AGC 25 解説

yutaka1999

2018 年 6 月 3 日

For International Readers: English editorial starts at page 7.

A : Digits Sum

求める答えは、N が 10 のべき乗でないとき N の各位の和、N が 10 べきの場合は 10 となります。

N の各位の和が下限になるのは明らかです。（筆算を考えましょう。）ただし、N が 10 べきの場合は明らかに構成不可能なので、10 が答えになります。これも、筆算から明らかです。

B : RGB Coloring

G で塗られたブロックを、R と B で同時に塗られたブロック、と考えることにします。こうすることで、3 色で塗るのではなく、各ブロックを 2 色で塗る、ただし、同じマスを 2 回塗ってもよいという問題になります。

よって、この問題は以下のような問題になります。

> $A \times a + B \times b = K$ となる $0 \leq a, b \leq N$ に対する ${}_NC_a \times {}_NC_b$ の和を求めよ。

これは a を全探索することで全体として $O(N)$ の時間計算量で解くことができます。よって、この問題を解くことができました。

C : Interval Game

高橋君の最適な移動の仕方は、すでに含まれている場合は動かない、そうでない場合は、与えられた区間の近い方の端点まで動く、というものです。これは、区間の近い方の端点まで移動してからまた動くとしたら、そのステップが終わったあとにその分を動けばよいからです。

さて、高橋君の最適戦略は分かったので、青木君の最適戦略を考えましょう。ここで、青木君はすべての区間を使う必要がないとしてもよいです。つまり、何個かの区間を余して途中でやめたとしても、答えが大きくなることはありません。これは簡単に証明できます。

この明らかなことが何に使えるかということですが、実は青木君は不必要な区間を考えないことで、高橋君の動きを単純化することができます。まず、高橋君が 2 回連続で同じ方向、つまり、連続で右に移動したり連続で左に移動する場合は無視できます。これは、先の高橋君の最適戦略を考えれば明らかです。よって、青木君は不必要な区間を無視することで、高橋君が移動する向きを毎回変えるように動くようにすることができます。このことを考えると、青木君は高橋君を右に動かすための区間と左に動かすための区間の 2 つの役割に、各区間を分けて考えることができるようになります。

つまり、各区間を、右向きに L_i まで動かす、または、左向きに R_i まで動かすためのものとして考えることができます。そして、移動距離への影響はどうなるかというと、右向きに使うなら $2L_i$、左向きに使うなら $-2R_i$ かかります。よって、その影響が大きいものから順番に貪欲に使っていけばよさそうです。ただし、右向きに使うものと左向きに使うものの個数の差は 1 以内に収まらなければならないので、その条件を満たすように貪欲に取らなければいけません。

ここで気になるのは、同じ区間を両方の向きで使ってしまわないかということですが、各区間について $2L_i - 2R_i < 0$ なので、同じ区間を両方の向きで使うことが最適になることはありません。

以上をまとめると、右向きに使う個数と左向きに使う個数を決め打ったときに、どの区間を用いるかは貪欲に使っても問題ないということです。よって、後は適当にソートなどをすることで、$O(N \log N)$ の計算量で解くことができます。

D : Choosing Points

$4N^2$ 個の点から N^2 個の点を選ぶ、しかも制約が 2 個、という時点で勘がいい人は気づくかもしれませんが、これは 2 つの二部グラフが与えられたときに、どちらでも独立集合となる $\frac{V}{4}$ サイズの頂点集合を求める問題です。

ということで本質は格子点の集合が与えられたときに、距離 $\sqrt{D}$ である 2 点間に辺を張ってできるグラフが 2 部グラフとなることです。これを D の帰納法により示していきましょう。

D が奇数のときは簡単です。2 点間の距離が $\sqrt{D}$ になるとき、x 座標と y 座標の差をそれぞれ s, t とすると、$s^2 + t^2 = D$ が奇数より、s, t の偶奇が異なることになります。つまり、各点 (x, y) を $x + y$ の偶奇で 2 色に塗ることを考えると、それらが今考えてるグラフの 2 彩色になります。よって、D が奇数のときはできました。

次に、$D \equiv 2 \pmod 4$ のときを考えます。このときは、x 座標と y 座標の差をそれぞれ s, t とすると、$s^2 + t^2 \equiv 2 \pmod 4$ より、s, t がともに奇数となります。よって、各点 (x, y) を x の偶奇で 2 色に塗ることを考えると、それらが今考えてるグラフの 2 彩色になります。よって、$D \equiv 2 \pmod 4$ のときもできました。

最後に、$D \equiv 0 \pmod 4$ のときを考えましょう。同様にすると、s, t はともに偶数になります。よって、各点 (x, y) を $(x \bmod 2, y \bmod 2)$ で振り分けて、それぞれを適当に $\frac{1}{2}$ 倍に収縮すると、$\frac{D}{4}$ の場合に帰着することができ、帰納的に 2 部グラフになることが示せます。

以上より、距離 $\sqrt{D}$ である 2 点間に辺を張ってできるグラフが 2 部グラフであると示せたので、あとは最初に述べた問題を解けばよいだけです。

まず、各点についてそれぞれの距離で 2 彩色のどちらに属するかを求めます。それによって、$2 \times 2 = 4$ パターンの塗り分けが生じますが、そのうち一番点が多いパターンを選ぶと、そこには $\frac{4N^2}{4} = N^2$ 個以上の点が属しているので、同じパターンだけから N^2 個の点を選ぶことができます。すると、それらの間はどれも距離 $\sqrt{D_1}, \sqrt{D_2}$ にならないので、条件を満たします（これはグラフの作りかた、および、彩色の仕方から明らかです）。よって、この問題を解くことができました。

計算量はグラフの構成が一番重いですが、粗い評価でも $O(N^3)$ に収まります。実際は距離が $\sqrt{D_1}, \sqrt{D_2}$ になるような (s, t) の候補が少ないので、十分高速に動きます。

E : Walking on a Tree

まず、答えの上界を考えてみましょう。木の各辺 i に対して、その辺を通る散歩の個数 c_i を考えます。このとき、明らかに $\sum \min(c_i, 2)$ は上界となります。というのも、各辺につき向きは 2 通りしかないので、その辺を通ることで得られる楽しさは高々 2 であるからです。

では実際にこの上界を達成できることを示しましょう。これは帰納法によって示すことができます。まず、$n = 1$ のときはそもそも辺がないので明らかです。よって、n がより小さい場合に帰着できればよいです。

まず、適当な葉を v とし、v に隣接する辺 e を考えます。$c_e = 0$ の場合は v を無視してしまえばよいから明らかですし、$c_e = 1$ の場合も、v を端点にもつ散歩の端点を、v から e のもう一方の端点に変えればよいので、明らかに帰着できます。なので、$c_v \geq 2$ の場合を考えるとしてよいです。このとき、v を端点に持つ散歩を 2 つ取れます。それらの v でない方の端点を a, b としましょう。このとき、これらの散歩の向きを、$a \to v \to b$ もしくは $b \to v \to a$ と定めることで、これらの散歩の共通部分は両向きで通ったことにしたうえで a から b への散歩を追加する、という状況と同一視できます。よって、v を通る残りの散歩も、端点を v から e のもう一方の端点とみなすことで、$n - 1$ サイズの場合に帰着することができます。よって、帰納的に上界を達成できることが示せました。

実際の構成は上の方法を素直に実装すれば各ステップ $O(N + M)$ で可能です。なので、全体としては $O(N(N + M))$ の計算量で解くことができます。

F : Addition and Andition

X, Y の 2 進表記に対応する文字列 S, T を、説明の都合上以下のような配列として定義しなおします。

- X の i bit 目が 1 ならば $S_i = 1$、0 ならば $S_i = 0$ とする。
- Y の i bit 目が 1 ならば $T_i = 1$、0 ならば $T_i = 0$ とする。

ここで、問題文に書かれている操作を配列に対する操作に言い換えると、以下のようになります。

- $S_i = T_i = 1$ となる i すべてに対して $S_i = T_i = 0$ とする。
- 次に、先の i すべてに対して S_{i+1} と T_{i+1} に 1 を足す。
- その後、S_i, T_i の繰り上がりを処理していく。

ここで、「繰り上がりを処理」する方法はいくつかありますが、以下のようにして行うことを考えてみましょう。

- i を大きい方から小さい方へ順に見ていって、$S_i = 2$ となる i を見つけたら以下を繰り返す。
  - j を $i, i+1, i+2, ...$ とインクリメントしていく。
  - $S_j = 2$ なら $S_j = 0$ にして S_{j+1} に 1 を足す。
  - そうでないなら j に対する操作をやめて、i のネストに戻る。

この方法で繰り上がりを計算することを考えると、問題文の操作は次のように言い換えられます。

- i を大きい方から小さい方へ順に見ていって、$S_i = T_i = 1$ となる i を見つけたら $S_i = T_i = 2$ として以下を繰り返す。…(X)
  - j を $i, i+1, i+2, ...$ とインクリメントしていく。
  - $S_j = 2$ なら $S_j = 0$ にして S_{j+1} に 1 を足す。
  - $T_j = 2$ なら $T_j = 0$ にして T_{j+1} に 1 を足す。
  - どちらでもないなら j に対する操作をやめて、i のネストに戻る。

この操作を K 回繰り返した後の S, T の状態を求めるのがこの問題です。これを愚直に表すと以下のようになります。

- $t = 0, 1, ..., K-1$ に対して、以下を行う。
  - i を大きい方から小さい方へ順に見ていって...

そこで一つの仮説が立ちます。つまり、このループの順番を逆にしても得られる最終状態は同じではないかということです。ここで、ループの順番を逆にするとは、以下のようにして操作を行うことを指しています、

- i を大きい方から小さい方へ順に見ていって、以下の操作を行う。
- ただし、残り何回操作できるかを表すカウンター z 持っておく。最初は $z = K$ である。
  - j を $i, i+1, i+2, ...$ とインクリメントしていく。
  - $S_j = T_j = 1$ かつカウンター z が正なら、$S_j = T_j = 0$ として S_{j+1}, T_{j+1} に 1 を足し、z から 1 を引く。ただし、$z < 0$ ならばその操作を行わずに終了する。
  - $S_j = 2$ なら $S_j = 0$ にして S_{j+1} に 1 を足す。
  - $T_j = 2$ なら $T_j = 0$ にして T_{j+1} に 1 を足す。
  - いずれでもないなら j に対する操作をやめて、i のネストに戻る。

なぜこのように、ループの順番を変えてもいいのかを示しましょう。注目すべきなのは、操作 (X) において、操作後は $S_i = T_i = 0$ となることです。これを使うと、今注目してる i より小さなところで操作 (X) を行ったとしても、増える値の合計は高々 $2^i - 1$ であるから、i bit 目より上の bit には影響を与えないことが分かります。この事実を用いると、$S_i = T_i = 1$ なる最大の i を考えたときに、その i に対して (X) を行った際、i より上の位 j で $S_j = T_j = 1$ となって終わったとすると、i より小さいところでの操作をする前に先に j に対する操作を行ってしまっても、結果が変わらないことが分かります。よってこれを繰り返し用いることで、ループの順番を変える、つまり、先にあげたような操作をした際の最終状態を求めればよいと分かります。

後はループの順番を変えた後の操作について解けばいいだけです。このパートも簡単ではありませんが、先程よりは説明が簡明にできると思うので、分かりやすいと思います。まず、先程の操作を言い換えてみましょう。

- i を大きい方から小さい方へ順に見ていって、$S_i = T_i = 1$ ならば $S_i = T_i = 0$ として以下の操作を行う。
- ただし、残り何回操作できるかを表すカウンター z および下の桁から繰り上がりでくる桁を表す文字列 x を持っておく。
- 最初は $z = K, x = 11$ である。ここで、$x = 10$ ならば S のみ、$x = 01$ ならば T のみ、$x = 11$ ならば両方繰り上がりがあることを指す。
  - j を $i+1, i+2, ...$ とインクリメントしていく。
  - $(S_j, T_j, x) = (0, 0, 10)$ ならば $(S_j, T_j) = (1, 0)$ にして操作を終える。
  - $(S_j, T_j, x) = (0, 0, 01)$ ならば $(S_j, T_j) = (0, 1)$ にして操作を終える。
  - $(S_j, T_j, x) = (0, 0, 11)$ ならば $(S_j, T_j, x) = (0, 0, 11), z \mapsto z - 1$ とする。ただし、$z < 0$ となるならば $(S_j, T_j) = (1, 1)$ として操作を終える。
  - $(S_j, T_j, x) = (1, 0, 11)$ ならば $(S_j, T_j, x) = (0, 1, 10)$ にする。
  - $(S_j, T_j, x) = (1, 0, 10)$ ならば $(S_j, T_j, x) = (0, 0, 10)$ にする。
  - $(S_j, T_j, x) = (1, 0, 01)$ ならば $(S_j, T_j, x) = (0, 0, 11), z \mapsto z - 1$ とする。ただし、$z < 0$ となるならば $(S_j, T_j) = (1, 1)$ として操作を終える。
  - $(S_j, T_j) = (0, 1)$ のときも同様。

ただし、$(S_j, T_j) = (1, 1)$ となるところまで操作でたどりつかないことに注意してください。これは先のループの順序を入れ替えてよいということと同様にして言えます。

このように操作を整理してしまうと、$(S_j, T_j) = (0, 0)$ なる場所はほとんど機械的に処理できると分かります。よって、$(S_j, T_j) = (1, 0), (0, 1)$ となっている j だけを保持しておけばよさそうです。

具体的には、$(S_j, T_j) = (1, 0), (0, 1)$ となっている j を、 j が大きいほど深くにあるように、stack に積みます。この stack を A としましょう。すると操作はどうなるかというと、$S_i = T_i = 1$ なる i を見つけたときに、A の一番上を取り出して、そこまでの $(0, 0)$ のケースを処理します。そして、実際に一番上の要素まで操作でたどりつくならば、そのケースを処理して、次の要素に行きます。ただし、操作後に $(S_j, T_j) = (0, 0)$ とならない場合はその情報を stack に最後に追加しないといけないので、適当な配列に覚えておきます。これを繰り返して、先のシミュレーションを終わらせた後に、途中で生じたあまりを順に stack に積みます。これでシミュレーションが完全に実行できます。

このシミュレーションの計算量を解析しましょう。操作のアルゴリズムをよく見ると、操作後に $(S_j, T_j) = (0, 0)$ とならない場合は $x = 11$ である場合のみで、特に $x = 11$ の後は必ず $x = 10, 01$ のいずれかになります。$x = 10, 01$ の場合は A の先頭の要素数が減り、$x = 11$ の場合も増えはしないから、シミュレーションの 2 回に 1 回に A の要素数が減ることになります。また、A の要素数が増えるのは、各 i に対して $(S_j, T_j) = (0, 0)$ にぶつかって終了するときのみであるから、全体として N 回しか要素数が増えません。よって、シミュレーションの 2 回に 1 回 A の要素数がへり、A に追加される要素数の合計は N であるから、stack の要素を見る回数は高々 $2N$ 回です。よって、stack の出し入れは $O(1)$ でできることから、全体で $O(N)$ の計算量で解くことができます。ただし、「A の要素数」に途中で生じたあまりの個数も含んでいることに注意してください。

A : Digits Sum

We can compute the answer by brue force, but here's more interesting solution: the answer is 10 if N is a power of 10, otherwise the answer is the sum of digits of N.

It's clear that the answer can't be smaller than the sum of digits of N: consider computing the sum on paper. The sum of the sum of digits decreases whenever we get a carry. However, when N is a power of 10, obviously the answer can't be 1, so the answer will be 10 (it must be the same in modulo 9).

B : RGB Coloring

Let's assume that a layer painted by green is a layer painted by both red and blue (and ignore green). Then, in this problem, we paint layers by two colors (but we may paint the same layer with two colors).

The problem will be the following:

> Find the sum of $\binom{N}{a} \times \binom{N}{b}$ for all $0 \leq a, b \leq N$ such that $A \times a + B \times b = K$.

Here, a is the number of layers painted by red, and b is the number of layers painted by blue.

By trying all possible values for a, we can compute the value above in $O(N)$ time.

C : Interval Game

Takahashi's optimal strategy is as follows. If he is already contained in the new interval, he shouldn't move at all. Otherwise, he should move to the nearest endpoint of the new interval. It doesn't make sense to move more than that - instead, he can "postpone" the extra movement and the total distance he moves won't be greater.

What is Aoki's optimal strategy? Since we know Takahashi's optimal strategy, we can regard the game as Aoki's single-player game. Whenever Aoki puts an interval, Takahashi moves in a deterministic way (as we mentioned above), and Aoki wants to maximize the total distance Takahashi moves. Let's modify the rule of the game a bit: Aoki doesn't necessarily have to put all intervals (and after he stops putting intervals, Takahashi returns to the origin). Anyway, since ignoring some intervals never make Takahashi's distance greater, the answer won't change.

Now, since we can omit unnecessary intervals, we can assume that we never put an interval that contains Takahashi. Also, we never move Takahashi to the same direction twice in a row (in this case we can omit the first interval). Thus, we need to consider only two types of movements by Aoki:

- 1. When Takahashi is to the left of L_i, move him to L_i by choosing the interval i.
- 2. When Takahashi is to the right of R_i, move him to R_i by choosing the interval i.

Also, note that these two types of movements must be performed alternately.

To simplify things, instead of forcing Takahashi to start/end the trip at origin, let's add an interval $[0, 0]$ to the set of intervals. How can we compute the total distance if we know the set of intervals used by Aoki? If k intervals with indices $i_1, \ldots, i_k$ are used for the first type of movements, and k intervals with indices $j_1, \ldots, j_k$ are used for the second type of movements (since those two types must be performed alternately, the numbers of intervals must be the same), the total distance will be:

$2(L_{i_1} + \cdots + L_{i_k} - R_{j_1} - \cdots - R_{j_k})$

Thus, for a fixed k, we can just choose intervals greedily, and by trying all possible values of k we can solve the problem in $O(N \log N)$ time.

Note that we will never choose the same interval twice (in two types of operations) in the optimal solution because for each interval $2L_i - 2R_i \leq 0$ holds.

D : Choosing Points

It turns out that in this problem, you are given two bipartite graphs with V vertices, and you are asked to choose $\frac{V}{4}$ vertices that is an independent set on both graphs.

First, let's prove that the following graph is a bipartite graph: the set of vertices is the set of integer points on a plane, and there is an edge between two integer points if the distance between them is $\sqrt{D}$. Let's do an induction on D.

In case D is odd, this is easy. If there is an edge between (x, y) and $(x+s, y+t)$, since $s^2+t^2 = D$ is odd, the parities of s and t are different. Thus, we can color a point (x, y) based on the parity of $x + y$ (standard chessboard coloring).

Suppose that D is even. If two cells p, q are at the distance of $\sqrt{D}$, they must be painted in the same color in a chessboard (assume that they are black cells). Now consider only black cells in a chessboard: these cells form a new square grid that is $\sqrt{2}$ times larger than the original grid (and rotated by 45 degrees). If we change the unit distance by a factor of $\sqrt{2}$, the (original) distance of $\sqrt{D}$ corresponds to the distance of $\sqrt{D/2}$ in the new grid. Thus, we can reduce to the case with $D/2$, and by the induction, we prove that the graph is bipartite.

Therefore, we now see that the graphs are biartite, and we now want to solve the problem mentioned at the beginning.

First, let's color the cells black and white, such that it becomes a bipartite coloring for the distance $\sqrt{D_1}$. Similarly, let's color the cells red and blue for the distance $\sqrt{D_2}$. Each cell is painted in two colors, and there are $2 \times 2 = 4$ possible patterns for a pair of colors in a cell. Thus, there is a pattern (a pair of colors) such that there are $\frac{4N^2}{4} = N^2$ or more cells that are painted in this pattern. From the definition of the colors, it's clear that these points satisfy the conditions.

The slowest part is the construction of graphs. It is obviously $O(N^3)$, and a detailed analysis will show that it will work much faster (because there are not so many pairs (s, t) such that $s^2 + t^2 = D_1 or D_2$).

E : Walking on a Tree

First, let's get the upper bound of the happiness. For each tree edge i, let c_i be the number of walks that pass through this edge. Since there are only two directions for this edge, the number of happiness we can get from this edge is at most $\min(c_i, 2)$. Thus, the upper bound of the total happiness is $\sum \min(c_i, 2)$.

We'll prove that we can always achieve this upper bound, in a constructive way. We do an induction on n, the number of vertices. In case $n = 1$, there's no edge and this is trivial. Suppose that $n > 1$, and let's reduce to a case with smaller n.

Take an arbitrary leaf, and call it v. Let e be the only edge incident to v. In case $c_e = 0$, we can simply remove v from the tree. In case $c_e = 1$, we can also remove v from the tree. Here, the walk that involves v will be converted as follows: if this walk is between v and x, it will be converted to a walk between w and x, where w is an endpoint of e to the opposite of v. Thus, we assume that $c_v \geq 2$.

Take two walks that involve v. Suppose that they are (v, a) and (v, b). Let (v, c) be the intersection of these two paths. Now, we add a restriction: if we choose $a \to v$ we must choose $v \to b$, and if we choose $v \to a$ we must choose $b \to v$. With this restriction, we can ensure that the edges between v and c are covered in both directions, and in the future we can ignore those edges. Thus, $a \to v$ and $v \to b$ is equivalent to $a \to b$, and $v \to a$ and $b \to v$ is equivalent to $b \to a$. Now we can remove two walks (v, a) and (v, b) and add (a, b) instead. By repeating this process while $c_v \geq 2$, we will eventually get a case with $c_v < 2$. Therefore, by induction, we can always achieve the upper bound.

The proof above shows a construction. Each step can be done in $O(N + M)$, and this solution works in $O(N(N + M))$ time in total.

F : Addition and Andition

Define two strings S, T as follows:

- If the i-th bit in the binary representation of X is 1, $S_i = 1$, otherwise $S_i = 0$.
- Similarly define T for Y.

In each operation, we perform the following for the strings:

- For each i such that $S_i = T_i = 1$, let $S_i = T_i = 0$.
- Then, for each such i, add 1 to both S_{i+1} and T_{i+1}.
- Then, handle "carries" of S_i, T_i.

Let's handle "carries" as follows:

- In the decreasing order of i, when we find i such that $S_i = 2$, repeat the following:
  - For $j = i, i+1, i+2, \ldots$ in this order, do the following.
  - If $S_j = 2$, let $S_j = 0$and add 1 to S_{j+1}.
  - Otherwise break the loop (and proceed to the next i).

Now we can restate the operations in the statement as follows:

- In the decreasing order of i, when we find i such that $S_i = T_i = 1$, let $S_i = T_i = 2$ and repeat the following$\cdots(X)$
  - For $j = i, i+1, i+2, \ldots$ in this order, do the following.
  - If $S_j = 2$, let $S_j = 0$and add 1 to S_{j+1}.
  - If $T_j = 2$, let $T_j = 0$and add 1 to T_{j+1}.
  - If neither of above holds, break the loop (and proceed to the next i).

In this task, we are asked to compute the states of S, T after we repeat the operation above K times. Thus, we do the following:

- For each $t = 0, 1, ..., K-1$, do the following.
  - In the decreasing order of i, ...

Here we have a hypothesis: the order of the two loops (a loop for t and a loop for i) doesn't matter. If we change the order of the loops, we get the following:

- In the decreasing order of i, do the following.
- Let $z = K$ be a counter that represents the number of remaining operations we may perform.
  - For $j = i, i+1, i+2, \ldots$ in this order, do the following.
  - If $S_j = T_j = 1$ and z is positive, let $S_j = T_j = 0$, add 1 to both S_{j+1} and T_{j+1}, and decrement z by one. (However, if $z \leq 0$, break the loop.)
  - If $S_j = 2$, let $S_j = 0$and add 1 to S_{j+1}.
  - If $T_j = 2$, let $T_j = 0$and add 1 to T_{j+1}.
  - If neither of above holds, break the loop (and proceed to the next i).

Why can we swap the order of the loops like this? Notice that after the operation (X), $S_i = T_i = 0$ will always be satisfied. Therefore, even if we perform (X) for positions before current i, since the total increase of the value is at most $2^i - 1$, it doesn't affect the $i+1$-th bit or more significant bits. Using this fact, we get the following observation. Consider the maximum i such that $S_i = T_i = 1$. Suppose that when we perform (X) for position i, the operation ends at position j (with $S_j = T_j = 1$). Then, even if we perform an operation for j before performing operations before position i, the result doesn't change. By repeating this observation, we can prove that the order of the loops doesn't matter.

Now, it is sufficient to simulate the process quickly after the loops are swapped. Let's restate the problem as follows:

- In the decreasing order of i, when we find i such that $S_i = T_i = 1$, let $S_i = T_i = 0$ and do the following:
- We keep a counter z (the number of remaining operations we may perform) and a string x.
- Initially, $z = K, x = 11$. The length of x is always 2, and it "carries" from lower digits. If $x = 10$ we have a carry only from S, if $x = 01$ we have a carry only from T, and if $x = 11$ we have carries from both strings.
  - For $j = i, i+1, i+2, \ldots$ in this order, do the following.
  - If $(S_j, T_j, x) = (0, 0, 10)$, let $(S_j, T_j) = (1, 0)$ and finish the operation.
  - If $(S_j, T_j, x) = (0, 0, 01)$, let $(S_j, T_j) = (0, 1)$ and finish the operation.
  - If $(S_j, T_j, x) = (0, 0, 11)$, let $(S_j, T_j, x) = (0, 0, 11), z \mapsto z - 1$. However, in case $z < 0$, let $(S_j, T_j) = (1, 1)$ and finish the operation.
  - If $(S_j, T_j, x) = (1, 0, 11)$, let $(S_j, T_j, x) = (0, 1, 10)$.
  - If $(S_j, T_j, x) = (1, 0, 10)$, let $(S_j, T_j, x) = (0, 0, 10)$.
  - If $(S_j, T_j, x) = (1, 0, 01)$, let $(S_j, T_j, x) = (0, 0, 11), z \mapsto z - 1$. However, in case $z < 0$, let $(S_j, T_j) = (1, 1)$ and finish the operation.
  - The case with $(S_j, T_j) = (0, 1)$ is similar.

Note that the case $(S_j, T_j) = (1, 1)$ never occurs (we can prove this in the same way as the proof for swapping the order of loops). Since we can easily handle position j when $(S_j, T_j) = (0, 0)$, let's keep all j such that $(S_j, T_j) = (1, 0), (0, 1)$. In particular, let's hold all such j in a stack (call the stack A). The elements in A are sorted in the increasing order from top to bottom. Then, when we find i such that $S_i = T_i = 1$, pop the first element of A and handle all $(0, 0)$ cases till the element. In care we can really reach the first element, handle it, and go to the next element. However, when we don't finish the operation with $(S_j, T_j) = (0, 0)$, we keep this information in some array (these values should be added to the stack later). We repeat this, and after the simulation mentioned above finishes, we push all residues we get during the simulation. This way we can perform the simulation.

Let's analyze the time complexity of this simulation. If we look at the algorithm above carefully, the only case where we don't get $(S_j, T_j) = (0, 0)$ after the operation is $x = 11$. After we get $x = 11$, x will be 10 or 01. In case $x = 10, 01$, the number of elements of A decreases, and in case $x = 11$, the number of elements of A doesn't increase. Thus, the number of elements of A decrease in every two steps of the simulation. Also, since the number of elements of A increases only when $(S_j, T_j) = (0, 0)$ (and this happens at most once for each i), this happens at most N times. Thus, we perform operations for the stack at most $2N$ times. Since we can perform push/pop for a stack in $O(1)$, this solution works in $O(N)$ time in total. Note that "the number of elements in A" includes the number of new elements added during the simulation.