AGC 027 解説

writer : camypaper, sugim48

2018 年 9 月 15 日

For international readers: English editorial starts from page 9.

A: Candy Distribution Again

まず，$\sum_{i=1}^{N} a_i = x$ の場合，答えは N であり，そうでない場合，答えは $N-1$ 以下です．以降，$\sum_{i=1}^{N} a_i \neq x$ と仮定します．

子供たちの部分集合 $S \subsetneq \{1, 2, \ldots, N\}$ について，S 全員を喜ばせるための必要十分条件は $\sum_{i \in S} a_i \leq x$ です．(各 $i \in S$ について子供 i に a_i 個のお菓子を配った後，余ったお菓子は $j \notin S$ であるような子供 j に押し付ければよいです．) よって，$\sum_{i \in S} a_i \leq x$ を満たすような S の最大サイズを求めるのが目標です．これは，a を昇順にソートした後，$\sum_{i=1}^{k} a_i \leq x$ を満たすような最大の k を求めればよいです．もちろん，k が N である場合，代わりに $N-1$ を出力することを忘れずに．

B: Garbage Collector

ロボットがゴミを拾ったり，置くときにエネルギーを X 消費するのが厄介です．まずはこれを除去することを考えます．全てのゴミはちょうど 1 回ずつロボットに回収されるため，ゴミを拾うときに消費するエネルギーは 0 と考えても問題ありません (最後に NX を足せばよいです)．ロボットがゴミをゴミ箱に入れた回数 (これを k とします) を固定して考えてみます．すると，ゴミを捨てるときに消費するエネルギーも 0 として考えることができます (同様に，最後に kX を足せばよいです)．このとき，以下の問題が解ければよいことになります．

> k 台のロボットが原点から出発し，それぞれ 0 個以上のゴミを拾い，原点に戻ってくる．全てのゴミが拾われる場合に必要な最小限のエネルギーを求めよ．

各ロボットは拾うことに決めたゴミのうち，遠いほうから 1 つずつ拾うのが明らかに最適です．ロボットが位置 x にあるゴミを i 番目に拾うときに追加で必要なエネルギーを $E(i,x)$ とします．$E(i,x)$ は以下の式で表せます．

$$E(i,x) = \begin{cases} 5x & (i = 1) \\ (2i+1)x & (otherwise) \end{cases} \tag{1}$$

$1 \leq i < j$ を満たす任意の (i,j) について $E(i,x) \leq E(j,x)$ が成立していることが重要です．詳細は省きますが，この結果より右から i 番目のゴミは $\lceil \frac{i}{k} \rceil$ 番目に拾えばよいことが言えます．

よって，k を固定した場合の答えは $(N+k)X + \sum_{i=1}^{N} E(\lceil \frac{i}{k} \rceil, a_{N-i+1})$ となります．これを愚直に計算すると計算量は $O(N^2)$ となり，部分点を獲得することができます．

$\lceil \frac{i}{k} \rceil$ が等しいような i たちについて消費するエネルギーはまとめて計算することができます．たとえば，$E(\lceil \frac{1}{k} \rceil, a_N) + E(\lceil \frac{2}{k} \rceil, a_{N-1}) + \ldots + E(\lceil \frac{k}{k} \rceil, a_{N-k+1}) = E(1, \sum_{i=1}^{k} a_{N-i+1})$ です．累積和を用いることで $\sum_{i=1}^{k} a_{N-i+1}$ は前計算 $O(N)$，取得 $O(1)$ で処理できます．以上より k を固定したときの答えは $O(N/k)$ で求めることができます．$O(\sum_{i=1}^{N} \frac{1}{i}) = O(\log N)$ が成立するため，この改善により計算量は $O(N \log N)$ となり $N = 2 \times 10^5$ の場合でも十分高速に動作します．

k の値によっては $(N+k)X + \sum_{i=1}^{N} E(\lceil \frac{i}{k} \rceil, a_{N-i+1})$ が 64 bit 符号付き整数で表せない場合があるため，オーバーフローには注意が必要です．

C: ABland Yard

全ての文字列を列挙して，構成可能かどうかを実際に判定することは当然困難です．そこで，任意の文字列が構成可能である，と同値な条件を探してみましょう．

s を AABB を 10^{100} 回繰り返した文字列とします．s が構成不可能ならば，答えは No です．s が構成可能ならば，実は全ての文字列が構成可能です．

s が構成可能なとき，与えられるグラフには AABB の繰り返しで表されるような (辺素とも点素とも限らない) 閉路が必ず存在しています．これは鳩ノ巣原理から明らかです．このとき，そのような閉路に含まれる全ての頂点はラベルが A であるような頂点と，B であるような頂点の両方に隣接しています．よって，この閉路上を移動し続けることで任意の文字列が構成可能です．

s を実際に構成するのは時間がかかりすぎますが，AABB の繰り返しで表される閉路があるかどうかを判定するだけならば簡単です．例えば，隣接している頂点のラベルの種類が 1 種類以下であるような頂点を取り除く，という操作を繰り返して全ての頂点が取り除かれるかどうかを判定する，という方法があります．これはトポロジカルソートの要領で $O(N+M)$ で実行可能です．その他，頂点を 2 倍に増やし適切に辺を張った有向グラフに閉路があるかを判定する方法もあります．

D: Modulo Matrix

$N = 500$ の場合について解くことができれば，それ以外のケースは容易に解くことができます．以降，$N = 500$ の場合についてだけ考えます．

解の一例について説明します．隣接する 2 つの数について，最大値を最小値で割ったあまりが一定という条件が最も複雑です．そこで，この関係が容易に満たせるような数の割り当て方を考えてみます．

$N \times N$ のマス目を以下の図 1 のように市松模様に塗り分けることを考えます．黒マスについて数を書き込んだのち，白マスは「隣接する全ての黒マスに書かれた数の最小公倍数 $+1$」を書き込むことにします．すると，どの隣接する 2 つの数についても最大値を最小値で割ったあまりが必ず 1 となります．

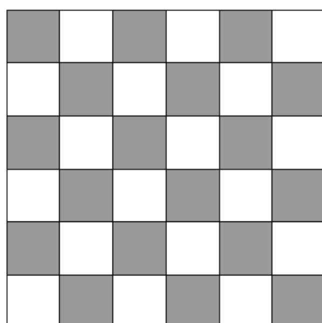

図 1　市松模様

残った問題は数の最大値が 10^{15} 以下であり，数が相異なるようにすることです．白と黒の市松模様ではなく，2 種類の斜め方向の線の重ね合わせの模様で考えてみます．$2N$ 本の線に，素数を小さい方から割り当てて 2 つの数の積を書き込むことにすると，どの数も相異なり，数の最大値が 4 つの素数の積 $+1$ 以下にすることができます．1000 番目の素数は 7919 のため，最大値が 10^{15} 以下になるようにするには割り当て方に多少の注意が必要ですが，さほど特別な工夫をせずとも達成することができます．

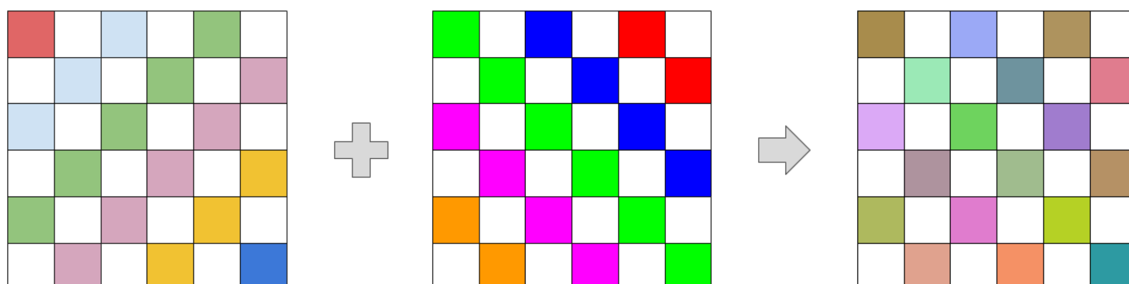

図 2　市松模様の塗り分け

E: ABBreviate

s 中に同一文字が隣り合う場所がない場合，答えは 1 です．以降，s 中に同一文字が隣り合う場所があると仮定します．

まず，操作に関する不変量を見つけましょう．a の重みを 1，b の重みを 2 とし，文字列 s の重みの総和を 3 で割った余りを $p(s)$ と定義します．このとき，$p(s)$ は操作の前後で不変です．

次に，文字列 s と文字 $c \in \{\mathsf{a}, \mathsf{b}\}$ について，s に操作を繰り返して c が得られるための必要十分条件を考えましょう．必要十分条件 $(*)$ は次のとおりです:

- $s = c$．または，
- $p(s) = p(c)$，かつ s 中に同一文字が隣り合う場所がある．

必要性は明らかです．十分性を示します．いま，$|s| \geq 2$，かつ $p(s) = p(c)$，かつ s 中に同一文字が隣り合う場所があることを仮定します．次のように操作を行えば，「s 中に同一文字が隣り合う場所がある」という条件を保ったまま，s を 1 文字分だけ短くすることができます: s 中の同一文字が連続する場所であって極大なものを適当に選び，t とします．t が s 全体である (すなわち，s が同一文字のみからなる) 場合，s の先頭 2 文字に対して操作を行えばよいです．(このとき，不変量の条件より，操作前は $|s| \neq 3$ であることに注意．) t が s 全体でない場合，t の前後いずれかに異なる文字が存在します．異なる文字が t の前に存在するならば，t の先頭 2 文字に対して操作を行えばよいです．逆に，異なる文字が t の後に存在するならば，t の末尾 2 文字に対して操作を行えばよいです．以上より，十分性を示せました．

問題は次のように言い換えられます:

> 文字列 s が与えられる．次の条件 (♣) を満たすような文字列 t は何通りか？
>
> 条件 (♣): うまく s を $|t|$ 個の区間に分割すると，各区間 (と対応する t の文字) において，条件 $(*)$ が成り立つ．

まず，t をひとつ固定したとき，条件 (♣) が成り立つか判定する方法を考えましょう．実は，これは次のようにして可能です: $j = 1, 2, \ldots, |t|$ の順に，$p(\cdot)$ の値が t_j と一致するような s の区間を左から貪欲に取っていきます．$|t|$ 個の区間を取りきることができ，かつ余った文字列の $p(\cdot)$ の値が 0 ならば，条件 (♣) が成り立ちます．例えば，$s = \mathsf{aaababbabababbaaaba}$, $t = \mathsf{abab}$ の場合，$s = \underline{\mathsf{a}}\ \underline{\mathsf{aa}}\ \underline{\mathsf{babb}}\ \underline{\mathsf{abababb}}\ \mathsf{aaaba}$ のように区間を取っていき，余った文字列について $p(\mathsf{aaaba}) = 0$ が成り立つので，この例では条件 (♣) が成り立ちます．

この判定法の正当性を確かめましょう．t がこの判定法に失敗した場合，明らかに条件 (♣) は成り立ちません．以降，t がこの判定法に成功したと仮定し，条件 (♣) が成り立つことを示します．まず，区間を貪欲に取っていることから，各区間は「長さ 1 である」または「同一文字が隣り合う場所がある」のいずれかの性質を持ち，条件 $(*)$ を満たします．よって，残る懸念事項は余った文字列の扱いのみになりました．実は，適切に区間を組み換えることで，余った文字列をいずれかの区間に含めることができます．その方法を説明します．余った文字列に同一文字が隣り合う場所がある場合，余った文字列を最後の区間に含めればよいです．以降，余った文字列に同一文字が隣り合う場所がないと仮定します．この

とき，余った文字列の $p(\cdot)$ の値が 0 なので，余った文字列は abab$\cdots$ab または baba$\cdots$ba の形をしています．ここで，(右から k 番目の区間) + $\cdots$ + (右から 1 番目の区間) + (余った文字列) の形の文字列を考えます．まず，この文字列に同一文字が隣り合う場所が含まれるような最小の k をとります．例えば，$s = \cdots$ $\underline{\mathtt{b}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ ababab の場合，$k = 4$ です．また，$s = \cdots$ $\underline{\mathtt{babaa}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ bababa の場合，$k = 5$ です．次に，右から $1, 2, \ldots, k-1$ 番目の区間を s の右端に詰めます．最後に，余った文字列を右から k 番目の区間に含めます．例えば，$s = \cdots$ $\underline{\mathtt{b}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ ababab の場合，$s = \cdots$ $\underline{\mathtt{bbababa}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ となります．また，$s = \cdots$ $\underline{\mathtt{babaa}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ bababa の場合，$s = \cdots$ $\underline{\mathtt{babaabababa}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ $\underline{\mathtt{b}}$ $\underline{\mathtt{a}}$ となります．このように区間を組み換えることで，余った文字列をいずれかの区間に含めることができ，かつ各区間が条件 $(*)$ を満たすようにできます．以上より，上述の判定法の正当性を確かめられました．

問題は，上述の判定法に成功するような t の数え上げに帰着されました．これは，文字列の部分列の数え上げと同様の DP によって可能です．時間計算量は $O(|s|)$ です．

F: Grafting

直感的には，操作回数の最大値は $N-1$ 以下になりそうです．(この直感は実際には間違っていますが) しばらくの間，答えは $N-1$ 以下であることを仮定しておきます．

答えが $N-1$ 以下のとき，操作されない頂点 r が必ず存在します．r を全探索しましょう．A, B ともに，r を根とした根付き木として考えます．r から順番に子孫を見ていきます．操作されない頂点 u を見ているとき，頂点 v が A においても B においても u の直接の子ならば，v もまた操作されない頂点です．そうでないなら，v は操作される必要があり，A においても B においても v を根とする部分木に含まれる頂点は全て操作される必要があります．

操作されるべき全ての頂点たちについて，valid な操作順序が存在するかどうかを調べましょう．操作すべき頂点どうしをつなぐ辺について，A においては子から親に向かって，B においては親から子に向かって辺を向き付けます．以下の図 3 はサンプル 2 のケース 1 です．A, B を操作されない頂点たちを無視して 1 つのグラフにまとめてトポロジカルソートすることが可能ならば，valid な操作順序が存在します．例えば，$(1, 4, 6, 8, 3, 5)$ の順番で操作をすれば A を B にすることができます．辺の数は $O(N)$ で抑えられるので，答えが $N-1$ 以下の場合は $O(N^2)$ で答えを求めることができます．$T \leq 20, N \leq 50$ より十分高速です．

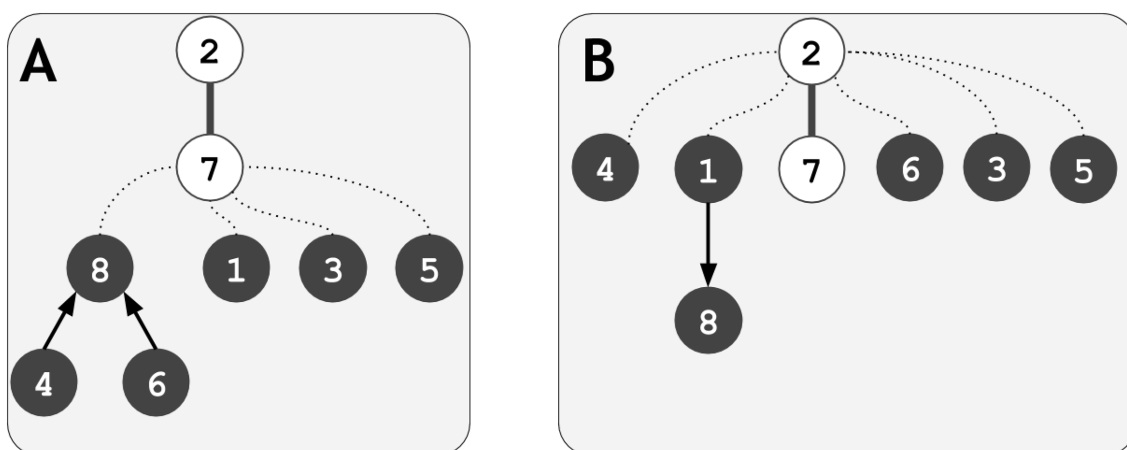

図 3　サンプル 2 ケース 1

上述の通り，操作回数が N の場合が存在します．図 4 はその例です．先程のアルゴリズムはこのケースではうまく動作しません (実際に試してみるとよいです)．このような場合であっても 1 回目の操作を実際に行うことで，操作されない頂点を作ることができます．1 回目の操作方法は $O(N^2)$ 通りなので，各ケースについて $O(N^3)$ で答えを求めることができます．$T \leq 20, N \leq 50$ より十分高速です．

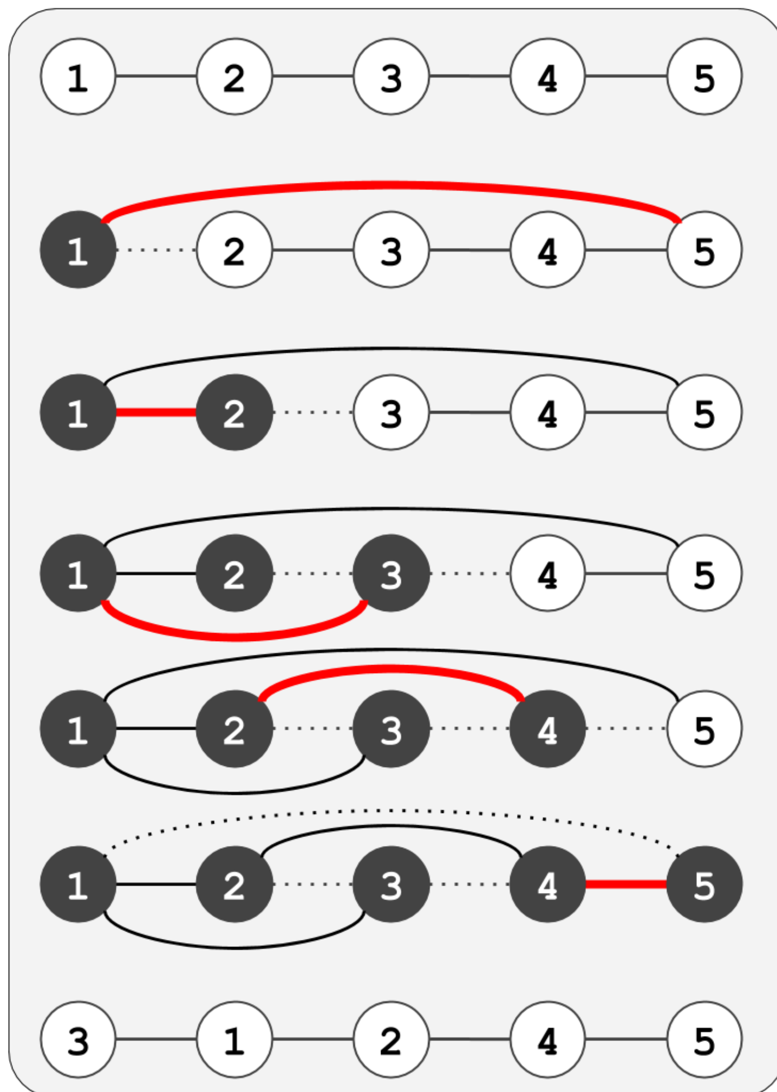

図 4　操作回数が N 回になるケース

AGC 027 Editorial

writer : camypaper, sugim48

September 15, 2018

A: Candy Distribution Again

In case $\sum_{i=1}^{N} a_i = x$, the answer is N, otherwise the answer is $N-1$. From now on, we assume that $\sum_{i=1}^{N} a_i \neq x$.

Let $S \subsetneq \{1, 2, \dots, N\}$ be a subset of children. We can satisfy all children in S if and only if $\sum_{i \in S} a_i \leq x$. (If this inequality holds, we can give a_i candies for each child i in S, and arbitrarily distribute remaining candies among remaining children.)

Thus, our objective is to find the maximum size of S that satisfies $\sum_{i \in S} a_i \leq x$. To do this, we first sort a in ascending order, and compute the maximum k such that $\sum_{i=1}^{k} a_i \leq x$. In case $k = N$, don't forget to print $N-1$ instead.

B: Garbage Collector

Let's ignore the cost of collecting trashes: since each trash will be collected exactly once, we can ignore it and add NX at the end.

Suppose that the robot starts from the origin, takes some subset of trashes, returns to the origin, and put all trashes it took into the trash bin. When the subset of trashes is fixed, the following strategy is optimal: The robot first moves to the position of the most distant trash (in the subset). Then, it directly returns to the origin. During the return trip, it takes trashes whenever it meets the trash in the subset.

The cost of doing this can be calculated as follows. Suppose that the i-th most distant trash is at position x. Let's define $E(i, x)$ as

$$E(i, x) = \begin{cases} 5x & (i = 1) \\ (2i + 1)x & (otherwise) \end{cases} \quad (1)$$

The cost is the sum of $E(i, x)$, plus X for putting trash into the bin.

In general, the movement of robot is of the following form. It consists of k phases. In each phase, the robot starts from the origin, takes a subset of trashes, and returns to the origin.

Let's fix the value of k. Since the cost for putting trash into the bin (kX) is a constant, we want to minimize the value comes from E.

To compute the value comes from E, we multiply each coordinate by some coefficient, and take their sum. The coefficient is $5, 5, 7, 9, 11, \ldots$ when the corresponding trash is the $1, 2, 3, 4, 5, \ldots$-th distant trash in its phase, respectively. Clearly, this value becomes the minimum when the coefficient 5 is assigned to the $2k$ most distant trashes, 7 is assigned to the next k distant trashes, and so on.

By using prefix sums, we can get this value in $O(N/k)$ time for a fixed k. Since $O(\sum_{i=1}^{N} \frac{1}{i}) = O(\log N)$, by trying all possible values of k, we can solve the problem in $O(N \log N)$ time in total.

Note that in some values of k you may get values greater than the limit of 64-bit integers during calculations. Be careful with overflow.

C: ABland Yard

Let s be the infinite repetition of the string `AABB`. If we can't make s, the answer is `No`. If we can make s, it turns out that we can make arbitrary strings.

Suppose that we can make s. Then, there must be a (not necessarily simple) cycle that is a repetition of `AABB`. Each vertex on this cycle is adjacent to both a vertex with `A` and the vertex with `B`. Thus, by keep moving on this cycle, we can make arbitrary strings.

There are various ways to check if we can make s. For example, we keep removing a vertex that is adjacent to only one or zero types of vertices. If a non-empty graph remains after this process, the graph contains s. It's also possible to construct an extended directed graph whose vertices correspond to pairs (the vertex in the original graph, the position in s modulo four), and check if this graph is a DAG.

D: Modulo Matrix

How can we make sure that "max mod min" is a constant? One natural way is the following.

First, paint a $N \times N$ board like a chessboard, as in the picture 1. Write arbitrary numbers on black cells. Then, on white cells, write "The LCM of all neighboring black cells" plus one. This way, the value "max mod min" is always 1.

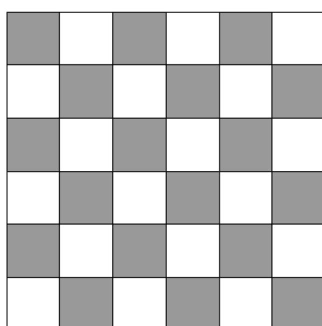

図 1　Chessboard

The main trouble is that, if we don't write numbers on white cells in a right way, the LCMs can be big and exceed the 10^{15} limit. We want to make sure that the LCM of four white numbers (that are adjacent to the same black cell) is always small.

One possible way is as follows. See the following picture. A white cell is always at the intersection of two diagonals. There are $2N$ diagonals in total, and we assign a small prime number on each diagonal. The value in a white cell is the product of two primes assigned to the two diagonals passing through it.

Then, the numbers on black cells will be "the product of four small primes" plus one, which is small enough. (The 1000-th prime is 7919. For example, you can assign the first 500 primes to one direction, and the next 500 primes to the other direction.)

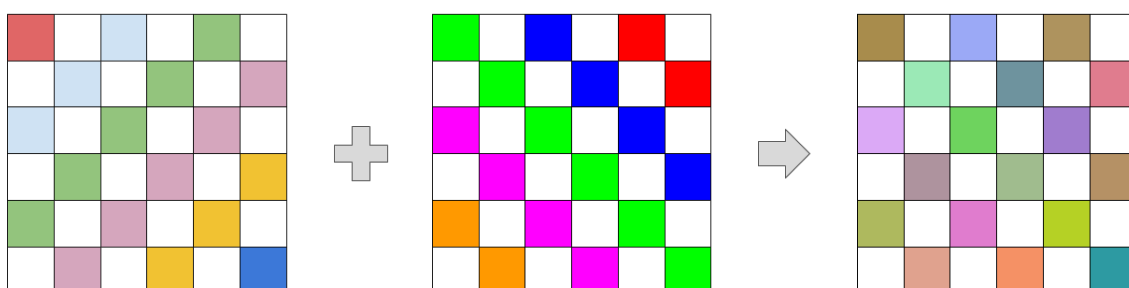

図 2　Diagonals

E: ABBreviate

Let $p(s)$ be the sum of all letters in s, modulo 3. Here, we assume that `a` is 1 and `b` is 2. This is an invariant.

When can we convert a string s into a single letter c? It turns out that the conditions are folloing $(*)$:

- $s = c$. or,
- $p(s) = p(c)$ and s contains two adjacent same letter.

It's clear that these conditions are necessary. Let's prove that this is sufficient. Suppose that $|s| \geq 2$, $p(s) = p(c)$, and s contains two adjacent same letter (i.e., s is not like "ababab..."). Then, by observing a few cases carefully, it turns out that we can always keep performing operations on this string until $|s|$ becomes 1. Since p is an invariant, when $|s|$ becomes 1, it must be c.

We can restate the original problem as follows:

> You are given a string s. Count the number of strings t that satisfies the following:
>
> By partitioning s into $|t|$ intervals properly, the condition $(*)$ holds for each corresponding pair of an interval and a letter in t.

Let's fix a string t, and check if we can divide s into $|t|$ parts as described above.

First, in case t doens't have two adjacent same letters, the condition is obviously $s = t$. Otherwise, we do this as follows. For each $j = 1, 2, \ldots, |t|$ in this order, take the shortest possible interval of s such that the value of $p(\cdot)$ matches t_j. For example, when $s =$ `aaababbababababbaaaba`, $t =$ `abab`, the intervals are $s =$ `a` `aa` `babb` `abababb` `aaaba`. (`aaaba` is the remaining part). It turns out that the condition is equivalent to the following: we can successfully take all $|t|$ intervals and the $p(\cdot)$ value of the remaining part is 0.

Again, it's clear that these conditions are necessary. Let's prove that this is sufficient. Since the p value of the remaining part is 0, by attaching it to the last interval, we can almost always get a desired partition. The only concern is that, this way the last part can be of the form "ababab...". However, it's not hard to see that this can be avoided by properly rearranging the intervals.

We can count such t by a simple DP. This solution works in $O(|s|)$ time.

F: Grafting

Suppose that there exists a vertex r such that we never perform an operation on it (i.e., paint it black). (It's possible that we perform an operation on each vertex exactly once - this case is harder, and we handle it later.)

Once we fix such r, we can get almost all information about the operations, as follows:

- Suppose that in both A, B a vertex v is adjacent to the root r. What happens if we move this edge? First, to move this edge, we must perform an operation on v (because we are not allowed to perform an operation on r). Then, immediately before we span an edge between v and r again, we must perform an operation on r again. However this means that we perform two operations on v. Thus, we must never move the edge between r and v.
- Suppose that in both A, B a vertex u is adjacent to v (as mentioned above, it is adjacent to the root in both trees). Then, by a similar reason, we must never move an edge between u and v. This way, by running a DFS from the root, we can get a set of edges that must not be moved. Let's call them "fixed edges".
- If an edge between vertices a and b exists in one of the trees but not in the other tree, this edge must be moved. All edges adjacent to "fixed edges" we got from the DFS above satisfy this property (otherwise more edges will be included in the set of fixed edges by DFS), so all such edges must be moved.
- Consider two fixed edges and a path between them. All edges on the path can't be moved because their endpoints can't be leaves. Thus, the set of fixed edges are connected. This means that we now fully get the information about fixed edges: the edges we got from the DFS are fixed, all other edges are not fixed (and must be moved).
- If a vertex is incident to at least one fixed edge, we can never perform operations on the vertex, because whenever it becaomse a leaf it is incident to a fixed edge. We call such vertices "fixed vertices". Now we get all information about fixed edges and vertices - and by their definition, the number of fixed edges and the number of fixed vertices are always the same.
- We can also uniquely determine "directions". Suppose that r is the root of both trees. In this tree, suppose that there is an unfixed edge between x and y (and x is closer to the root). Then, the only way to make sure that all unfixed edges are moved is to move it from "children", that is, to move the edge between x and y, we must choose y as the leaf.

Now, we know which edges are moved by operations (and each edge is moved exactly once). Also, for each edge, we know from which vertex the edge is moved. (If the parent of x is y in A and z in B, when we perform an operation on x, we must move an edge from $x - y$ to $x - z$.) The only remaining thing we have to decide is the order of operations.

An order of operations is valid if whenever we perform an operation on v, v is a leaf. This condition can be restated as constraints of the form "we must perform an operation on this vertex before on this vertex". If there is no cyclic dependencies among such constraints, a valid order exists.

Thus, for a fixed r, we can get the answer in $O(N)$ time.

The following picture 3 shows the first case of sample 2.

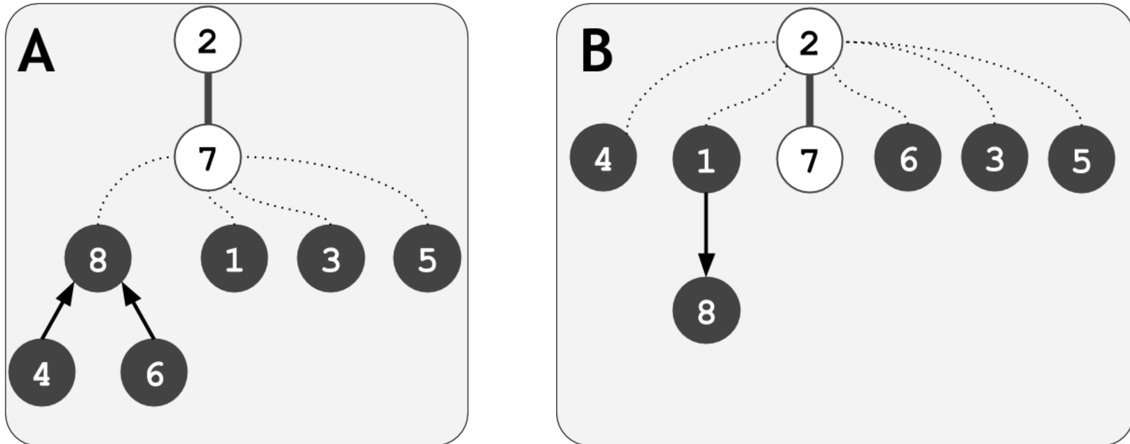

図 3　Sample 2, first case

As mentioned above, in some cases the answer is N. The picture 4 shows such an example.

To handle it, let's try all possible valid moves for the first move - there are only $O(N^2)$ moves. If the vertex v is painted black in the first move, we never perform operations on this vertex again in the future. Thus, we can regard this v as the root in the case above, and we can solve it in $O(N)$.

In total, this solution works in $O(N^3)$ time per testcase.

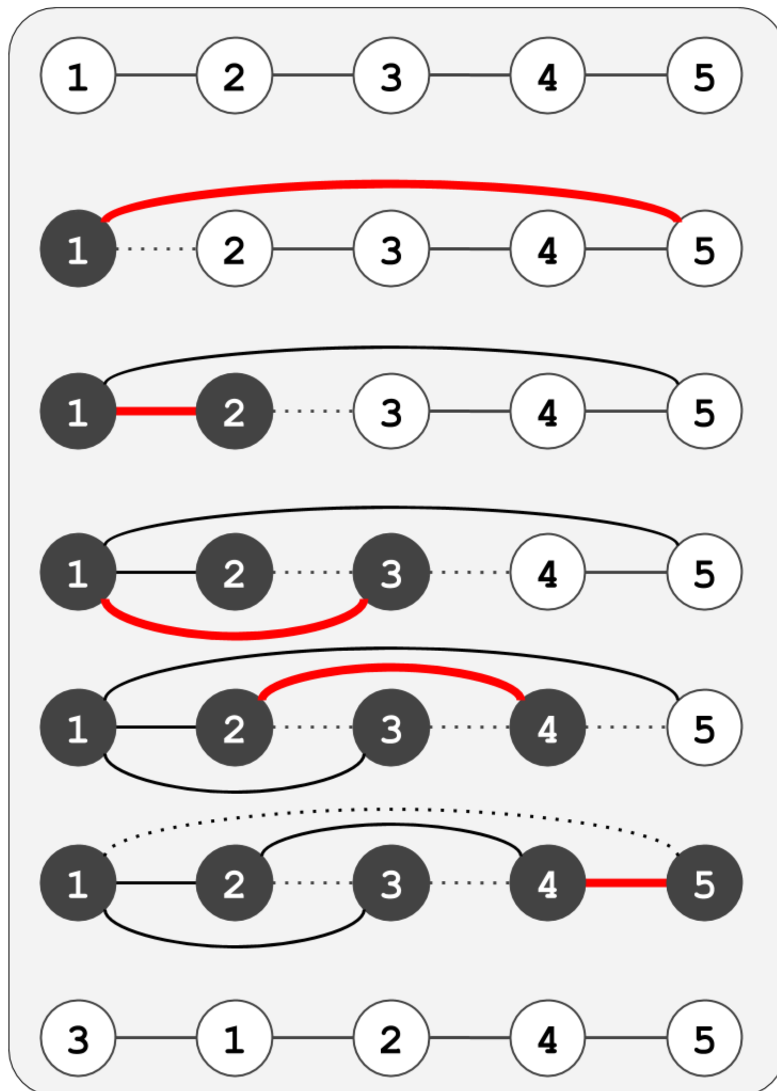

図 4　The answer is N in this case