take a sequence of operations with at least one bad move. We show that we can alwats reduce the number of operations after the last bad move; then, by repeating this process, we can prove the lemma.

Suppose that in the last bad move, we choose token A at vertex u and token B at vertex v (and u is an ancestor of v). There are several cases depending on the next move.

- If the next move doesn't involve A and B, obviously, we can swap those two moves.
- If the next move is between A and C for some C, we can replace $(A, B) \to (A, C)$ with (B, C).
- If the next move is between B and C for some C, we can swao the two moves unless the distance between u and v is two; if the distance is two, we can simply drop the move between A and B and the multiset of positions of the tokens won't change.

Lemma B. Let's call an operation "special" if the LCA of two chosen tokens is the root. There is a valid sequence of operations that starts with zero or more non-special moves, followed by zero or more special moves.

Proof. This is easier to prove; if there is a non-special move directly after a special move, we can always swap those two moves.

Now, let's return to the original problem. For each vertex x (in the order from leaves to the root), we compute the following values when we consider the subtree rooted at x:

- The number of tokens in the subtree.
- $high$: the sum of distances from each token to the root (i.e., x) in the initial configuration.
- low: the minimum possible value of the sum of distances from each token to the root, when we are allowed to perform arbitrary operations within the subtree.

Then, the set of possible values of "the sum of distances" is $low, low+2, ..., high$. Suppose that $y_1, ..., y_k$ are children of x. To compute low for x, we do the following:

- First, for each y_i, we choose the value s_i: it should be a value between low and $high$ of y_i, with the correct parity.

- Replace s_i by $s_i + cnt_{y_i}$.
- $low = max(\sum s_i mod 2, \max s_i * 2 - \sum s_i)$. (This corresponds to moves whose LCA are x. We should choose s_i that minimizes this value.)

The answer is "Yes" if the value low for the root is zero.

F: RNG and XOR

Let $p_i(= A_i/S)$ be the probability that i is generated. Let x_i be the i-th answer. This is the same as the expected number of moves to reach zero when we start from i, so we get the following for each nonzero i:

$$x_i = \sum_j p_j x_{i \hat{} j} + 1 \tag{1}$$

Let $\bigoplus$ denote XOR convolution. Then, these equations can be written as follows:

$$(x_0, x_1, \ldots, x_{2^N-1}) \bigoplus (p_0, p_1, \ldots, p_{2^N-1}) = (?, x_1 - 1, \ldots, x_{2^N-1} - 1) \tag{2}$$

Here, in the XOR convolution, the product of the i-th term of the first sequence and the j-th term of the second sequence is added to the $i \hat{} j$-th term of the third sequence (all zero-based).

Since $\sum p_i = 1$, the sum of the first sequence and the sum of the third sequence must be equal. Thus, we get:

$$(x_0, x_1, \ldots, x_{2^N-1}) \bigoplus (p_0, p_1, \ldots, p_{2^N-1}) = (x_0 + 2^N - 1, x_1 - 1, \ldots, x_{2^N-1} - 1) \tag{3}$$

and thus

$$(x_0, x_1, \ldots, x_{2^N-1}) \bigoplus (p_0 - 1, p_1, \ldots, p_{2^N-1}) = (2^N - 1, -1, \ldots, -1) \tag{4}$$

Now we can use an algorithm similar to Hadamard Transform. (For simplicity, assume that $N = 3$.)

First we get the following two equations:

$$(x_0 + x_1, x_2 + x_3, x_4 + x_5, x_6 + x_7) \bigoplus (p_0 - 1 + p_1, p_2 + p_3, p_4 + p_5, p_6 + p_7) = (6, -2, -2, -2) \tag{5}$$

$$(x_0 - x_1, x_2 - x_3, x_4 - x_5, x_6 - x_7) \bigoplus (p_0 - 1 - p_1, p_2 - p_3, p_4 - p_5, p_6 - p_7) = (8, 0, 0, 0) \tag{6}$$

By repeating the same process recursively, we can (almost) get the values $x_0+x_1, x_2+x_3, x_4+x_5, x_6+x_7$ and $x_0 - x_1, x_2 - x_3, x_4 - x_5, x_6 - x_7$, and we can get the values x_i.

There is a small catch: in one of the deepest recursion, we get the following:

$$(x_0 + \ldots + x_7) \bigoplus (p_0 - 1 + p_1 + \ldots + p_7) = (0) \tag{7}$$

This gives no information for the first term because both the second and the third terms are zeroes.

However, it is easy to see that if we change the value for $x_0 + \ldots + x_7$ here, we add the same constant to all x_i. Thus, we can choose an arbitrary value here, and then modify the values we get (x_i) using $x_0 = 0$.

This solution works in $O(N2^N)$ time.