AtCoder Grand Contest 007 Editorial

writer : dreamoon

English editorial starts on page 8.

A : Shik and Stone

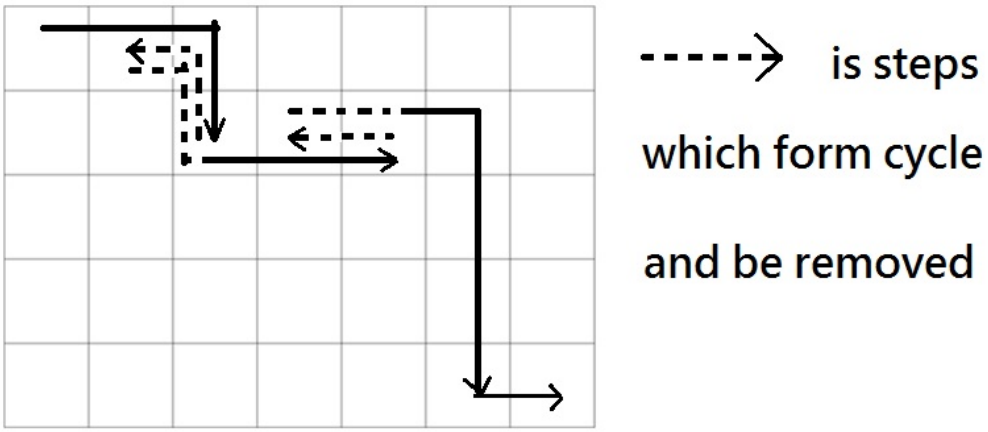
この問題には簡潔な解法が存在します。入力中の文字 `#` の個数を数えて、その個数が $H+W-1$ と等しければ `Possible`、等しくなければ `Impossible` を出力すればよいです。以下、この解法の妥当性を示します。

駒が上下左右に 1 マス動くことを「ステップ」と呼ぶことにします。まず、左上隅のマスから右下隅まで、右または下へのステップのみを使って移動するときのステップ列の集合 (この集合を S_A とします) と、左上隅のマスから右下隅まで最少のステップ数で移動するときのステップ列の集合 (この集合を S_B とします) が等しいことを示します。

i 番目に駒が通ったマスの行番号と列番号の和を g_i とします。i 番目のステップが右または下へのステップなら $g_{i+1} = g_i + 1$ 、左または上へのステップなら $g_{i+1} = g_i - 1$ が成り立ちます。したがって、数列 g_i の隣接する二つの要素の左の絶対値は常に 1 となります。駒が行ったステップの数を n とすると、$g_1 = 2$ 、$g_{n+1} = H + W$ となります。よって、n は少なくとも $H+W-2$ であり、$n = H+W-2$ のときはすべての i について $g_{i+1} = g_i + 1$ となることになります。ゆえに、S_A と S_B はともに、$n = H+W-2$ であるようなステップ列全体の集合となります。

次に、'#' の個数は駒が一度以上通ったマスの個数であるため、$n+1 \geq$ (`#` の個数) が成り立ちます。`#` の個数が $H+W-1$ より大きいと、n が $H+W-2$ より大きくなり、ステップ数が最少でないことになります。したがって、この場合は `Impossible` が答えとなります。

最後に、`#` の個数が $H+W-1$ と等しければ `Possible` を答えとしてよいことを示します。問題の制約より、入力された情報に対応するステップ列が存在し、そのうちの一つを選びます。このステップ列から、通るマスの列がサイクルを形成するようなステップを取り除きます（下図を参照してください)。`#` の個数を減らすことはできない（減らせるようであれば問題の制約に反する）ため、このようなステップを取り除くことは常に可能です。このようなステップを取り除くと、ステップ数が (`#` の個数) -1 であるようなステップ列が得られます。これは最少のステップ数であり、このステップ列は入力で与えられる情報と整合します。



おまけ: この問題から「問題文および a で与えられる情報と整合するような駒の動き方が存在する。」という制約を取り除いた場合、同じ解法が使えるでしょうか？もし使えないならば、どうすればよいでしょうか？

B : Construct Sequences

まず解の一例を示し、それからその解がすべての条件を満たすことを示すことにします。以下が解の一例です。

$r_{p_i} = i\ (1 \leq i \leq n)$ として、
$A_i = 30000 \times i\ (1 \leq i \leq n)$
$B_i = 30000 \times (n - i) + r_i\ (1 \leq i \leq n)$

正当性の証明（概略）は以下の通りです。
$30000 \times 1 \leq A_i \leq 30000 \times n \leq 30000 \times 20000 \Rightarrow 1 \leq A_i \leq 10^9$.
$30000 \times 1 + 1 \leq B_i \leq 30000 \times n + n \leq 30000 \times 20000 + 20000 \Rightarrow 1 \leq B_i \leq 10^9$.
$A_i = 30000 \times i < 30000 \times (i + 1) = A_{i+1}$.
$B_i = 30000 \times (n - i) + r_i > 30000 \times (n - i) = 30000 \times (n - i - 1) + 30000 > 30000 \times (n - i - 1) + r_{i-1} = B_{i-1}$.
$A_{p_i} + B_{p_i} = 30000 \times p_i + 30000 \times (n - p_i) + r_{p_i} = 30000 \times n + i < 30000 \times n + i + 1 = 30000 \times p_{i+1} + 30000 \times (n - p_{i+1}) + r_{p_{i+1}} = A_{i+1} + B_{i+1}$.

C : Pushing Balls

球を一つ転がしたあと、残りの球と穴に番号を振りなおすと、やはり i 番の球が i 番の穴と $i+1$ 番の穴の間にあることがポイントです。球を一つ転がしたとき、球の転がし方として可能なもの

すべてを考慮して、隣接する球と穴の間の距離の期待値を更新します。計算すると、期待値の列 $\mathbf{E}[d_i]$ はやはり等差数列をなし、その長さは 1 短くなっています。

よって、求めたい期待値を再帰的に求めることができます。最初に転がす球が移動する距離の期待値を求め、残った球が移動する距離の期待値を「サイズの減った元の問題」として再帰的に求めた結果と足し合わせればよいです。

D : Shik and Game

以下、ゲームが行われる直線は左右に伸び、右に進むほど座標が大きくなるものとします。プレイヤーがゴールと反対方向に戻ってコインを拾いに行くとき、まだ拾っていないコイン（まだ出現していないものも含む）のうち最も左にあるものの位置まで戻り、もしまだ出現していなければ出現まで待つべきです。でないと、そのコインを取りに再び戻る必要が生じてしまいます。したがって、このゲームにおける戦略は、クマたちを何個かの連続する区間に分割したもので表されます。この分割が表す具体的な行動は、プレイヤーがまず最初の区間にいるクマたち全員にキャンディを与え、この区間の左端のクマの位置まで戻ってすべてのコインを拾い、次の区間に進む、というものです。

戦略における区間の個数を M 、i 番目の区間の左端の座標を L_i 、右端の座標を R_i とおくと、この戦略における所要時間は $E + \sum_{i=1}^{M} \max(t, 2 \times (R_i - L_i))$ となります。左から i 匹のクマからすべてのコインを回収するのに要する最短時間を $dp[i]$ とおくと、これは $dp[i] = \min_j(dp[j] + \max(t, 2 \times (a_i - a_{j+1})))$ とする動的計画法により $\mathrm{O}(N^2)$ 時間で計算できます。

この解法の計算量を改善しましょう。上式の $\max$ の部分に注目し、j が動く範囲を $t > 2 \times (a_i - a_{j+1})$ であるような範囲と $t \leq 2 \times (a_i - a_{j+1})$ であるような範囲に分割します。各 i に対し、しきい値 p が存在して前者の範囲が $j > p$ 、後者の範囲が $j \leq p$ となることがわかります。さらに、p の値が i に関して広義単調増加することもわかり、すべての i に対する p の値を合計 $\mathrm{O}(N)$ 時間で求めることができます。

範囲 $t > 2 \times (a_i - a_{j+1})$ に対しては、動的計画法の式の右辺は $\min_j(dp[j] + t)$ となります。配列 dp の値は広義単調増加するため、この値は範囲内の最も小さい j で最小値をとります。この最小値はすべての i に対し合計 $\mathrm{O}(N)$ 時間で求めることができます。

範囲 $t \leq 2 \times (a_i - a_{j+1})$ に対しては、動的計画法の式の右辺は $\min_j(dp[j] + 2 \times (a_i - a_{j+1}))$ となります。これを $2 \times a_i + \min_j(dp[j] - 2 \times a_{j+1})$ と変形します。動的計画法に用いるもう一つの配列 $dp2$ を用意し $dp2[i] = \min_{j=1}^{i}(dp[i] - 2 \times a_{i+1})$ として、この値をすべての i に対し合計 $\mathrm{O}(N)$ 時間で求めることで、$2 \times a_i + \min_j(dp[j] - 2 \times a_{j+1}) = 2 \times a_i + dp2[j]$ をすべての i に対し合計 $\mathrm{O}(N)$ 時間で計算することができます。

以上より、元の解法の計算量 $\mathrm{O}(N^2)$ を $\mathrm{O}(N)$ に改善することができました。

E : Shik and Travel

直感的に、この問題を解く上で最初に答えの値に対する二分探索を行うべきであると感じた人がいるかもしれません。ここで説明する解法もその方針をとります。すなわち、ある金額 V に対して、「負担金額が V 以下となるような、木の葉に相当する都市（以下、これらを葉都市と呼びます）の訪問順序が存在するか？」という部分問題を解くことを考えます。

まず、次の点を踏まえます。旅行中、i 番の道を一度目に通ったら、二度目にその道を通るまでに、$i+1$ 番の都市を根とする部分木に存在するすべての都市を訪れなければなりません。つまり、ある都市を一度目に訪れたら、その日からその都市を根とする部分木に存在するすべての葉都市で宿泊していくことになります。より正確には、都市 i を根とする部分木に存在する葉都市の個数を n_i とすると、ある数 st_i が存在し、以下の期間には都市 i を根とする部分木の内部にいることになります: 「旅行の st_i 日目の途中から」、「st_i+1 日目から st_i+n_i-1 日目の間ずっと」、「st_i+n_i 日目の途中まで」。そして、その他の期間には部分木の外部にいることになります。

このことから、部分問題への再帰的な愚直解が得られます。各都市で、二つの子が持つ訪問列を再帰的に併合していき、その結果が条件を満たすか判定していきます。以下に疑似コードを記します。

```
function f
argument: root city i of some subtree
return value: a set S_i of ordered pair of numbers

S_i に要素 (a, b) が含まれるとき、葉都市の訪問順序であって以下のようなものが存在することを表す。
- st_i 日目に発生する料金のうち、この部分木の内部で発生する分は a である。
- st_i + n_i 日目に発生する料金のうち、この部分木の内部で発生する分は b である。
- st_i + 1 日目から st_i + n_i - 1 日目に発生する料金はいずれも V 以下である。

f(i) {
    if n_i = 1:
        return {(0, 0)}
    set S_i as empty set
    denote the number of two son cities of city i as j and k
    let S_j = f(j) and S_k = f(k)
    for all combination of element (a,b) in S_j and element (c,d) in S_k:
        if (b+c+v_j+v_k <= V) then add (a+v_j, d+v_k) to S_i
        if (a+d+v_j+v_k <= V) then add (c+v_k, b+v_j) to S_i
    return S_i
}
```

この $f(1)$ を呼び、S_i が空であるかどうかが部分問題の答えとなります。

問題は $|S_i|$ が大きくなりすぎることで、これを小さく抑えたいです。目標は S_1 が空であるかどうかを知ることのみであるため、S_i の要素 (a,b) であって、$a' \leq a$ and $b' \leq b$ を満たすような他の要素 (a',b') が存在するようなものは S_i から省くことができます。

このようにすると、$|S_i| \leq \min(\ |S_j|, |S_k|)\ \times 2$ が成立します。

$|S_j| \leq |S_k|$ とします。各 (a, b) について、それを用いて生成される新たなペアは $(a + v_j, u)$ または $(u, b + v_j)$ (u は任意の数) のいずれかの形式で表されます。このそれぞれに対して、その形式で表されるような最終的に残るペアはたかだか一つです。よって $|S_i| \leq |S_j|$ が示されました。$|S_k| \leq |S_j|$ の場合も同様です。

この性質は、さまざまな点において役立ちます。特に、$\sum_{i=1}^{N} |S_i| = \mathrm{O}(NlogN)$ が成り立ちます。以下にこれを示します。

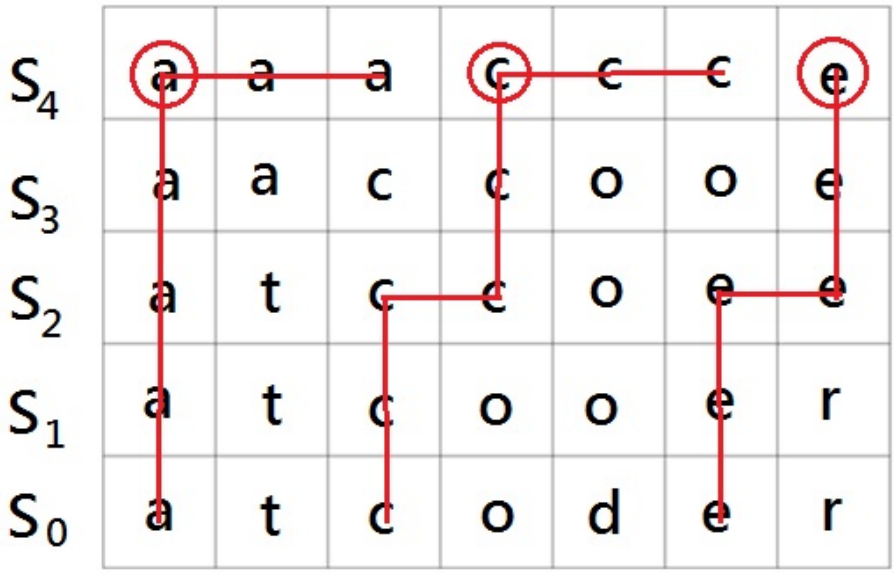
i 番の都市から 1 番の都市までの距離（深さ）を d_i とします。各都市 i について、i を根とする部分木において d_j が最小となるように都市 j を選びます。すると、$|S_i| \leq 2^{d_j - d_i}$ となります。したがって、$|S_i|$ の値を、i を根とする部分木に存在する深さ d_j のすべての都市に 1 ずつ「償却」することができます。この「償却」を行っていく過程において、$|S_i|$ の和がすべての頂点の値の和を超えることはありません。また、各頂点に「償却」される値はたかだか $log_2(N + 1)$ です。よって、$\sum_{i=1}^{N} |S_i| \leq N \times log_2(N + 1)$ となります。

以上から、尺取り法を用いて S_i をソートされた状態に保ち、$(a + v_j, u)$ と $(u, b + v_j)$ のそれぞれの形式における u の最小値を求めていくことにより、部分問題を時間計算量 $\sum_{i=1}^{N} |S_i| = \mathrm{O}(NlogN)$ で解くことができ、元の問題を $O(NlogN \times log\ answer)$ 時間で解くことができます。

F : Shik and Copying String

まず、$S_0 = T$ のとき、答えは 0 です。この場合は例外として扱います。

下図に、操作が行われていき文字列が S_0 から T に変遷する様子を示します。



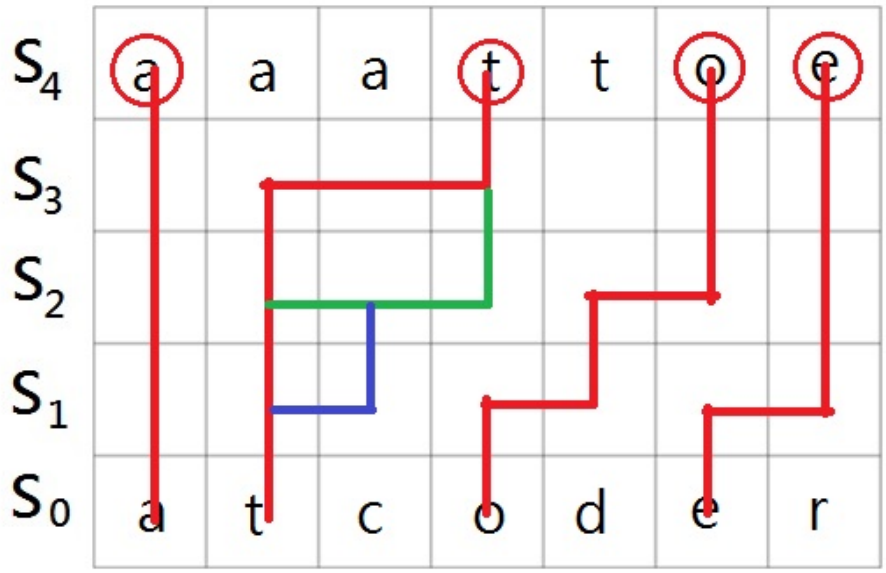
これは、$S_0 = \texttt{atcoder}$、$T = \texttt{aaaccce}$ のときの操作列であって、$S_4 = T$ であるようなものを示しています（最小の操作回数を求めているわけではありません）。同じ文字が連続するとき、それらのうち最初のもののみに注目すればよいことがわかります（図ではそのような文字は丸で囲まれています）。そのような T の丸で囲まれた文字を、操作の途中過程における文字列に含まれる同じ文字と線で結んでいき（この線は他の線と重なってはいけません）、最後に S_0 に含まれる同じ文字に到達することができます。線を引ける方向は左と下のみです。

操作の過程の途中で文字列がどのようなものであるかが不明であっても、T の丸で囲まれた文字と S_0 の文字を結ぶことさえできれば、そこから操作列を構成することができます。よって、元の問題の代わりに、$S_i = T$ の丸で囲まれた文字すべてから S_0 の同じ文字に線を引くことができるような最小の i を求めればよいです。

i を定めたとき、$S_i = T$ であるような線の引き方が存在するかどうかは、以下のような貪欲なアルゴリズムにより判定できます。T の丸で囲まれた文字を後ろから見ていき、それぞれの文字から可能な限り線を下に引いていき、左に引いていかざるを得ないときのみ左に引いていきます。もし途中で線がマス目からはみ出るようなことがあれば、$S_i = T$ であるような線の引き方は存在しません。

この正当性は以下のように示せます。$S_i = T$ であるような線の引き方が存在するとき、上のような貪欲な引き方でも線を引くことができることを示します。

任意の線の引き方に対し、それを貪欲な引き方で得られるようなものに変形していくことが以下のステップを繰り返すことで可能です。各ステップでは、引き方が貪欲なステップで得られるような引き方と最初に相違する箇所を見つけます。そこでは、貪欲な引き方が下に線を引いていっているところで、元の引き方では（我々から見て）左に曲がっているはずです。この左に曲がるのを 1 マス分遅らせ、曲がったら直前の引き方と一致するまで左に線を引き続けます。このように線の引き方を変形していく過程を下図に示します（赤 $\rightarrow$ 緑 $\rightarrow$ 青）。



この貪欲法と二分探索を組み合わせると答えを得られますが、時間計算量が $\mathrm{O}(log(N) \times N^2)$ となってしまいます。

この解法を高速化することを考えます。配列 H を用意し、$H[j]$ に線が j 列目を通ったときの縦方向の位置のうち最も上のものを記録します。はじめ、すべての j に対し $H[j] = 0$ とします。k を N から 1 にイテレートします。各イテレーションでは、答えが少なくとも $H[k]$ であること ($T[k] = T[k+1]$ のとき)、もしくは少なくとも答えが $H[k+1]$ であること ($T[k] \neq T[k+1]$ のとき) がわかります。$T[k]$ が丸で囲まれた文字であるときに H の値を更新します。まず、文字 $S_0[l]$ であって、$T[k]$ と同じ文字であり、$h[l] = 0$ であるようなもののうち最も右にあるものを求めます。すると、l 以上 $k-1$ 以下の r に対し、新しい $H[r]$ の値は (古い $H[r+1]$ の値) $+1$ となります（なぜなら、r 列目の線は $(H[r+1]+1)$ 行 $(r+1)$ 列目のマスから入ってくるからです)。これはマス目の行数を表す i の値によりません。したがって、二分探索が不要となり、解法の時間計算量が $\mathrm{O}(N^2)$ となりますが、まだ不十分です。

そこで、「l 以上 $k-1$ 以下の r に対し、(新しい $H[r]$ の値) $=$ (古い $H[r+1]$ の値) $+1$」という操作に注目します。この操作は、配列 H の $H[l+1]$ から $H[k]$ の区間にあるそれぞれの値を左隣に移して 1 を加える、という操作とみなすことができます（$H[k]$ 以降の値は以後無視されます)。したがって、もう一つの配列 h と二つの値 $\mathrm{offset}_{\mathrm{value}}$, $\mathrm{add}_{\mathrm{value}}$ を管理し、$h[i + \mathrm{offset}_{\mathrm{value}}] + \mathrm{add}_{\mathrm{value}} = H[i]$ を成り立たせることができます。このとき、操作を行う際は単に $\mathrm{offset}_{\mathrm{value}}$ と $\mathrm{add}_{\mathrm{value}}$ に 1 を加えればよいです（実際にはもう少し複雑な処理が必要ですが、省略します)。これにより、解法の時間計算量が $\mathrm{O}(N)$ となります。

AtCoder Grand Contest 007 Editorial

writer : dreamoon

A : Shik and Stone

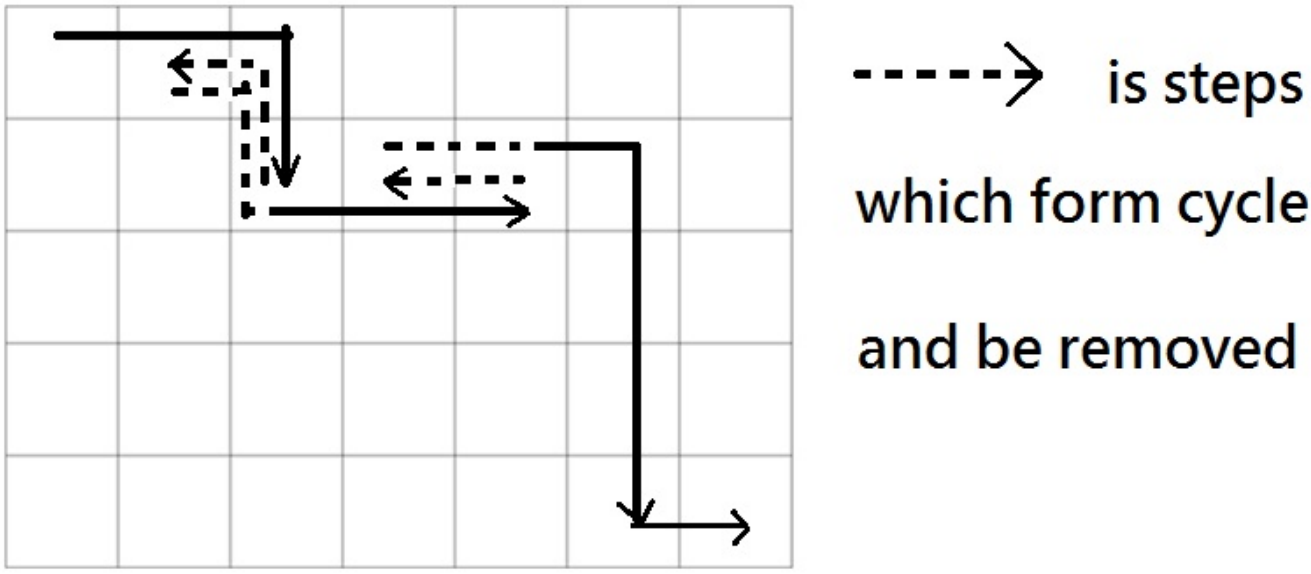
The solution of this problem is quite short. We can get the answer by counting the number of '#'s in the input. If the number is equal to $H+W-1$, the answer is 'Possible'. Otherwise, the answer is 'Impossible'. Now we want to prove this conclusion.

Firstly, the set of steps which only uses right and down moves(denoted as S_A) is equal to the set of steps using minimum moves(denoted as S_B). Let's prove it.

Denote g_i as the sum of row number and column number of the i-th position of stone. If using only right and down moves in i-th step, $g_{i+1} = g_i + 1$. Otherwise, for left and up moves, $g_{i+1} = g_i - 1$. That is, the absolute difference of two consequtive numbers in sequence g_i is always 1. Suppose Shik uses n steps to move the stone. We know that $g_1 = 2$ and $g_{n+1} = H + W$. Therefore, we can know the minimum value of n is $H + W - 2$ and $g_{i+1} = g_i + 1$ for all i must be satisfied when $n = H + W - 2$. Then we get that S_A and S_B are all sequences of moves such that $n = H + W - 2$.

Secondly, because '#' denotes that the stone had ever located at this position, $n + 1 \geq$ the number of '#'s. If the number of '#'s is larger than $H + W - 1$, n will be larger than $H + W - 2$, then Shik doesn't use the minimum moves. So in this condition the answer is always 'Impossible'.

Finally, because there always exists a valid sequence of moves for Shik in inputs. We can choose any of valid sequence first, then remove these steps such that the position sequence of stone form cycles (you can see the picture below for understanding). The removing is always possible becuase it's impossible to decrease the number of '#'s. If it is possible, it will conflict with the fact that the input is valid. After removing all unnecessary cycle steps, we can get a moving sequence which the number of steps added one is equal to the number of '#'s. Now we get a possible moving sequence using minimum steps which match the input.



Bonus problem: If the input dosn't need to be generate by some valid moving sequence. Can we use the same method to determine whether such input is generate by Shik with using only right and down moves in all steps.? If not, how to solve it?

B : Construct Sequences

First we give a possible solution then we justify that it satisfies all of the conditions:

Let $r_{p_i} = i$ for $1 \leq i \leq n$. $A_i = 30000 \times i$ for $1 \leq i \leq n$. $B_i = 30000 \times (n - i) + r_i$ for $1 \leq i \leq n$.

Now the (sketch of the) proof:

$30000 \times 1 \leq A_i \leq 30000 \times n \leq 30000 \times 20000 => 1 \leq A_i \leq 10^9$. $30000 \times 1 + 1 \leq B_i \leq 30000 \times n + n \leq 30000 \times 20000 + 20000 => 1 \leq B_i \leq 10^9$. $A_i = 30000 \times i < 30000 \times (i + 1) = A_{i+1}$. $B_i = 30000 \times (n-i) + r_i > 30000 \times (n-i) = 30000 \times (n-i-1) + 30000 > 30000 \times (n-i-1) + r_{i-1} = B_{i-1}$. $A_{p_i} + B_{p_i} = 30000 \times p_i + 30000 \times (n - p_i) + r_{p_i} = 30000 \times n + i < 30000 \times n + i + 1 = 30000 \times p_{i+1} + 30000 \times (n - p_{i+1}) + r_{p_{i+1}} = A_{i+1} + B_{i+1}$

C : Pushing Balls

Notice that after rolling one ball, if we renumber all the remaining balls and holes, the balls still satisfy that the i-th ball is between the i-th hole and the $(i + 1)$-th hole. Let's also recalculate the expected value of the distance between neighbouring items for all possible rolls of the last ball. We can find that the the expected value $\mathbf{E}[d_i]$ is still an arithmetic sequence with N decreased by 1.

So we can calculate the answer recursively: calculate the expected value of rolling distance of the first ball, and sum it with the remaining similar problem in reduced size.

D : Shik and Game

If the player decided to go back to pick up some coins, he will go back to the very first coin he hasn't picked up or even wait for the first coin to appear, otherwise he needs to go back for that coin again. So, we could partition all the bears into continuous segments to represent a strategy of the game. More specifically, we first give candies to all the bears in a segment, go back to collect all the coins in this segment and then proceed to the next segment.

Assuming we have M segments and the i-th segment starts at position L_i and ends at position R_i, the answer to the problem is $E + \sum_{i=1}^{M} \max(t, 2 \times (R_i - L_i))$. Let $dp[i]$ be the answer to the problem with only first i bears, we have $dp[i] = \min_j(dp[j] + \max(t, 2 \times (a_i - a_{j+1})))$, an $O(N^2)$ dp solution.

To accelerate the solution, we can split the max part in the above equation into two parts: $t > 2 \times (a_i - a_{j+1})$ and $t \leq 2 \times (a_i - a_{j+1})$. It's trivial that for each i we can find a pivot p such that the first part consists of all $j > p$ and the second part holds for all $j \leq p$. Furthermore, it's easy to show that p is non-decreasing for all i, so we can maintain p using amortized $O(N)$ time for all i.

For the first part, the equation is now $\min_j(dp[j]+t)$. $dp[]$ is non-decreasing, thus the smallest possible j will leads to the optimal solution in all j. We can calculate this part greedily using $O(N)$ time for all i.

For the second part, the equation is $\min_j(dp[j] + 2 \times (a_i - a_{j+1}))$. We can rewire it as $2 \times a_i + \min_j(dp[j] - 2 \times a_{j+1})$. By maintaining another dp array $dp2[i] = \min_{j=1}^{i}(dp[i] - 2 \times a_{i+1})$ in $O(N)$ time for all i, we can calculate $2 \times a_i + \min_j(dp[j] - 2 \times a_{j+1}) = 2 \times a_i + dp2[j]$ in $O(N)$ time for all i.

To this point, we have improved the time complexity of the solution from $O(N^2)$ to $O(N)$.

E : Shik and Travel

You may have the intuition that the first step of solving this problem is to apply binary search on answer. It's also the first step of the solution provided here. So, we are going to study the sub problem here: whether there exists an arrangements of leaf cities to stay during the travel such that the answer is not larger than a certain value V.

Firstly, we should know one thing: in the travel, once the employee passed a road i, he/she will visit all cities belong to the descendants in the subtree rooted at city $i + 1$ before passing

the road i again. So for any given city, Shik always stay at all leaf cities in the subtree rooted from this city in consecutive days. More precisely, if the subtree of city i contains n_i leaves, there exist some number st_i such that in the last part of day st_i, the first part of day $st_i + n_i$ and all days from day $st_i + 1$ to day $st_i + n_i - 1$, Shik is travelling in the subtree of city i. Nevertheless, in other days the employee won't be in this subtree.

After knowing that, we give a straightforward recursive algorithm to solve this sub problem. The main idea is to recursively merge possible answers from both children to produce all possible answers of a vertex. Following is the pseudo code:

```
function f
argument: root city i of some subtree
return value: a set S_i contains ordered pairs (a,b),
              each representing there exists one possible
              order of stay sequence of leaf cities such
              that the sum of tolls at the st_i day in
              this subtree is a, the sum of tolls at the
              day t_i + n_i in this subtree is b and sum
              of tolls in any days from day st_i + 1 to day
              st_i + n_i - 1 is not larger than V.

f(i) {
    if vertex i is leaf:
        return (0,0)
    set S_i as empty set
    let two children cities of city i to be j and k
    let S_j = f(j) and S_k = f(k)
    for all combinations of (a,b) in S_j and (c,d) in S_k:
        if(b + c + v_j + v_k ≤ V) add (a + v_j, d + v_k) to S_i
        if(a + d + v_j + v_k ≤ V) add (c + v_k, b + v_j) to S_i
    return S_i
}
```

After calling $f(1)$ and getting S_1, we can know the answer by whether S_1 is empty or not.

But the function may produce quite large $|S_i|$. So we want to reduce it. Because our purpose is only to know whether S_1 is empty or not. We can remove all (a, b) in S_i if there is another pair (a', b') in S_i satisfying $a' \leq a$ and $b' \leq b$.

Now we want to prove after performing such removal, we have inequality: $|S_i| \leq 2\min(|S_j|, |S_k|)$:

Assuming $|S_j| \leq |S_k|$. When a specific pair $(a, b) \in S_j$ matching all pairs in S_k, the generated pairs will be in the form of $(a + v_j, u)$ and $(u, b + v_j)$ (exact value of u depends on the matched pair in S_k). Each aforementioned form will remain at most one pair for a given pair $(a, b) \in S_j$ (the one with minimum u among all possible values of u). So the maximum size of resulting set is at most double of $|S_j|$. The proof for $|S_k| \leq |S_j|$ part is similar.

Such inequality make many things wonderful! The most important thing for us is that

$\sum_{i=1}^{N} |S_i| = O(NlogN)$. Proof is given as the following:

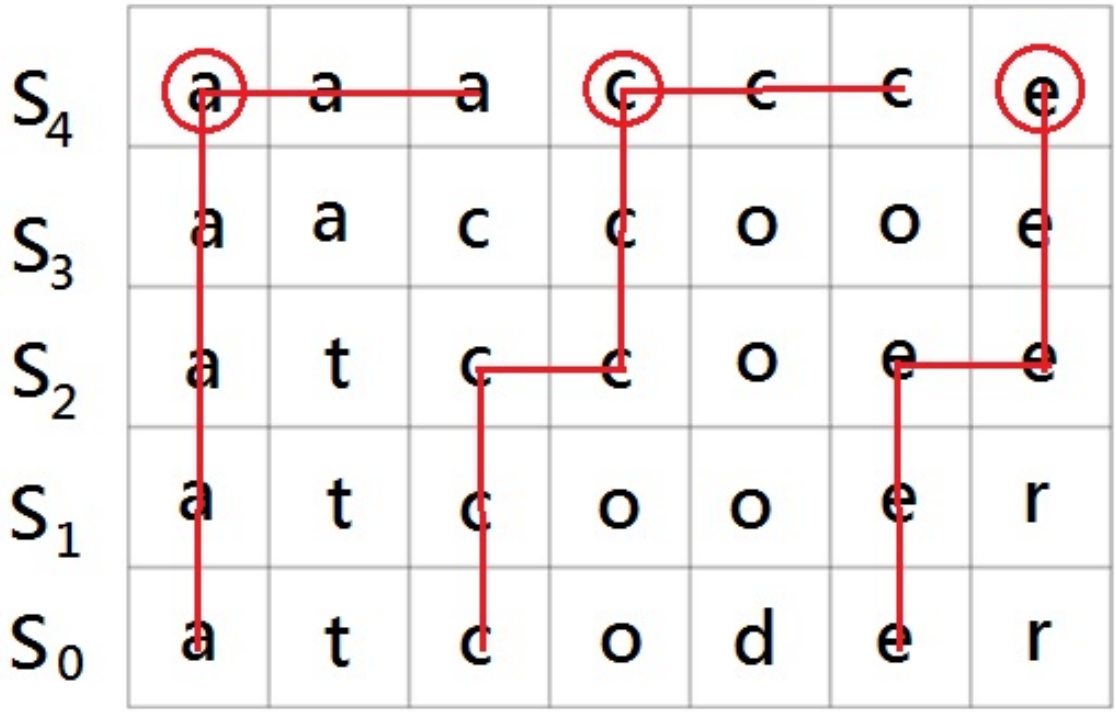
Denote the distance between node i and the root (i.e. depth) as d_i. For each city i, let j be a leaf city in the subtree of city i which has minimum d_j, we have $|S_i| \leq 2^{d_j - d_i}$. So, we can amortize the value of $|S_i|$ to all cities in the subtree of city i with depth exactly equals to d_j by adding value 1 to each of such city. Note that there are exactly $2^{d_j - d_i}$ such cities because d_j is the smallest depth in the subtree. The amortized value in one vertex is at most $log_2(N+1)$; This is because only the parents of a node who has a maximum distance of $\log_2(N+1)$ to the node has the chance to add value to the node, otherwise the size of the tree would be larger than N (since we have a complete subtree of depth greater than $log_2(N+1)$). As a conclusion, we get $\sum_{i=1}^{N} |S_i| \leq N \times log_2(N+1)$ so $\sum_{i=1}^{N} |S_i| = O(NlogN)$.

With the fact we can use two-pointer technique to find the minimum value of u in the two forms $(a+v_j, u)$ and $(u, b+v_j)$ since we can maintain that S to be sorted. The time complexity of the sub problem will be $\sum_{i=1}^{N} |S_i| = O(NlogN)$. Go back to the original problem. The time complexity will be $O(NlogN \times log\ answer)$.

F : Shik and Copying String

Note that we don't consider the case $S_0 = T$. It's answer is 0 obviously.

In the beginning, I'd like to roughly give a picture about how the string copying works and what's the relation between string S_0 and string T. Please see picture below.



In this picture, we suppose $S_0 =$ "atcoder" and $T =$ "aaaccce". It shows one possible process such that $S_4 = T$ (Note that this picture does NOT show the process of the optimize solution for minimum i). We are only concerned about the first letter in and these letters which is different to previous letter of themself (the circled letter in the picture). We can connect these circled letters to their source letters one by one with disjoint lines and finally stop at letters in S_0. You may notice that these line can only go either left or down.

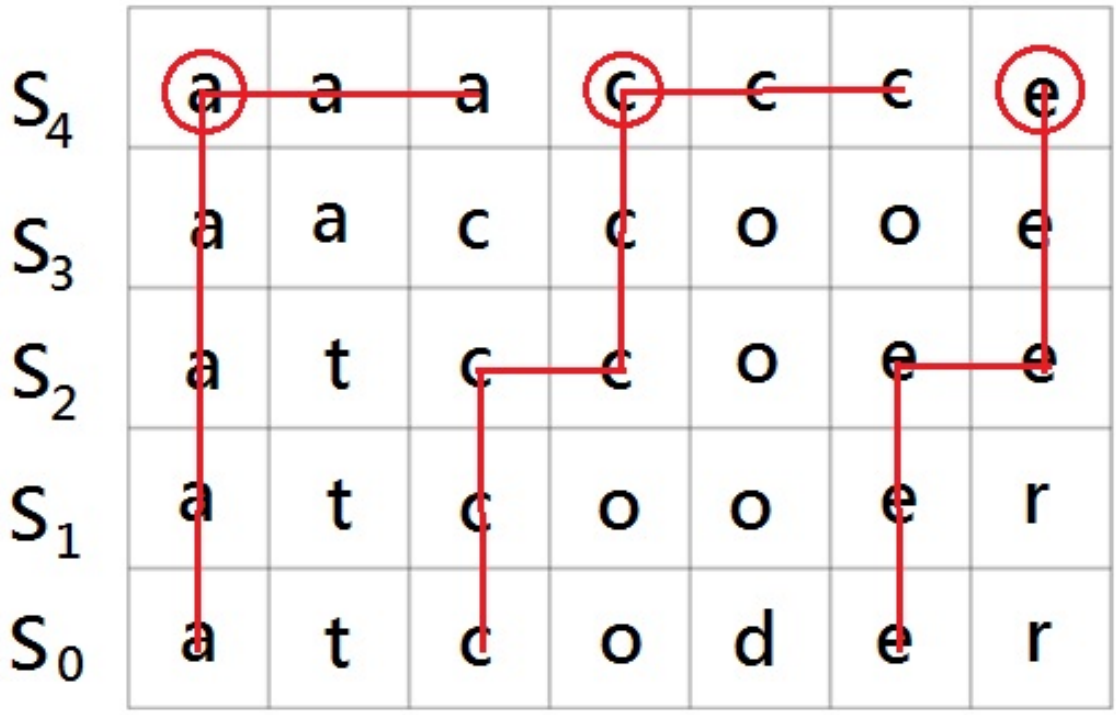
In the other face, suppose we don't know the process between S_0 and T. But if we can draw lines from all circled letters in T to the same letter in S_0, we can construct one possible process easily. So we can transform the original problem of finding minimum i to check if we can draws lines from $S_i = T$ to S_0.

Now we give a greedy algorithm in intuition to test whether exist configures of line for some i with $S_i = T$: for each circled letter from back to front, we draw line to down first if possible, otherwise we draw it to left. If in some step the drawn lines go out of bound, then there exists no valid process such that $S_i = T$.

This solution can be proven easily. What we should prove is, if there exists some valid process for $S_i = T$, then the algorithm in this solution will also construct a valid process.

We can always adjust any valid configuration of lines to match the lines produced by our algorithm step by step. In each step, we find the first time that the lines don't draw by our algorithm and change the direction(always from left to down) of the drew line at this point

for one unit, and continue to draw the line to left until meeting the lines in previous step. The picture below show the steps from red lines to meet our algorithm (red -¿ green -¿ blue).



Using this greedy algorithm we can get a binary search answer + simulation solution for this problems. But the time complexity is $O(log(N) \times N^2)$.

For speeding up this algorithm, we maintain an array $H[j]$ denoting the highest position passed by the line in column j. Initially, $H[j] = 0$ for all j. We iterate k from N to 1. In each iteration, we can know the answer should at least be $H[k]$ (if $T[k] = T[k+1]$) or $H[k+1]$ (if $T[k] \neq T[k+1]$). If $T[k]$ is a circled letter, we will update values in array H. Next, we will find the most right letter $S_0[l]$ which is the same with $T[k]$ and $h[l] = 0$. Then the new $H[r]$ will be equal to old $H[r+1]+1$ for r from l to $k-1$ (it's because the line in the r-th column is come from the $(H[r+1]+1)$-th row in the $(r+1)$-th column) no matter which i satisfies $S_i = T$. Now we have an algorithm without binary search and time complexity is $O(N^2)$. But it's still not fast enough to solve this problem.

Now we focus on the operation: new $H[r]$ = old $H[r+1]+1$ for r from l to $k-1$. This operation just give the the part of array with index from $l+1$ to k value of H in array one unit of left offset and add one to all of them and we don't care the value in array H not small than k more! So we can maintain another array h along with two values $offset_{value}$ and add_{value} such the the value $h[i + offset_{value}] + add_{value}$ equals to $H[i]$. After that, in each time we want to do this operation we can only increase both $offset_{value}$ and add_{value} by one(with

some more detail not covered here). Finally, we get an algorithm with time complexity $O(N)$ which can solve this problem!