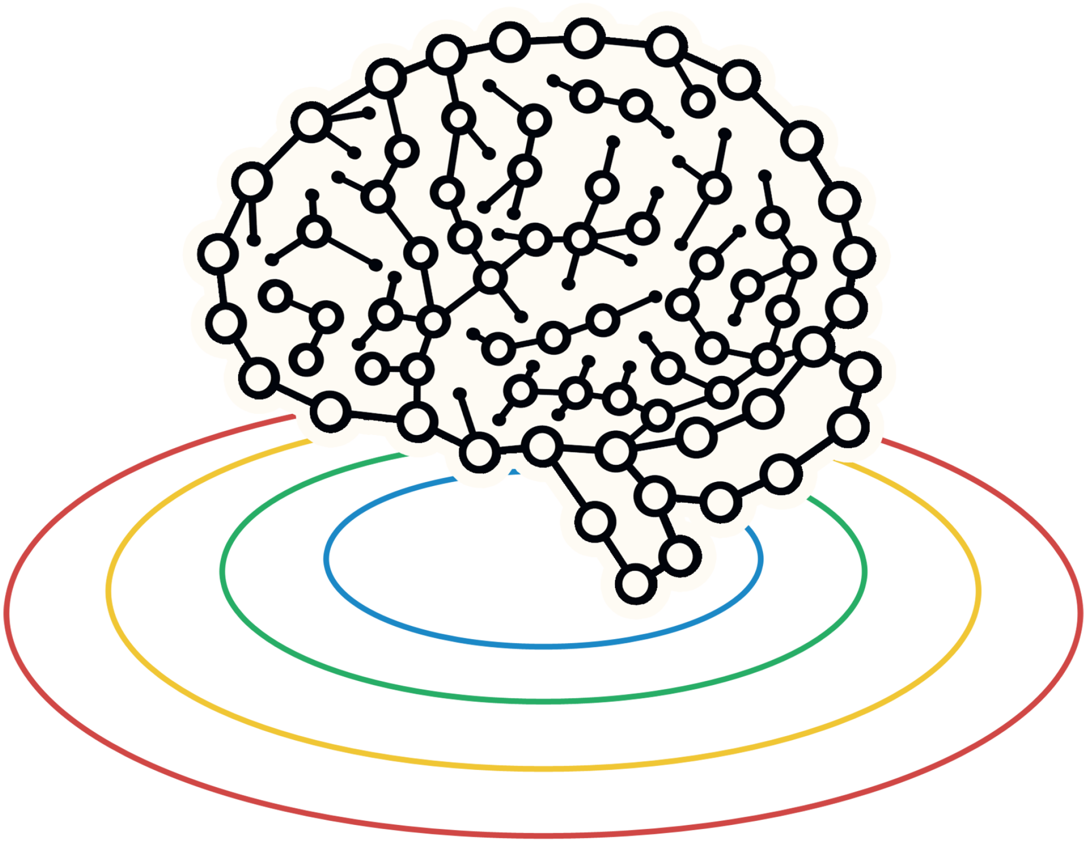
Association for Computing Machinery
University of the Philippines Diliman Student Chapter, Inc.



ALGOLYMPICS 2020

UP ACM PROGRAMMING COMPETITION

SOLUTIONS FOR THE FINAL ROUND PROBLEMS

Introduction

We hope you enjoyed the problems from the UP ACM Algolympics On-Site Final Round!

You can view the solutions of the problems here: https://algo2020.upacm.net/resources/algo2020finals_solutions.zip

We feel that this round's problems are more interesting and harder on average than the Elimination Round problems. We believe it is on par with an average ICPC regionals problem set.

We also think that the participants underperformed during the final round. We urge you to train further to get much better results and fare better next time, especially in higher-stakes contests like ICPC.

The problems will be available on the Algolympics Codeforces group page for practice. They will be available forever.[1]

If you wish to discuss the problems (hints, solutions, etc.) with fellow participants and/or the Algolympics scientific committee members, a Discord server has been set up just for that. Please email algolympics@upacm.net with your name, team name and school to get an invite to the Algolympics Discord server.

Many members of the Algolympics scientific committee are also part of the National Olympiad in Informatics - Philippines (NOI.PH), an algorithmic competition aimed at high-school students. We invite you to say hello in our Discord! Please email ask@noi.ph with your name and school if you wish to have access to the NOI.PH Discord server.

If you need more practice, please stay tuned for the NOI.PH on-site final round on April. A mirror will be held around the same time which you can join for practice. The NOI.PH online elimination round is also still available for practice.

All past problems of official NOI.PH contests can be accessed through this link: https://noi.ph/past-problems.

Please regularly check your email, the Algolympics Discord and UP ACM's Facebook page for news and other potentially important announcements.

Thanks for joining Algolympics 2020, and we hope to see you next year!

[1] where "forever" means "for as long as Codeforces exists" :P

Contents

Problem A: The Slowden Files

Setters: Kevin Atienza

Testers: TBA

Statement Authors: JD Dantes

Test Data Authors: JD Dantes

Editorialists: JD Dantes

A.1 Solution

The minimum moves needed to turn one string to the other using the given operations (add, remove, or replace a character) is more commonly known as the *Levenshtein distance* or *string edit distance*. Getting the Levenshtein distance of two strings is a classic dynamic programming (DP) problem. If you are new to dynamic programming, I encourage you to read up and search about it as there are many resources online; I have also written and compiled https://gitlab.com/jddantes/the-code-project/-/tree/master/dynamic-programming before.[2]

For the Levenshtein distance specifically, try coming up with the DP recurrence. If you analyze it, you'll find that you can find the Levenshtein distance in $O(mn)$ time complexity. However, for this problem, the bounds are in the order of 10^5, so the usual $O(mn)$ approach would lead to TLE. In other words, implementing this DP approach directly is not the intended solution for this problem. What else can we try?

We observe that we are not looking for the general case, but only for a smaller, more specific subset: edit distance from 0 to 3. Thus, we can try going through some test cases for these smaller edit distances.

If the allowable distance is zero, then we simply check if the two strings are equal.

The next case is if the distance is exactly one. There are two cases for this. The first is if the lengths of the strings are exactly one character apart. In this case, we must erase one character from the longer string, and then both strings should match. The second case is if the two strings are of the same length. In this case, there should be exactly one index where the characters in both strings are mismatched.

Let's now go to the scenario where the distance is exactly two. We consider the cases where the lengths of the two strings differ by zero, one, or two only; we note that generally, if the distance between two strings is d, then the difference between their lengths should only be at most d as well. Let's denote the two strings as A and B, with $|A| \leq |B|$ (where we use $|\cdot|$ to denote the length). We can look at the extreme cases:

- $|B| - |A| = 2$. Similar to before, we choose two characters to delete from B. We can keep matching A and B from the start, then delete from B if a mismatch occurs.
- $|B| - |A| = 0$. The lengths are equal, so there could be two indices where A and B are mismatched (e.g., "`abcde`" and "`xbcdy`"). It could also be the case where it's better

[2]See https://gitlab.com/jddantes/the-code-project/-/tree/master/dynamic-programming

to shift or align the strings (e.g., "`xabcd`" and "`abcdy`") rather than trying to match the A_i and B_i if they are not equal.

Thus, we still account for the different moves that we can take, and not just rely on simply erasing or matching characters at the current index. If $f(i, j)$ refers to the suffixes starting at A_i and B_j, then we can make use of the familiar Levenshtein recurrence. If $A_i = B_j$, we can proceed to '$f(i+1, j+1)$'. Otherwise, we can explore the different actions:

- Match the characters. Move to '$f(i+1, j+1)$'.
- Delete from A. Move to '$f(i+1, j)$'.
- Delete from B. Move to '$f(i, j+1)$'.

The difference from the plain Levenshtein formulation is that we know the maximum number of edits that we are allowing, and we can include it as a parameter in the recursion to limit the search space. So let's use a new parameter, "e", to denote the maximum number of allowed edits. Then from '$f(i, j, e)$', we move to '$f(i+1, j+1, e)$' if $A_i = B_j$. Otherwise, we move to '$f(i+1, j+1, e-1)$', '$f(i+1, j, e-1)$', or '$f(i, j+1, e-1)$' as listed before.

This may still look like exponential time, but as the maximum number of allowed edits is small, we end up with what's essentially '$O(m+n)$' time with a reasonably small constant factor. We can quantify this precisely by considering the more general problem where the maximum allowed edits is k. Then the complexity of this approach is $O(3^k(m+n))$. For this problem, $k = 3$, so the constant is small.

Alternatively, you can attempt to memoize $f(i, j, e)$ with an additional case: if the current suffixes' lengths differ by more than e, then we stop early and prune the search, since we already know that their edit distance is more than e. Using memoization, the complexity becomes $O(k(m+n))$.

Problem B: C.U.P.S.

Setters: Kevin Atienza

Testers: Josh Quinto

Statement Authors: Patrick Celon

Test Data Authors: Kevin Atienza

Editorialists: Josh Quinto, Kevin Atienza

B.1 Problem Summary

You are given n craters, each crater initially either open or closed. In one move, you can flip (open $\rightarrow$ closed, closed $\rightarrow$ open) the state of exactly m craters. You are tasked to find a series of at most n moves such it results in all craters being closed or determine if it is impossible to do this.

B.2 Solution

With n up to 80, and m bounded only by n, naive solutions that involve complete search will result in a Time Limit Exceeded verdict, even with efficient pruning. Experienced solvers would have the intuition to pursue a greedy solution – and they would be right!

B.2.1 Insight 1: Two-flip strategy

The first major insight needed is that you can flip any two arbitrary craters within two moves, while retaining the states of others. This is demonstrated by the example below.

```
            1110111100100
Move 1      --XXAXX------
Move 2      --XX-XX-B----
Result      1110011110100
```

In the above example, we have $n = 13$ and $m = 5$. We flip the 5th and 9th craters by choosing to flip them once each on the first and second move, respectively, while flipping 4 other craters twice. By flipping the 4 others twice, we retain their original states.

In general, we can choose two craters A and B to flip once, then choose any arbitrary $m - 1$ craters (that isn't A and B) to flip twice. This works for any $m < n$ (when $m = n$, this devolves into a special case that's even easier to solve – more on this later).

In the case that we have O open craters, then we can reduce the number of open craters by exactly 2 every two moves. If O is even, then we'll need O moves to close all craters. Since O is at most n, then we have found a solution for even O! What about for odd O?

B.2.2 Insight 2: Odd O

With odd O, then we must first consider the parity of m. If m is even, then it is unsolvable. This is because by flipping any even number of craters, we do not change the parity, hence

O will always be odd – a non-zero number.

If m is odd, then flipping any m craters once will flip the parity, making O even. Then we can solve it using the algorithm mentioned above for even O.

However, be careful in choosing the craters to flip initially – if the crater states after the initial flip result with $O = n$, then it cannot be solved as we only have $n - 1$ flips left, as demonstrated below.

```
            100000     (1 means closed, and 0 means open)
Move 1      X-----     (m=1)
Result      000000     (not solvable in 5 moves)
```

As such, ensure that there is at least one open crater chosen in the initial flip.

B.2.3 Special case: $m = n$

In the case where $m = n$, the only two cases that are solvable are when $O = n$ or $O = 0$ (all are open, or all are closed). When $O = n$, we flip all craters once (which is always possible since $n \geq 1$). Otherwise when $O = 0$, we don't flip anything. Of course, any other case is impossible because there is really only one possible move, which is to flip everything.

B.2.4 Alternative solution: BFS

There is an alternative solution that uses breadth-first search (BFS). We can think of a sequence of moves as a sequence of integers $O_0, O_1, O_2, \dots, O_k$, where O_i is the number of open craters after the ith move. A valid sequence of moves is one that:

- begins with $O_0 = O$, the initial number of open craters;
- ends with $O_k = 0$, i.e., all craters closed;
- for every i from 1 to k, it is possible to go from O_{i-1} open craters to O_i open craters by flipping exactly m craters.

The key thing to notice here is that it doesn't really matter *which* O_i craters are currently open; if there is at least one selection of O_i open craters such that it is possible to get O_{i+1} open craters in exactly one move, then the same is true for *any* selection of O_i open craters, by simply permuting the craters (and the move) properly.

Therefore, for each n and m, we can build this (undirected) graph of n nodes and up to n^2 edges, and simply go *backwards*, i.e., find all the reachable nodes from the state with 0 open craters, and also find the *shortest path* to them using BFS. We precompute the shortest paths along with the predecessors. Then, for each test case, we simply construct the shortest path from the precomputed stuff, or determine that it is impossible.

The advantage of this approach is that we don't have to deal with lots of cases. For example, the case $m = n$ is not a special case, and also, even and odd m are handled identically. We don't even have to know that all nodes are solvable in at most n moves; we can just check if the shortest path is at most n. Of course, this alternative formulation does also yield a proof that at most n moves are needed; there are only $n+1$ nodes, and every shortest path only goes through any node at most once, so therefore, any shortest path is at most n in length.[3]

The running time of the precomputation for a fixed (n, m) pair is linear in the size of the graph, i.e., $O(n^2)$. Thus, the overall running time is $O(n_{\max}^2 \cdot n_{\max}^2) = O(n_{\max}^4)$. Afterwards, each test case can be answered in time linear in the size of the output, which is $O(n^2)$, since the shortest path can be extracted/reconstructed in a straightforward way. (Perhaps the trickiest part is determining which m craters to flip, but this can be done with a little bit of arithmetic.)

[3]Can you also show that this bound is tight?

Problem C: Senpai

Setters: Josh Quinto

Testers: TBA

Statement Authors: Patrick Celon

Test Data Authors: Patrick Celon, Kevin Atienza

Editorialists: Patrick Celon, Kevin Atienza

C.1 Solution

C.1.1 The formula

We need to select the functions P_i so that the following are satisfied:

$$P_i(0) = 0 \text{ for all } i \text{ from } 1 \text{ to } q \tag{1}$$

$$\sum_{i=1}^{q} \left(\frac{d}{dt} P_i(t) \right)^2 \le g^2. \tag{2}$$

Furthermore, they must be selected so that the minimum $t \ge 0$ that satisfies the following inequality is minimized:

$$\sum_{i=1}^{q} S_i(t) \le \sum_{i=1}^{q} P_i(t) \cdot W_i \tag{3}$$

where

$$S_i(t) = F_i \cdot t + C_i. \tag{4}$$

We can start by simplifying the inequality: on the left side, note that F_i and C_i are all constants, so their summations are also constant:

$$\sum_{i=1}^{q} S_i(t) = \sum_{i=1}^{q} F_i(t) \cdot t + \sum_{i=1}^{q} C_i(t) = F_{\text{sum}} \cdot t + C_{\text{sum}}. \tag{5}$$

F_{sum} and C_{sum} are the sums of F_i and C_i across all qualities.

Next, we recognize that the right side, $\sum_{i=1}^{q} P_i(t) \cdot W_i$, is simply the *dot product between two vectors* $P(t)$ and W, with individual components $P_i(t)$ and W_i respectively. In other words, $P(t) \cdot W$ is a scalar measuring *the component of* $P(t)$ *in the direction of* W.

Our goal is to find t_{min}, the minimum time $t \ge 0$ such that this dot product exceeds $F_{\text{sum}} \cdot t + C_{\text{sum}}$ *with the optimal choice of* P.

Let's now think about what the optimal choice of $P(t)$ should be. We want the dot product on the right to grow as fast as we can. We can choose $P(t)$ to be any continuously differentiable function satisfying (2). Let's think about what (2) says. It says that, at any moment, the instantaneous rate of change of our function, $P'(t)$, has magnitude at most g, i.e., $\|P'(t)\| \le g$. Think of this as the *speed* limit, and $P(t)$ is the path that we're taking in q-dimensional space.

Since we're maximizing, it makes sense to "maximize" this speed limit as well, i.e., choose that $\|P'(t)\| = g$ always. Also, note that we're only maximizing the *component of* $P(t)$ *in the direction of* W. Therefore, intuitively, it makes sense that *the optimal choice is that* $P'(t)$ *should point in the same direction as* W*, but with magnitude exactly* g. This forces $P'(t) \cdot W$ to be exactly $g\|W\|$, and thus, $P(t) \cdot W = g\|W\|t$.

Theorem C.1. *If* $P(t)$ *is chosen optimally, then* $P(t) \cdot W = g\|W\|t$.

Of course, the previous argument doesn't constitute an actual proof, since we mostly went by intuition. Unsurprisingly though, our intuition didn't fool us, since it can easily be translated to a formal proof. We will show the formal proof later, though you might want to try writing it on your own.

Now that we know what $P(t)$ should be, we should figure out t_{min} next. Using (5) and Theorem C.1, it is simply the smallest $t \geq 0$ such that

$$F_{\text{sum}} \cdot t + C_{\text{sum}} \leq g\|W\|t,$$

i.e., the smallest t such that

$$t \cdot (g\|W\| - F_{\text{sum}}) \geq C_{\text{sum}}.$$

Now, I know you're very tempted to divide by $(g\|W\| - F_{\text{sum}})$ at this point, but not so fast! Note that $ab \geq c$ implies $a \geq c/b$ *only if* $b > 0$. If $b < 0$, then the inequality reverses. Finally, if $b = 0$, then it isn't even valid since we'd divide by zero!

And the value "$(g\|W\| - F_{\text{sum}})$" can be any sign!

Instead, let's analyze it conceptually. The above is equivalent to

$$(g\|W\| - F_{\text{sum}})t - C_{\text{sum}} \geq 0.$$

In other words, we have some line function, $L(t)$, and we want to find the first time it "rises above 0." We can think about three cases, depending on the slope of L:

- If the slope of L is zero, then it is just a constant. Therefore, either no t satisfies $L(t) \geq 0$ (and there's no solution), or *all* of them do (and the minimum one is $t_{\text{min}} = 0$). But the problem guarantees there is a solution, so it must be the second case, $t_{\text{min}} = 0$.
- If the slope of L is negative, then the largest value of L at the interval $t \in [0, \infty)$ occurs at $t = 0$. Therefore, if there is at least one t satisfying $L(t) \geq 0$ (which is guaranteed by the problem), then $L(0) \geq L(t) \geq 0$, i.e., $t = 0$ is also valid, hence, the answer must be $t_{\text{min}} = 0$.
- If the slope of L is positive, then the earliest time when $L(t) \geq 0$ is its root, $L(t) = 0$. Let t' be this root, i.e., the unique t' such that $L(t') = 0$. If $t' \geq 0$, then t_{min} must be t'. However, if $t' < 0$, then the answer must be $t_{\text{min}} = 0$.

You can also arrive at the same conclusion algebraically.

Armed with this, we now have the mathematically correct answer:

- If $g\|W\| - F_{\text{sum}} \leq 0$ or if $C_{\text{sum}} \leq 0$, then $t_{\text{min}} = 0$.

- Otherwise, $t_{\text{min}} = \dfrac{C_{\text{sum}}}{g\,\|W\| - F_{\text{sum}}}$.

The case "$g\,\|W\| - F_{\text{sum}} \le 0$" corresponds to L having negative slope, while the case "$C_{\text{sum}} \le 0$" (and positive slope) corresponds to L having a negative root. Note that these are different cases.

C.1.2 Precision issues

Unfortunately, if you implement the solution above directly, you might find yourself getting "Wrong Answer". This is due to precision errors; the formula

$$\frac{C_{\text{sum}}}{g\,\|W\| - F_{\text{sum}}}$$

is numerically terrible![4]

In particular, this happens in the case where $g\,\|W\|$ and F_{sum} are *very close to each other*. In that case, if we subtract them, their most significant bits will cancel out, leaving the lower bits as the new most significant bits. Therefore, we lost some bits of significance!

In general, subtraction of two nearly equal numbers suffers from a problem known as "catastrophic cancellation".

To get around this, we take advantage of the fact that g, $\|W\|^2$ and F_{sum} are integers, and that subtraction of integers doesn't suffer from catastrophic cancellation (as long as you are using an integer data type!). Also, assuming $F_{\text{sum}} > 0$, note that

$$g\,\|W\| - F_{\text{sum}}$$

being close to zero is basically the same as

$$g^2\,\|W\|^2 - F_{\text{sum}}^2$$

being close to zero, but the latter is an integer, so we'd like to use the latter expression somehow. No worries; we can just "*multiply by the conjugate*":

$$\begin{aligned}\frac{C_{\text{sum}}}{g\,\|W\| - F_{\text{sum}}} &= \frac{C_{\text{sum}}}{g\,\|W\| - F_{\text{sum}}} \cdot \frac{g\,\|W\| + F_{\text{sum}}}{g\,\|W\| + F_{\text{sum}}} \\ &= \frac{C_{\text{sum}}\left(g\,\|W\| + F_{\text{sum}}\right)}{g^2\,\|W\|^2 - F_{\text{sum}}^2}.\end{aligned}$$

This is a mathematically equivalent formula. However, this formula doesn't suffer from precision issues! This is because in the numerator, we're *adding* two floating point numbers. Furthermore, the denominator is an integer, so we don't lose any significance, as long as we use integer data types to compute it. Note that $\|W\|^2$ must be computed as simply the sum of the squares of the W_i, without doing the square root, so that we stay at integers.

[4] C++ solutions are a bit lucky in this regard, since **`double`** behaves exactly the same as **`long double`** in Windows. So, C++ solutions might be ok. See: `https://docs.microsoft.com/en-us/cpp/cpp/fundamental-types-cpp?view=vs-2019`

However, there is still one issue. The trick only works if the denominator is *positive*. And the denominator can certainly be negative; although g and $\|W\|$ are nonnegative, F_{sum} could be a very negative number that dominates both of them, so the denominator becomes negative overall. And in this case, the *addition* expression "$(g\|W\| + F_{\text{sum}})$" now becomes the catastrophic one!

Luckily, this last issue is also easy to solve. Note that F_{sum} is negative, so the *subtraction* expression "$(g\|W\| - F_{\text{sum}})$" is now the numerically stable one, and so we should just use the original formula instead!

As an alternative to this formula manipulation, we could also use *bisection* to find the answer, since the problem guarantees that the answer is between 0 and 5000. In fact, using bisection, we can do away with division altogether, and work directly with the inequality

$$F_{\text{sum}} \cdot t + C_{\text{sum}} \le g\|W\|t.$$

However, note that the special cases "$g\|W\| - F_{\text{sum}} \le 0$" and "$C_{\text{sum}} \le 0$" must still be handled separately. You can do that by simply checking if $t = 0$ already satisfies the inequality.[5]

C.1.3 Proof of optimality

We now wish to prove Theorem C.1.

Proof. Note that the specified bound can be achieved by choosing $P(t) = \frac{W}{\|W\|}gt$. This is a valid choice because it is just a linear function (which is definitely continuously differentiable), and that

$$\|P'(t)\| = \left\|\frac{W}{\|W\|}g\right\| = \frac{\|W\|}{\|W\|}g = g \le g.$$

However, this choice isn't defined when $\|W\| = 0$. But in that case, W is just the zero vector, so the dot product is always 0 *regardless of our choice of* $P(t)$. Therefore, "$P(t) \cdot W = g\|W\|t$" is still true *for any* P, so we can just choose $P(t)$ to be anything.

All that remains is to show that this is optimal. Let $P(t)$ be any function satisfying the conditions of the problem. We want to show that $P(t) \le g\|W\|t$.

Let

$$f(t) := P(t) \cdot W = \sum_{i=1}^{q} P_i(t) \cdot W_i.$$

Note that $f(0) = 0$.

Differentiating once (which is possible since P_i is differentiable), we get

$$f'(t) = \sum_{i=1}^{q} P_i'(t) \cdot W_i = P'(t) \cdot W.$$

[5] However, that doesn't mean that this approach doesn't suffer from some numerical issues on its own. What can possibly go wrong with this approach?

Since P_i is continuously differentiable, f' must be continuous, and hence, integrable. Therefore, if we integrate:

$$\begin{aligned}\int_0^t (P'(u) \cdot W) du &= \int_0^t f'(u) du \\ &= f(t) - f(0) \\ &= f(t).\end{aligned}$$

However, remember that $v \cdot w \leq \|v\|\|w\|$.[6] Using this gives us the desired bound for $f(t)$:

$$\begin{aligned}f(t) &= \int_0^t (P'(u) \cdot W) du \\ &\leq \int_0^t \|P'(u)\| \, \|W\| \, du \\ &\leq \int_0^t g \, \|W\| \, du \\ &= g \, \|W\| \int_0^t du \\ &= g \, \|W\| \, t.\end{aligned}$$

□

[6]This is called the Cauchy-Schwarz inequality.

Problem D: Move to Remove Confidential Blunders

Setters: Kevin Atienza

Testers: Alvin Burgos

Statement Authors: JD Dantes

Test Data Authors: Josh Quinto

Editorialists: JD Dantes

D.1 Solution

To check if we can give access, we can hardcode the string rankings with their corresponding age thresholds with a couple of ‘if-else’ statements. Another approach is to use a data structure to map a string ranking to an integer threshold (e.g., C++ ‘`map`’, Python ‘**`dict`**’). Just be careful with the equalities/inequalities when writing your conditions.

Note that the title in the input is not needed, and you can even choose to not read it at all. In fact, it might be better not to read it to minimize the risk of getting an error or something.

Problem E: A Floor of Many Doors

Setters: Barbara David

Testers: Josh Quinto

Statement Authors: Barbara David

Test Data Authors: Kevin Atienza

Editorialists: Josh Quinto

E.1 Problem Summary

You are given a grid of doors ('D'), empty spaces ('.'), and walls ('#'). You are also given the starting ('A') and ending ('B') positions. You must find the minimum number of moves to reach the destination while keeping open a maximum of k doors at a time or determine if it is impossible.

Here, the following are considered moves: moving up/down/left/right one cell, opening a closed door at an adjacent cell, and closing an open door at an adjacent cell.

E.2 Solution

Without the restriction of the doors, this seems to be a straightforward shortest path problem, one solvable by BFS. If we do consider the doors, but ignore k, then moving to a cell with a closed door will cost us two moves instead of just one.

Since our edges are weighted now, this can instead be solved by the $O(E \log V)$ variant of Dijkstra's Algorithm (this can be optimized further, to be discussed later).

E.2.1 Door-limited operations

If we consider the limit of k open doors at a time, then it becomes slightly harder. That is, we must close some doors as we pass through them. This can lead to several pitfalls and naïve strategies.

The key observation here is that if we do decide that a certain door should be closed *eventually* to open another door, then the best time to close that door is as soon as we have *passed* it, since we won't be returning to the same spot.[7] Let's consider the 1D case below. It can be shown that 2D paths can be likewise expressed in terms of 1D strips similar to that shown (without loss of generality).

```
K = 3
A..D.DDD..B
      ^
```

Let's imagine that you are already on the 3rd door (which is open). You have 2 doors to your left already opened. In order to open the next door, you need to have either the first or

[7] It should be clear that there's no point in returning to the same spot after the first time you reach it. Proving this rigorously can be finicky, but it's possible. It's also unsurprising, since it's quite intuitive anyway.

second door closed (you can't close the door you are currently in). For example, if we choose to close the first door, then a valid set of operations could be:

- Move right twice (2 moves)
- Open D_1 (1 move)
- Move right twice (2 moves)
- Close D_1 and open D_2 (2 moves)
- Move right once (1 move)
- Open D_3 (1 move)
- Move right once (1 move)
- Open D_4 (1 move)
- Move right four times, reach exit (4 moves)

We have shown that if we need to close a door, then we can arbitrarily choose any of the passed doors which are still open (except possibly the last one). That is if we have k doors open, we can choose $k-1$ doors to open.

What is important here is that we don't really need to know which door we will close, since an open door is always available to close (except when $k=1$, which is a special case – more on this later). We only need to note that an added cost of 1 move is added in the case that we have k doors open and we need to open an additional door.

So now, we need to keep track of the position and the number of opened doors so far, (r,c,d). The moves can be generalized as follows:

1. Move to an open space $[(r,c\pm 1,d),(r\pm 1,d)]$—$1$ move.
2. $d<k$, move to a closed door $[(r,c\pm 1,d+1),(r\pm 1,d+1)]$—$2$ moves.
3. $d=k$, move to a closed door $[(r,c\pm 1,k),(r\pm 1,k)]$—$3$ moves.

This can still be solved using a shortest path on a $r\times c\times(k+1)$ state graph. With the sum of rc at most 3×10^5 and k at most 50, this is a valid strategy.

E.2.2 Special case: $k=1$

When $k=1$, however, we cannot go through two consecutive doors. This limits our move-set to the following:

1. Move to an open space $[(r,c\pm 1,d),(r\pm 1,d)]$—$1$ move.
2. $d=0$, move to a closed door $[(r,c\pm 1,0+1),(r\pm 1,0+1)]$—$2$ moves.
3. $d=1$, on an open area, move to a closed door $[(r,c\pm 1,1),(r\pm 1,1)]$—$3$ moves.

Note that we can only move to a closed door from an open space. Apart from the added restriction, this is still solvable using a shortest path algorithm on a state graph.

E.2.3 Extended BFS

Though Dijkstra's *could* pass as the shortest path algorithm, there is no guarantee. Instead, we could make use of the fact that the move costs are integers up to 3.

Consider the BFS pseudocode below:

```
Qt = []
Qtplus1 = []
while Qt is non-empty:
   for each state in Qt:
      if state has already been visited: continue
      mark state as visited
      for each adjacent state:
         push to Qtplus1
   Qt = Qtplus1
   Qtplus1 = []
```

That is, we assume that all moves have uniform (1) cost, and states from `Qt` are pushed to `Qtplus1`. If we extend this to consider up to a cost of 3,[8] then:

```
Qt = []
Qtplus1 = []
Qtplus2 = []
Qtplus3 = []
while Qt is non-empty:
   for each state in Qt:
      if state has already been visited: continue
      mark state as visited
      for each adjacent state:
         if cost is 1:
            push to Qtplus1
         elif cost is 2:
            push to Qtplus2
         elif cost is 3:
            push to Qtplus3
   Qt = Qtplus1
   Qtplus1 = Qtplus2
   Qtplus2 = Qtplus3
   Qtplus3 = []
```

This makes everything run in $O(V+E)$, removing the log factor. This should fit well within the time limit.

[8]In fact, this can be generalized for an arbitrary number of L "look-aheads", but we have to rewrite our time and space analysis to $O(VL+E)$. Note that this is only practical for integer weights and relatively small L.

Problem F: One Great Grater

Setters: JD Dantes

Testers: TBA

Statement Authors: JD Dantes

Test Data Authors: Kevin Atienza

Editorialists: Kevin Atienza

F.1 Solution

While walking, we can write the complete current "state" of the walk using three numbers: (i, j, d) where (i, j) denotes the current location and $d \in \{\mathtt{U}, \mathtt{D}, \mathtt{L}, \mathtt{R}\}$ denotes the current direction. We will count rows or columns starting from 1, so $(1, 1)$ represents the top-left corner.

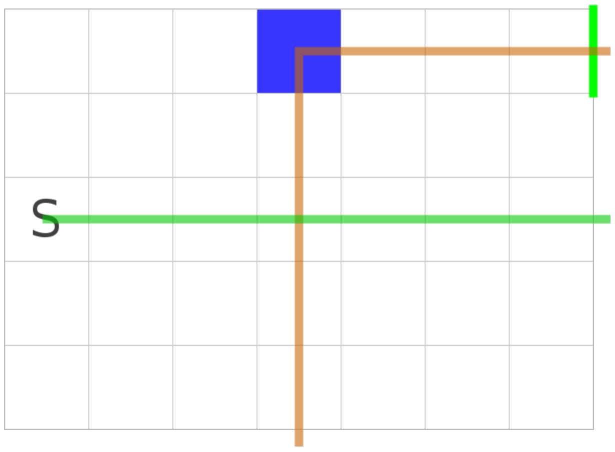
We can also create "dummy" states, corresponding to *walking into the wall segments*, by introducing extra cells at the borders of the grid, and then declaring that these dummy states don't have successors. For example, $(2, 0, \mathtt{L})$ corresponds to walking into the second left wall segment from the top, $(0, 5, \mathtt{U})$ corresponds to walking into the fifth top wall segment from the left, and so on.

Each state (i, j, d) has a unique successor (except for the dummy border states), which is clear from the statement. What's not as clear is the fact that each state also has a unique predecessor (if it exists). It can be shown easily: Suppose there are two different states that go to the same location (i, j). Then these two source states must have different locations and different directions. But then, the color of (i, j) determines the new direction in a one-to-one fashion (since turning left, turning right, or not turning at all is a one-to-one correspondence between directions), so the new directions must be different. Thus, two distinct states going to the same location yield different directions, and hence different states. (And of course, two distinct states going to different locations yield different states!)

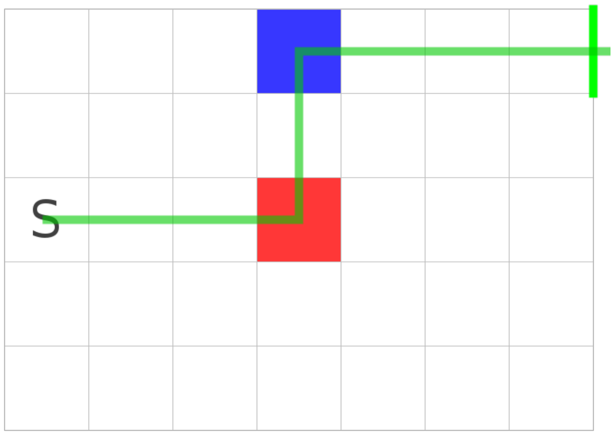
And actually, this unique predecessor can easily be computed. Just *undo* the turning caused by the current cell (if ever), and then undo the walk! So actually, all states (including the dummy states we made earlier) have a predecessor, if we introduce new dummy states corresponding to *starting from a wall segment*. For example, $(2, 0, \mathtt{R})$ corresponds to starting from the second left wall segment from the top, etc.

Thus, if we consider the *graph* of all states (i, j, d), then we find that there can never be any branching, and so we find that the only possible components are *cycles* and *paths*. Furthermore, the paths can only start and end at a wall segment, since these are the only states that don't have either a predecessor or successor!

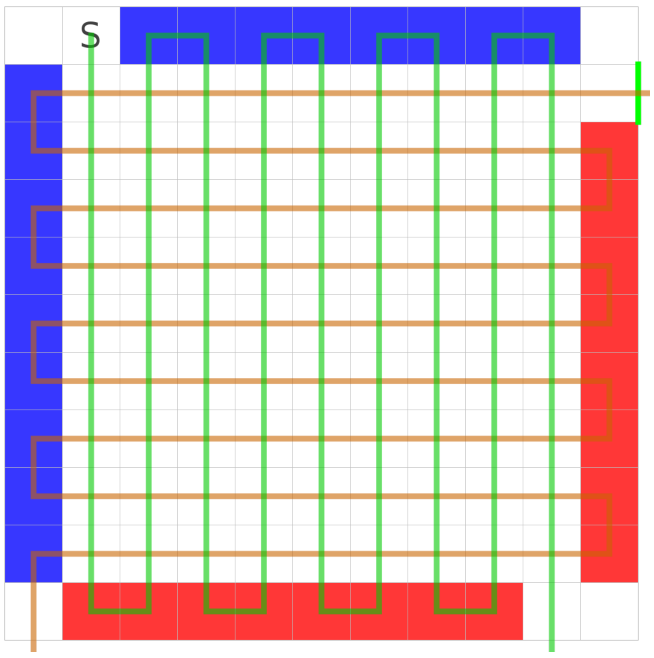
Now, what does changing the color of some cell (i, j) do to this graph? Well, all it does is it modifies the successors of (i, j, d) for all states starting at this cell. And there are only four of them. Usually, it just involves "tying" two distinct paths together. For example, in the simplest case, to get to the highlighted wall segment in the following:



we replace the “intersection” of their paths as follows:



Of course, you can’t simply check all possibilities, since that could take quite a long time. Consider this example:



Note that the downward path from S is very long and intersects a lot with the path towards the highlighted wall segment. Hence, if we try all possibilities, this will take a very long time—actually up to $O((rc)^2)$—and will not pass the time limit.

Instead, we need to find a way to compute the resulting path after modifying some cell, without actually doing the full walk every time. First, note that it only ever makes sense to modify a cell that belongs to some path from S in the initial grid. (Why?) Thus, we can simply compute all of those reachable cells, and for each of them, take note of its "initial direction", i.e., the direction during the first time we reach that cell.

Then, for each such white cell (i, j) with initial direction d, let's consider what happens if we change that cell into either red or blue. Both colors are processed similarly, so let's just consider one of them, say blue. Now, since the cell is blue, we will now turn right instead of going forward. This will take us to the new state $(i, j, right(d))$. We now want to figure out where this new state will end up in. But we can easily do this precomputation, since we know that the original state graph has a quite simple shape! Thus, we can obtain the eventual state with a simple lookup! Let's look at the possibilities:

- **The path from $(i, j, right(d))$ never revisits cell (i, j).** In this case, it cannot end up in a loop, and so the only possible outcome is that it ends up in a wall segment. Then we're done; we add this wall segment to our output!
- **The path from $(i, j, right(d))$ revisits cell (i, j).** Let (i, j, d') be the first state in the walk that revisits cell (i, j). Then in that case, since we modified the color of (i, j), we again teleport to the new state $(i, j, right(d'))$. And then, we repeat the procedure. After doing this repeatedly, either we end up in a wall segment, or we end up repeating some state at cell (i, j), in which case, we can stop the simulation since we must have already ended up on a loop!

Note that if we can compute the correct case in $O(1)$ (by preprocessing the graph), then this only takes $O(1)$ time, since there are at most 4 states in a cell!

Therefore, we can compute the result of changing each cell in $O(1)$. Since there are only $O(2rc) = O(rc)$ possible changes, the overall algorithm runs in $O(rc)$!

F.1.1 The Preprocessing

We must preprocess the graph of states (i, j, d) so that we can easily compute the following:

- The state corresponding to the first time it revisits cell (i, j), if it exists.
- The ultimate state, if it eventually ends up at a wall segment.

Let's denote this state as $follow(i, j, d)$.

After computing the graph (which has $4(rc + r + c)$ nodes), we can preprocess each path and cycle separately.

- For each path, collect all the distinct cells passed through that path. Then, for each such cell (i, j), collect the sequence of directions $(d_1, \ldots, d_k)$ corresponding to the states visiting that cell, in order of visitation. Then we determine that

$$\begin{aligned}
\mathit{follow}(i,j,d_1) &= (i,j,d_2)\\
\mathit{follow}(i,j,d_2) &= (i,j,d_3)\\
\mathit{follow}(i,j,d_3) &= (i,j,d_4)\\
&\dots\\
\mathit{follow}(i,j,d_{k-1}) &= (i,j,d_k),
\end{aligned}$$

and $\mathit{follow}(i,j,d_k)$ is the wall segment at the end of this path.

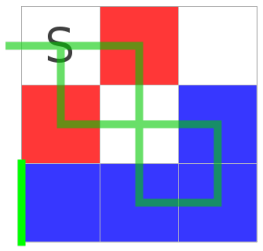
- A cycle is handled similarly, except that this time, $\mathit{follow}(i,j,d_k) = (i,j,d_1)$.

Each path and cycle is processed in time linear in the size of that path/cycle, and so the overall complexity is linear in the size of the graph, which is $O(4rc) = O(rc)$!

F.1.2 Gotchas

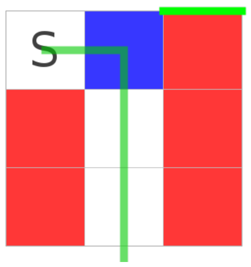
This problem is quite tricky. There are several ideas that sound reasonable but are actually wrong. Here are a couple.

Only process a cell on the initial visit. We must only process each cell *for the first time it is visited*, not for every time it is visited. Consider the following example, where the initial direction is "down":

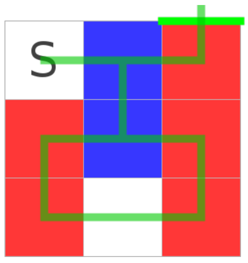


Note that the center cell is visited twice, the first time with direction R, and the second time with direction U. We must only process the direction R; if we wrongly process the direction U, say by going left after it, then we would arrive at the highlighted wall segment, which is actually unreachable.

Visiting a cell multiple times. It is tempting to guess that we only need the "single intersection" case, i.e., the case where the modified cell is only passed through once. It is incorrect! For example, it will be incorrect in cases like this:

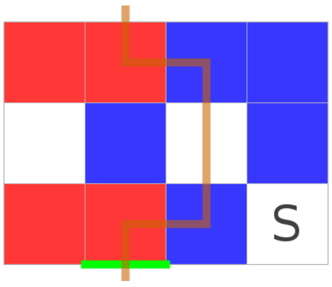


If we make the middle cell blue, then we get the following:

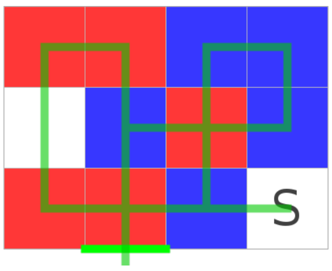


which ends up at the highlighted wall segment. Note that the modified cell was traversed twice!

It is not even enough to assume it only passes through the modified cell at most twice. Here's the smallest counterexample I could find:



If we modify the second white cell as follows:



then the highlighted wall segment can be reached. Note that the modified cell was passed through three times!

Curiously, it turns out that there are no cases where the path has to pass through the modified cell four times. Well, there are such cases, but you can show that you reach no new wall segments. The proof is actually quite clever, but in any case, the solution shouldn't depend on it. (Finally, of course there are no cases where the path passes through the modified cell more than four times, since then at least one state would repeat.)

Problem G: Generic Spy Movies

Setters: Patrick Celon

Testers: Kevin Atienza

Statement Authors: Patrick Celon

Test Data Authors: Josh Quinto

Editorialists: Patrick Celon

G.1 Solution

If there are a actors available for casting g guests per episode, there are always $\binom{a}{g}$ possible castings. There are many ways to enumerate such combinations. However the problem wants to find an ordering such that the change between two combinations is *minimal*, that is, only one element is changed. This corresponds to the 'one leaves, one enters' rule.

The problem also gives you a set starting point, but since this is simply an enumeration of combinations, we can reorder the list of a actors such that these first g guests appear lexicographically first! In short, the given initial sequence doesn't matter at all except for displaying who is subbed in or out.

We can do this enumeration in many ways.

G.1.1 Binary Reflected Gray Coding

Since we are limiting the changes at each step to be minimal, we can do something similar to Gray coding, but slightly different, since we're enumerating combinations.

One of the easiest ways to generate gray codes is to start at the minimal sequence, $(0,1)$ and build from there by creating a reflected second sequence, while adding 0 to the first, and 1 to the second, like so:

$$(0,1)$$
$$(00,10,11,01)$$
$$(000,100,110,010,011,111,101,001)$$
... and so forth.

This way generates a sequence in which exactly 1 position changes/flips at each step. However, in our case, we have to change at exactly 2 places - one in which there is a 1, and another position in which there is a 0. Furthermore, we need to have exactly g 1's at each step. This can be solved by modifying the reflection a little bit: First, we generate a valid combination sequence for $a-1$ actors and $g-1$ guests, and $a-1$ actors and g guests. We can see that both of these sequences sum up to the $\binom{a}{g}$ required. We can then append a 1 to the first one and add 0 to the second, while also reflecting second sequence, like normal reflected gray codes.

Starting from the sequences (0) and (1), the list can then be generated:

$$a=1:(0)\ \ (1)$$

$$a = 2 : (00)\ \ (01,10)\ \ (11)$$
$$a = 3 : (000)\ \ (001,100,010)\ \ (011,101,110)\ \ (111)$$

... and so forth.

This looks a lot like Pascal's Triangle, especially if we write it in the following form:

$$\begin{tikzcd}
 & & & \boxed{()} \arrow[dl] \arrow[dr] & & & \\
 & & \boxed{(0)} \arrow[dl] \arrow[dr] & & \boxed{(1)} \arrow[dl] \arrow[dr] & & \\
 & \boxed{(00)} \arrow[dl] \arrow[dr] & & \boxed{(01,10)} \arrow[dl] \arrow[dr] & & \boxed{(11)} \arrow[dl] \arrow[dr] & \\
\boxed{(000)} & & \boxed{(001,100,010)} & & \boxed{(011,101,110)} & & \boxed{(111)}
\end{tikzcd}$$

This also illustrates that the number of $\binom{n}{k}$ combinations possible and the $\binom{n-1}{k-1} + \binom{n-1}{k}$ expression that we are using right now are equal.

Again, note that the right parent is reflected, while the left parent is not.

The reason this works is the following.

Theorem G.1. *The construction has the following properties:*

- *For $0 < g \leq a$, the first terms of the enumeration for $(a, g-1)$ and (a, g) differ in exactly one position.*
- *For $0 < g \leq a$, the last terms of the enumeration for $(a, g-1)$ and (a, g) differ in exactly one position.*
- *For $0 < g < a$, the first and last terms of the enumeration for (a, g) differ in exactly one position.*

This is provable by a simple induction, which we'll leave you to figure out. And the last item in particular shows that the enumeration is *cyclic*.

Since we can generate the list, we only need to add the changes to that list like so:

```
combi_list = new empty list // list of all the g-combinations

def gen_comb(a, g, is_reflected, suffix):
   if g == 0:
      combi_list.add(string(a, '0') + suffix) // all zeroes
      return
   elif g == a:
      combi_list.add(string(a, '1') + suffix) // all ones
      return

   if not is_reflected:
      // first part of the list, add '0' to the suffix as you work your way down
      gen_comb(a-1, g-1, false, '1'+suffix)

      // second part of the list, add '1' to the suffix as you work your way down
      gen_comb(a-1, g, true, '0'+suffix)
```

```
    else:
        // reversed, so flip
        gen_comb(a-1, g, false, '0'+suffix)
        gen_comb(a-1, g-1, true, '1'+suffix)
```

If we run the function above, we generate the sequences described before. Furthermore, we notice that we start at the generation where all the 1's are at the rightmost, and end where they are shifted by 1 to the left by one step. In other words:

Theorem G.2. *For $0 \le g < a$,*

- *the first string given by* `gen_comb(a, g)` *is the string* $\underbrace{\texttt{000..0}}_{a-g}\underbrace{\texttt{11..1}}_{g}$.
- *the last string given by* `gen_comb(a, g)` *is the string* $\underbrace{\texttt{00..0}}_{a-g-1}\underbrace{\texttt{11..1}}_{g}\texttt{0}$.

Again, it can easily be proven by induction. Note that this also shows that the enumeration is *cyclic*.

From the sequence of combinations, you could easily work out the *transitions*, i.e., the removed and added elements between consecutive combinations. However, that would be too slow,[9] so instead, we'd like to generate the transitions directly. But since we have a general idea of the forms of the first and last element of each sequence, we can then determine the transition between two sequences. The pseudocode for generating the *transitions* is then:

```
change_list = new empty list // list of transitions between the g-combinations

def gen_changes(a, g, is_reflected):
    if g == 0 or g == a:
        return // no transitions as the sequence is a singleton

    if not is_reflected:
        // generate transition list for left parent
        gen_changes(a-1, g-1, false)

        // add transition between parents to change_list
        if g == a-1: // right parent is all ones
            change_list.add(a, a-1)
        else:
            change_list.add(a, a-g-1)

        // generate transition list for right parent
        gen_changes(a-1, g, true)
    else:
        // generate transition list for left parent (reversed)
        gen_changes(a-1, g, false)

        // add transition between parents to change_list
        if g == a-1: // left parent is all ones
```

[9]unless you use tricks like bitmasks.

```
            change_list.add(a-1, a)
        else:
            change_list.add(a-g-1, a)

        // generate transition list for right parent (reversed)
        gen_changes(a-1, g-1, true)
```

Of course, this solution enumerates *all* $\binom{a}{g}$ combinations, but the problem only asks for the first n. Enumerating all $\binom{a}{g}$ combinations and then removing all but the first n would be too slow, since $\binom{a}{g}$ is usually very large. Thus, we need to modify the code above to stop as soon as n combinations (or equivalently, $n-1$ changes) are already obtained. This is accomplished by adding lines throughout the code like the following:

```
    if change_list.size() >= n - 1: return
```

G.1.2 Backtracking

A more straightforward way to generate the solutions is to just use *backtracking* the same way you generate all g-combinations of a set of a numbers. Following the formula

$$\binom{a-1}{g-1} + \binom{a-1}{g} = \binom{a}{g},$$

we keep track of two things:

- The list of people *in the cast*, in_cast.
- The list of people *outside the cast*, out_cast.

We define a function generate(in_cast, out_cast) that generates all the transitions needed to traverse the cast. This function call represents the sequence of all g-combinations from a elements, where g is in_cast.length and a is in_cast.length + out_cast .length, assuming that the initial combination is in_cast.

We can implement generate(in_cast, out_cast) as follows:

1. Remove the last element in out_cast, and let it be out_last.
2. Recursively call generate(in_cast, out_cast). (This corresponds to $\binom{a-1}{g}$.)
3. Remove the last element in in_cast, and let it be in_last.
4. Place in_last in out_cast.
5. Record the transition (in_last is replaced by out_last).
6. Recursively call generate(in_cast, out_cast). (This corresponds to $\binom{a-1}{g-1}$.)
7. Place out_last in in_cast.

The first recursive call corresponds to all the g-combinations not containing out_last (there are $\binom{a-1}{g}$ of those), while the second corresponds to the ones containing it (there are $\binom{a-1}{g-1}$

of those). Before the second call, we simply swap `out_last` with any member of the current `in_cast`. Any one will do, so we just arbitrarily choose the last one in the list.

This corresponds to the following pseudocode:

```
change_list = new empty list // list of all the g−combinations

def generate(in_cast, out_cast):
   if in_cast.empty() or out_cast.empty():
      return // this corresponds to g == 0 or g == a. hence, no transitions

   // last element of out_cast
   out_last = out_cast.pop_back()

   // recurse for 1st parent
   generate(in_cast, out_cast)

   // last element of in_cast
   in_last = in_cast.pop_back()

   // swap in_last with out_last
   out_cast.push_back(in_last)
   // ... and record the transition
   change_list.add(in_last, out_last)

   // recurse for 2nd parent
   generate(in_cast, out_cast)

   // put out_last back
   in_cast.push_back(out_last)
```

One difference of this approach from the first one is that the sequence this generates is not necessarily cyclic (i.e., it cannot go back to the first generation by only swapping 1 element).

Again, this enumerates all combinations. Add the appropriate return lines as soon as you obtain the required number of combinations.

G.2 Editorial

Some enumerations of combinations are hard to compute, like the Beckett-Gray code, wherein the one who is going out should be the one who has been staying the longest.

Problem H: Maggie and Dana's Mass Supper

Setters: Kevin Atienza

Testers: TBA

Statement Authors: Patrick Celon

Test Data Authors: Kevin Atienza

Editorialists: Kevin Atienza

H.1 Solution

Let's denote the cell at the ith row and jth column as (i, j). We start counting at 0, so the top-left and bottom-right cells are $(0,0)$ and $(w-1, \ell-1)$, respectively. Also, let $p(i, j)$ be the number of paths from cell $(0,0)$ to the cell (i, j). Then the answer is simply $p(w-1, \ell-1)$.

Of course, we can compute $p(i, j)$ using the familiar recurrence $p(i, j) = p(i, j-1) + p(i-1, j)$, with the base case $p(0,0) = 1$, and $p(i, j) = 0$ for every blocked cell or cell outside the grid. However, this takes $O(\ell \cdot w)$ time, so it will not pass the time limit. We need to find a faster way.

H.1.1 The special cells

Clearly, we can't compute all $p(i, j)$. The number of empty cells themselves are also $O(\ell w)$. At best, we can compute $p(i, j)$ for just a *subset* of the cells.

Let's consider computing $p(i, j)$. If there were no blocked cells, then the answer is easily seen to be $\binom{i+j}{i}$. Let's denote this value as $c(i, j)$. Note that $c(i, j)$ is easy to compute; we can just precompute factorials and inverse factorials (modulo 104857601) up to $\ell + w$.

However, $p(i, j)$ is in general less than $c(i, j)$. But note that if we can compute by how much, exactly, then we can compute $p(i, j)$ as follows:

$$p(i, j) = c(i, j) - \text{(no. of paths to } (i, j) \text{ passing through some blocked cell)}.$$

So let's try to compute the number of paths to (i, j) passing through at least one blocked cell. Let's consider the *first blocked cell* in this path. This cell is either $(i'+1, i')$ or $(i', i'+\ell-w+1)$, since these are the only possible *first* blocked cell. But note that to reach $(i'+1, i')$, the previous cell must have been the cell (i', i'), since $(i'+1, i')$ is the *first* blocked cell. Similarly, to reach $(i', i'+\ell-w+1)$, the previous cell must have been $(i', i'+\ell-w)$. Therefore,

- The number of paths to (i, j) whose first blocked cell is $(i'+1, i')$ can be computed as follows:
$$p(i', i') \cdot c(i-(i'+1), j-i'),$$
since the path towards $(i'+1, i')$ must end up at cell (i', i') *and* must not pass through any blocked cell (there are $p(i', i')$ such paths), and the rest of the path, i.e., from $(i'+1, i')$ to (i, j), can be any path, regardless of any blocked cells in the way (there are $c(i-(i'+1), j-i')$ such paths).

- Similarly, the number of paths to (i,j) whose first blocked cell is $(i',i'+\ell-w+1)$ can be computed as

$$p(i',i'+\ell-w)\cdot c(i-i',j-(i'+\ell-w+1)).$$

Therefore, we can compute $p(i,j)$ as follows:

$$\begin{aligned}p(i,j)=c(i,j)\\&-\sum_{i'}p(i',i')\cdot c(i-(i'+1),j-i')\\&-\sum_{i'}p(i',i'+\ell-w)\cdot c(i-i',j-(i'+\ell-w+1)).\end{aligned}$$

But note that these recurrences only require us to compute $p(i,j)$ of the form (i,i) and $(i+\ell-w)$ (i.e., the cells adjacent to blocked cells), and there are only $2w$ such cells! Thus, we can compute $p(w-1,\ell-1)$ without computing *all* $p(i,j)$; we only need $2w$ of them.

The recurrence is $O(w)$ in length, so this algorithm is $O(w^2)$ which is still too slow. But this should feel like progress, since the number of states this time is now $O(w)$, which is seemingly more manageable. From this point on, we start seeking faster ways of computing them.

H.1.2 A better representation

Since we're now only considering special cells (i,j), let's introduce some new notation. Let:

$$\begin{aligned}p_L(i)&:=p(i,i)\\p_R(i)&:=p(i,i+\ell-w)\\u_L(i)&:=c(i,i)\\u_R(i)&:=c(i,i+\ell-w)\\c_L(i)&:=c(i-1,i)\\c_R(i)&:=c(i-1,i+\ell-w).\end{aligned}$$

We can now adapt the recurrences above using the new notation as follows:

$$\begin{aligned}p_L(i)&=u_L(i)-\sum_{j=0}^{i}p_L(j)\cdot c_L(i-j)-\sum_{j=0}^{i-(\ell-w)}p_R(j)\cdot c_R(i-j-(\ell-w))\\p_R(i)&=u_R(i)-\sum_{j=0}^{i}p_L(j)\cdot c_R(i-j)-\sum_{j=0}^{i}p_R(j)\cdot c_L(i-j).\end{aligned}$$

But now, writing the formulas this way, we now see what they really are: they are actually a bunch of convolutions. And that likely means that the Discrete Fourier Transform (DFT) is at play!

So now, let's use some *generating function* notation, and hope that we can massage these equations into something useful. Let's use:

$$\begin{aligned} P_L(x) &= \sum_{i\ge 0} p_L(i)x^i & P_R(x) &= \sum_{i\ge 0} p_R(i)x^i \\ U_L(x) &= \sum_{i\ge 0} u_L(i)x^i & U_R(x) &= \sum_{i\ge 0} u_R(i)x^i \\ C_L(x) &= \sum_{i\ge 0} c_L(i)x^i & C_R(x) &= \sum_{i\ge 0} c_R(i)x^i. \end{aligned}$$

In other words, we use capital letters to denote the generating function, and small letters for the coefficients. We can now use the standard generating function trick: Multiply both sides by x^i, then sum across i, and see where it goes! Let's start with p_R:

$$\begin{aligned} p_R(i) &= u_R(i) - \sum_{j=0}^{i} p_L(j)c_R(i-j) - \sum_{j=0}^{i} p_R(j)c_L(i-j) \\ \sum_{i\ge 0}(p_R(i))x^i &= \sum_{i\ge 0}\left(u_R(i) - \sum_{j=0}^{i} p_L(j)c_R(i-j) - \sum_{j=0}^{i} p_R(j)c_L(i-j)\right)x^i \\ \sum_{i\ge 0} p_R(i)x^i &= \sum_{i\ge 0} u_R(i)x^i - \sum_{i\ge 0}\sum_{j=0}^{i} p_L(j)c_R(i-j)x^i - \sum_{i\ge 0}\sum_{j=0}^{i} p_R(j)c_L(i-j)x^i \\ P_R(x) &= U_R(x) - P_L(x)C_R(x) - P_R(x)C_L(x), \end{aligned}$$

where in the last step, we note that multiplying two generating functions corresponds to convolving the coefficients. Similarly, for p_L, we get:

$$P_L(x) = U_L(x) - P_L(x)C_L(x) - x^{\ell-w}P_R(x)C_R(x),$$

where the $x^{\ell-w}$ factor is needed to ensure that the indices are aligned.

Thus, we get a system of two equations, where the "knowns" are U_L, U_R, C_L and C_R (since they are easy to compute), and the "unknowns" are P_L and P_R (since they are the ones we are computing via the DP above). In fact, the two equations are linear in P_L and P_R!

Also, note that the answer is $p_R(w-1)$, and so we need to solve for the coefficients of $P_R(x)$. So we need to "eliminate" P_L somehow. So, let's write the formulas above in a more "standard" form:

$$\begin{aligned} (1+C_L(x))P_L(x) + x^{\ell-w}C_R(x)P_R(x) &= U_L(x) \\ C_R(x)P_L(x) + (1+C_L(x))P_R(x) &= U_R(x) \end{aligned}$$

Since we want to eliminate P_L, we can scale the equations appropriately and then subtract, just like we usually do with linear equations. Thus, we multiply the first with $C_R(x)$ and the second with $1+C_L(x)$, and then subtract:

$$\begin{aligned} C_R(x)(1+C_L(x))\,P_L(x)\,+x^{\ell-w}C_R(x)^2\,P_R(x) &= U_L(x)C_R(x) \\ C_R(x)(1+C_L(x))\,P_L(x)\,+(1+C_L(x))^2\,P_R(x) &= U_R(x)(1+C_L(x)) \\ \left((1+C_L(x))^2-x^{\ell-w}C_R(x)^2\right)\cdot P_R(x) &= U_R(x)(1+C_L(x))-U_L(x)C_R(x) \\ P_R(x) &= \frac{U_R(x)(1+C_L(x))-U_L(x)C_R(x)}{(1+C_L(x))^2-x^{\ell-w}C_R(x)^2}. \end{aligned}$$

H.1.3 An FFT-based solution

At this point, it is now actually possible to arrive at a solution.

Note that this is a formula involving only the "known" ones U_L, U_R, C_L, C_R. We have successfully eliminated P_L! Thus, if we can compute the the first w terms of the generating function to the right, then we can compute the answer.

Thus, we can simply keep the first w terms of every generating function on the right, and they become "polynomials" instead of generating functions.

But then, all operations on the right-hand side become only *polynomial multiplications* and *divisions*, and so can be computed using the fast Fourier transform (FFT) in $O(w\log w)$![10]

The slowest part is actually computing the polynomial division (which also runs in $O(w\log w)$ but has a high constant factor), but thankfully, there is only one such operation.

Thus, the answer can be computed in $O(w\log w)$ time!

Note that the modulus, 104857601, is $2^{22}\cdot 25+1$, and so we can easily compute up to 500000 terms using a number-theoretic version of the FFT.

H.1.4 A formula-based solution

We can actually derive an explicit formula for the answer by continuing to manipulate our generating function expression for $P_R(x)$ above. The key is to notice that the generating functions U_L, U_R, C_L, C_R actually have expressions in terms of the generating function of the *Catalan numbers*. Recall that the Catalan number $c(n)$[11] is defined as the number of paths from $(0,0)$ to (n,n) that stays "below the diagonal", and we have an explicit formula for them:

$$c(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Also, recall that their generating function is given by

$$C(x) := \sum_{n\ge 0} c(n)x^n = \frac{1-\sqrt{1-4x}}{2x}.$$

We will not prove these since these are standard results found in almost any generatingfunctionology book that's written by a person with surname Wilf.

[10] By "division", we mean multiplication by the inverse generating function.

[11] This is not the standard notation, but we're using "$c(n)$" for consistency with our earlier notation.

Theorem H.1. *For a fixed $k \geq 0$, the generating function of the $c(i,i+k)$s is given by*

$$C_k(x) := \sum_{i\geq 0} c(i,i+k)x^i = \frac{C(x)^k}{\sqrt{1-4x}},$$

where $C(x)$ is the generating function of the Catalan numbers.

Proof. We will prove the case $k = 0$ algebraically, and then the rest combinatorially. For $k = 0$, note that $c(i,i) = \binom{2i}{i}$, which is very close to the Catalan number formula. In fact, using the d/dx trick, we can show that:

$$\begin{aligned}
C(x) &= \sum_{i\geq 0} \frac{1}{i+1}\binom{2i}{i}x^i \\
xC(x) &= \sum_{i\geq 0} \frac{1}{i+1}\binom{2i}{i}x^{i+1} \\
\frac{d}{dx}(xC(x)) &= \frac{d}{dx}\sum_{i\geq 0} \frac{1}{i+1}\binom{2i}{i}x^{i+1} \\
\frac{d}{dx}\left(\frac{1-\sqrt{1-4x}}{2}\right) &= \sum_{i\geq 0} \binom{2i}{i}x^i
\end{aligned}$$

The right side is exactly what we want—$C_0(x)$—while the left side can easily be simplified to $\frac{1}{\sqrt{1-4x}}$. This proves the theorem for the case $k = 0$.

For $k > 0$, we can prove the formula combinatorially.[12] Consider the path from $(0,0)$ to $(i,i+k)$. The second coordinate starts out equal to the first coordinate, but eventually becomes greater than it by exactly k. For each j from 1 to k, there will be a *first* location in the path where the second coordinate becomes greater than k by exactly j. We can thus partition the path into $k+1$ parts:

- The 1st part starts from the beginning until the point where the second coordinate *first* becomes greater than the first coordinate by 1.
- The 2nd part starts from where it left off until the point where the second coordinate *first* becomes greater than the first coordinate by 2.
- The 3rd part starts from where it left off until the point where the second coordinate *first* becomes greater than the first coordinate by 3.
- ...
- The kth part starts from where it left off until the point where the second coordinate *first* becomes greater than the first coordinate by k.
- The $(k+1)$th part starts from where it left off until the end of the path.

[12] You can also do it algebraically if you want.

Consider some part $j \leq k$. The coordinates start out differing by $j-1$ and ends when the difference becomes j. However, all throughout the path, the difference never becomes j until the end (by the definition of the jth part). Therefore, it is a path from some point $(t, t+j-1)$ to some point $(t', t'+j)$ such that every intermediate point never crosses the "diagonal". But this is precisely what the Catalan number counts!

So, if we denote the sizes of the parts by $2i_1+1, 2i_2+1, \dots, 2i_k+1, 2i_{k+1}$ (it should be easy to see that the first k parts have odd sizes, and the last part even), then the number of possibilities for the jth part, for $j \leq k$, is exactly $c(i_j)$, the i_jth Catalan number. Finally, the last part is unrestricted, so there are exactly $c(i_{k+1}, i_{k+1})$ such paths. We therefore have the following formula:

$$c(i, i+k) = \sum_{i_1+i_2+\ldots+i_{k+1}=i} c(i_1) \cdot c(i_2) \cdot c(i_3) \cdots c(i_k) \cdot c(i_{k+1}, i_{k+1}).$$

Using the "multiply by x^i then sum across i" trick again, we get the result:

$$C_k(x) = \underbrace{C(x) \cdot C(x) \cdot C(x) \cdots C(x)}_{k} \cdot C_0(x) = \frac{C(x)^k}{\sqrt{1-4x}}.$$

□

In particular, the theorem shows that:

$$\begin{aligned} U_L(x) &= \frac{1}{\sqrt{1-4x}} \\ U_R(x) &= \frac{C(x)^{\ell-w}}{\sqrt{1-4x}} \\ C_L(x) &= \frac{xC(x)}{\sqrt{1-4x}} \\ C_R(x) &= \frac{xC(x)^{\ell-w+1}}{\sqrt{1-4x}}. \end{aligned}$$

Therefore, if we substitute these to our formula for $P_R(x)$ and simplify, we will get some complicated formula in terms of the Catalan generating function. But before we do that, let's first derive a nice expression for $1+C_L(x)$, since it is what appears in the formula. We just have to note that $\sqrt{1-4x} = 1-2xC(x)$. Then:

$$\begin{aligned} 1 + C_L(x) &= 1 + \frac{xC(x)}{\sqrt{1-4x}} \\ &= 1 + \frac{xC(x)}{1-2xC(x)} \\ &= \frac{1-xC(x)}{1-2xC(x)} \\ &= \frac{1-xC(x)}{\sqrt{1-4x}} \end{aligned}$$

Then we also note[13] that $1 - xC(x) = \frac{1}{C(x)}$ to obtain:

$$1 + C_L(x) = \frac{1/C(x)}{\sqrt{1-4x}}.$$

We can now proceed with $P_R(x)$:

$$\begin{aligned}
P_R(x) &= \frac{U_R(x)(1+C_L(x)) - U_L(x)C_R(x)}{(1+C_L(x))^2 - x^{\ell-w}C_R(x)^2} \\
&= \frac{\frac{C(x)^{\ell-w}}{\sqrt{1-4x}} \cdot \frac{1/C(x)}{\sqrt{1-4x}} - \frac{1}{\sqrt{1-4x}} \cdot \frac{xC(x)^{\ell-w+1}}{\sqrt{1-4x}}}{\left(\frac{1/C(x)}{\sqrt{1-4x}}\right)^2 - x^{\ell-w}\left(\frac{xC(x)^{\ell-w+1}}{\sqrt{1-4x}}\right)^2} \\
&= \frac{C(x)^{\ell-w} \cdot (1/C(x)) - xC(x)^{\ell-w+1}}{(1/C(x))^2 - x^{\ell-w}\left(xC(x)^{\ell-w+1}\right)^2} \\
&= \frac{C(x)^{\ell-w-1} - xC(x)^{\ell-w+1}}{(C(x))^{-2} - x^{\ell-w+2} \cdot C(x)^{2(\ell-w+1)}} \\
&= \frac{C(x)^{\ell-w-1}(1 - xC(x)^2)}{(C(x))^{-2} - x^{\ell-w+2} \cdot C(x)^{2(\ell-w+1)}} \cdot \frac{C(x)^2}{C(x)^2} \\
&= \frac{C(x)^{\ell-w+1}(1 - xC(x)^2)}{1 - x^{\ell-w+2} \cdot C(x)^{2(\ell-w+2)}}.
\end{aligned}$$

At this point, we recognize (with a bit of algebra) that $1 - xC(x)^2 = C(x)\sqrt{1-4x}$, but, judging from Theorem H.1, we might prefer the $\sqrt{1-4x}$ to be at the denominator, so we'll write it as $\frac{(1-4x)C(x)}{\sqrt{1-4x}}$, to get:

$$P_R(x) = \frac{(1-4x)C(x)^{\ell-w+2}}{\sqrt{1-4x}} \cdot \frac{1}{1 - C(x)^{2(\ell-w+2)} \cdot x^{\ell-w+2}}.$$

At this point comes the weird part: we expand the rightmost fraction using the series

$$\frac{1}{1-t} = \sum_{j \ge 0} t^j.$$

This gives us an expression that, while looks complicated, is actually a sum of terms of the

[13] using the fundamental relation for the Catalan generating function: $C(x) = 1 + xC(x)^2$.

form given in Theorem H.1.

$$\begin{aligned}
P_R(x) &= \frac{(1-4x)C(x)^{\ell-w+2}}{\sqrt{1-4x}} \cdot \sum_{j\ge 0} \left(C(x)^{2(\ell-w+2)} \cdot x^{\ell-w+2}\right)^j \\
&= (1-4x)\sum_{j\ge 0} \frac{C(x)^{(\ell-w+2)(2j+1)}}{\sqrt{1-4x}} \cdot x^{(\ell-w+2)j} \\
&= (1-4x)\sum_{j\ge 0} C_{(\ell-w+2)(2j+1)}(x) \cdot x^{(\ell-w+2)j} \\
&= \sum_{j\ge 0} C_{(\ell-w+2)(2j+1)}(x) \cdot x^{(\ell-w+2)j} \\
&\quad -4\sum_{j\ge 0} C_{(\ell-w+2)(2j+1)}(x) \cdot x^{(\ell-w+2)j+1}.
\end{aligned}$$

We're almost there! At this point, note that the answer we're looking for is the $(w-1)$th term of $P_R(x)$, which we can obtain as the sum of the $(w-1)$th terms of all the summands. But each of the summands is just $C_k(x)$ with some offset power x^u, hence, we can obtain the desired coefficients easily. Using the notation "$[x^u]\mathit{something}$" as the uth coefficient of "$\mathit{something}$", we get the answer as:

$$\begin{aligned}
p_R(w-1) &= [x^{w-1}]P_R(x) \\
&= \sum_{j\ge 0} [x^{w-1}]C_{(\ell-w+2)(2j+1)}(x) \cdot x^{(\ell-w+2)j} \\
&\quad -4\sum_{j\ge 0} [x^{w-1}]C_{(\ell-w+2)(2j+1)}(x) \cdot x^{(\ell-w+2)j+1} \\
&= \sum_{j\ge 0} [x^{w-1-(\ell-w+2)j}]C_{(\ell-w+2)(2j+1)}(x) \\
&\quad -4\sum_{j\ge 0} [x^{w-2-(\ell-w+2)j}]C_{(\ell-w+2)(2j+1)}(x) \\
&= \sum_{j\ge 0} c(w-1-(\ell-w+2)j, \ell+1+(\ell-w+2)j) \\
&\quad -4\sum_{j\ge 0} c(w-2-(\ell-w+2)j, \ell+(\ell-w+2)j)
\end{aligned}$$

This is great, since each $c(i,j)$ is a term that is easy to compute if we precompute factorials! Using this formula, we can compute the answer in $O(\ell+w)$ time![14]

Stating it in terms of binomial coefficients, we get the following equivalent formula:

$$\text{answer} = \sum_{j\ge 0} \binom{\ell+w}{w-1-(\ell-w+2)j} - 4\sum_{j\ge 0} \binom{\ell+w-2}{w-2-(\ell-w+2)j}.$$

Can you show how to use this formula to find a $O(w)$-time solution?

[14]The formula itself is only $O\left(\frac{w}{\ell-w+2}\right)$ terms long. The running time is actually dominated by the factorial precomputation.

H.2 Editorial

This is my favorite problem in this round. It is simple to state, but there are several (interesting) insights needed to get a full solution. In fact, it's so natural to state that, although I came up this problem on my own, I would not be surprised if it has already appeared elsewhere.

Admittedly, this is more effective as a problem when actually joining the contest onsite, without internet access, because otherwise, searching stuff in OEIS would likely give a lot of hints towards the solution.

The FFT approach is probably your best bet if you wanted to solve this problem in a time-limited environment, since the rest of the derivation involves some heavy-duty combinatorics.

Problem I: Glory to Algotzka

Setters: Kevin Atienza

Testers: TBA

Statement Authors: Patrick Celon

Test Data Authors: Kevin Atienza

Editorialists: Kevin Atienza

I.1 Solution

Let us call a node marked C and S as a **C-node** and an **S-node**, respectively.

Note that answering each query in $O(n)$ is too slow, since there are too many queries. We must find a way to preprocess the tree so that the queries can be answered more quickly.

The solution rests on a crucial insight, which we can state as follows:

Theorem I.1. *Fix the node i and the subtree size t, and let $c_{\min}$ and $c_{\max}$ be, respectively, the minimum and maximum number of C-nodes among all subtrees of size t rooted at node i.*

Then any integer c in $[c_{\min}, c_{\max}]$ is obtainable! In other words, for every $c \in [c_{\min}, c_{\max}]$, there is a subtree of size t rooted at i with exactly c C-nodes and $t - c$ S-nodes.

This is very convenient, since if we can precompute all $c_{\min}$ and $c_{\max}$ for every pair (i,t), then every query can then be answered in $O(1)$!

The proof is quite elegant.

Proof. Let $T_{\min}$ and $T_{\max}$ be trees with exactly $c_{\min}$ and $c_{\max}$ C-nodes (respectively) of size t rooted at node i.

Note that we can "transform" $T_{\min}$ into $T_{\max}$ by changing one node at a time, since they are both trees rooted at node i. For example, starting at $T := T_{\min}$, we can just repeatedly replace a leaf of T not contained in $T_{\max}$ with a leaf in $T_{\max}$ not contained in the current T. The mentioned leaves should always exist, otherwise $T_{\max}$ would be a subset of T (or vice versa), which is impossible if they have the same size and are unequal.

However, note that if we replace nodes one by one this way, then the number of C-nodes can only change by -1, 0 or $+1$. Furthermore, we start at $c_{\min}$ and end at $c_{\max}$. Therefore, every integer c along the way, from $c_{\min}$ to $c_{\max}$, is encountered at least once! $\square$

I.1.1 Computing the extremes

We would now like to compute $c_{\min}(i,t)$ and $c_{\max}(i,t)$ for every node i and every $t \in [0, size(i)]$. Note that $c_{\max}(i,t) = t - s_{\min}(i,t)$, so computing $c_{\max}$ is easily reduced to computing $s_{\min}$, which can be computed analogously to $c_{\min}$. Therefore, we are really only looking for one kind of procedure, and we can focus on $c_{\min}(i,t)$ without loss of generality.

A straightforward solution would be to compute $c_{\min}(i,t)$ via DP, since $c_{\min}(i,t)$ depends on the values of $c_{\min}$ for the children nodes of i. For example, suppose i had two children i_1 and i_2. Then we have $c_{\min}(i,0) = 0$, and for $t > 0$, we have

$$c_{\min}(i,t) = \min_{\substack{0 \le t_1 \le size(i_1) \\ 0 \le t_2 \le size(i_2) \\ t_1 + t_2 + 1 = t}} \left(c_{\min}(i_1,t_1) + c_{\min}(i_2,t_2) + [i \text{ is a C-node}]\right),$$

or, in terms of just one iterator, say t_1 (and using $t_2 = t - 1 - t_1$),

$$c_{\min}(i,t) = \min_{\substack{0 \le u \le size(i_1) \\ 0 \le t-1-u \le size(i_2)}} \left(c_{\min}(i_1,u) + c_{\min}(i_2,t-1-u) + [i \text{ is a C-node}]\right).$$

In general, for a node i with $r(i)$ subtrees $i_1,\dots,i_{r(i)}$, let $c_{\min}(i,t,k)$ be value of $c_{\min}$ while only considering the first k subtrees of i. Then you can easily find a recurrence for $c_{\min}(i,t,k)$ similar to the above, e.g.,

$$c_{\min}(i,t,k) = \min_{\substack{0 \le u \le size(i_k) \\ 1 \le t-u \le size(i,k-1)}} \left(c_{\min}(i_k,u) + c_{\min}(i,t-u,k-1)\right)$$

where $size(i,k)$ is the size of the subtree rooted at i while only considering the first k children. This also has a simple recurrence:

$$size(i,k) = size(i,k-1) + size(i_k).$$

And we have the base cases:

$$\begin{aligned} c_{\min}(i,0,k) &= 0 \\ c_{\min}(i,1,0) &= [i \text{ is a C-node}] \\ size(i,0) &= 1. \end{aligned}$$

We can now compute $size(i)$ and $c_{\min}(i,t)$ as simply $size(i,r(i))$ and $c_{\min}(i,t,r(i))$, respectively. After doing this (and the same thing for $s_{\min}$), we can now answer a query (i,c,s): the answer is `COMPROMISED` if and only if all of the following are true:

$$\begin{aligned} c + s &\le size(i) \\ c &\ge c_{\min}(i,c+s) \\ s &\ge s_{\min}(i,c+s). \end{aligned}$$

Thus, each query can be answered in $O(1)$! But how about the precomputation step? Well, there are up to $O(n^2)$ triples (i,t,k), and for each such triple, $c_{\min}(i,t,k)$ is computed with a loop of size $O(n)$. Thus, the complexity is $O(n^3)$, which doesn't seem enough for $n = 10000$.

But if you look very very carefully, this DP is actually really $O(n^2)$! The proof is not trivial. I usually call this DP pattern the "tree convolution DP", and a proof can be found in this document I wrote.[15] The gist is that we can show the following:

[15] `https://drive.google.com/open?id=1nhL63QcjUiRm1pGGmzi1QHceKAGeBsRY`

Theorem I.2. *For a fixed i, the complexity of computing $c_{\min}(i,t,k)$ for all $0 \le k \le r(i)$ and $0 \le t \le size(i,k)$, assuming the $c_{\min}$ values have been computed for i's children, is*

$$O\left(size(i)^2 - \sum_{j=1}^{k} size(i_j)^2\right).$$

And so, if we sum up all these complexities, we get some telescoping sum that results in

$$O\left(size(root)^2 + \sum_{i=1}^{n} size(i)\right) = O(n^2).$$

So the preprocessing step actually runs in $O(n^2)$, which is enough to pass the time limit! The overall solution is $O(n^2+q)$.

I.2 Editorial

Have you played “Papers, Please” yet? You should!

Problem J: A Cold Macchiato

Setters: Josh Quinto, JD Dantes

Testers: Patrick Celon

Statement Authors: JD Dantes

Test Data Authors: Kevin Atienza

Editorialists: Kevin Atienza

J.1 Solution

J.1.1 The Probability

Let m_i be the probability that the ith dispenser malfunctions, and $d_{i,j}$ the probability that the ith dispenser releases water at the temperature of the jth dispenser if it malfunctions.

Then there are $3 \cdot 3 = 9$ possibilities in total. We can label these possibilities as (i, j) meaning the ith dispenser malfunctions *and* releases at the temperature of the jth dispenser. The probability that this happens is clearly $m_i \cdot d_{i,j}$, and you can verify that the sum of the probabilities across all 9 possibilities is 1.

A *strategy* is a selection (s_1, s_2, s_3) of amounts to obtain from each dispenser. Note that s_i must be ≥ 0.

Note that the statement requires $s_1 + s_2 + s_3$ to be exactly 1000 milliliters, but actually, we can set it to anything we want, since you can easily verify that only the *proportions* matter, i.e., (s_1, s_2, s_3) has the same "probability of success" as $(\alpha s_1, \alpha s_2, \alpha s_3)$, since these values are only used as weights for the average. Therefore, instead of requiring the sum to be 1000 milliliters, it should be more convenient to set it at $s_1 + s_2 + s_3 = 1$. The answer will still be the same.

What is the "probability of success" of strategy (s_1, s_2, s_3)? Well, we know that the dispensers give out water in temperatures (t_1, t_2, t_3), but that's if they are functioning correctly. Let (t'_1, t'_2, t'_3) be the *actual* temperatures they give out. These t'_i values are easily computed for each of the 9 possibilities. Then the temperature of the concoction will be:

$$\frac{s_1 t'_1 + s_2 t'_2 + s_3 t'_3}{s_1 + s_2 + s_3}.$$

but note that we conveniently set $s_1 + s_2 + s_3 = 1$, so this is just

$$s_1 t'_1 + s_2 t'_2 + s_3 t'_3.$$

We can now compute the "probability of success" of strategy (s_1, s_2, s_3). It is simply the sum of the probabilities ($m_i \cdot d_{i,j}$) across all the (i, j) such that this average temperature is in the range $[\ell, u]$. Formally, it is:

$$\sum_{\substack{1 \leq i,j \leq 3 \\ \ell \leq s_1 t'_1 + s_2 t'_2 + s_3 t'_3 \leq u}} m_i d_{i,j}.$$

Thus, for any given strategy (s_1, s_2, s_3), this is easy (and quick) to compute.

The answer is then the maximum of this sum across all valid strategies (s_1, s_2, s_3).

J.1.2 The Candidates

Unfortunately, there are an infinite number of candidates, since the s_is can be any (nonnegative) real number. We must find a way to limit the number of candidates to check, while ensuring that the optimal strategy is always included in our candidates.

First, note that $s_3 = 1 - s_1 - s_2$, so we only actually have two degrees of freedom: selecting s_1 and s_2. Thus, we can represent a strategy as a point (s_1, s_2) in the 2D plane. The only valid strategies are in the region bounded by the following inequalities:

$$\begin{aligned} s_1 &\geq 0 \\ s_2 &\geq 0 \\ s_1 + s_2 &\leq 1 \end{aligned}$$

(The last one is equivalent $s_3 \geq 0$.)

In other words, the valid region is the triangle with vertices $(0,0)$, $(0,1)$, $(1,0)$.

Also, note that the inequality

$$s_1 t'_1 + s_2 t'_2 + s_3 t'_3 \leq u$$

becomes

$$\begin{aligned} s_1 t'_1 + s_2 t'_2 + (1 - s_1 - s_2) t'_3 &\leq u \\ s_1 (t'_1 - t'_3) + s_2 (t'_2 - t'_3) &\leq u - t'_3. \end{aligned}$$

Similarly,

$$s_1 t'_1 + s_2 t'_2 + s_3 t'_3 \geq \ell$$

becomes

$$s_1 (t'_1 - t'_3) + s_2 (t'_2 - t'_3) \geq \ell - t'_3.$$

But now, note that all our constraints are of the form $as_1 + bs_2 \leq c$.[16] Furthermore, the set of points (s_1, s_2) satisfying $as_1 + bs_2 \leq c$ is a *half-plane*! The boundary of this half-plane is the line $as_1 + bs_2 = c$. Thus, for a given candidate (s_1, s_2), the "probability of success" is determined by the set of half-planes containing the point (s_1, s_2).

But if we collect all boundaries of the half-planes, then the plane will be decomposed into a bunch of *convex* regions, whose boundaries are line-segment subsets of the half-plane boundaries. In each of these regions, the "probability of success" for all points there will be the same, since the probability can only change if an inequality $as_1 + bs_2 \leq c$ becomes true or false (or vice versa), and that can only happen if we pass through a half-plane boundary.

This also means, in particular, that the optimal solution lies in some convex region. But since the probability of success is the same for all points in this region, this means that *any* point

[16] The form "$x \geq y$" is equivalent to "$-x \leq -y$".

in this convex region is also an optimal solution. This means that we can just take an easy-to-identify point in each region as our candidate. For convex regions, the obvious candidates are simply the *vertices* of the region. But vertices of regions are simply intersections of two boundary lines. Therefore, we can limit our candidates to those points, because the number of intersections is finite!

In summary, we have shown that there is an optimal solution that lies in the intersection of two boundary lines.[17]

So the solution is simply this:

1. Collect all the half-plane boundaries $as_1 + bs_2 \leq c$ (including $s_1 \geq 0$, $s_2 \geq 0$ and $s_1 + s_2 \leq 1$).
2. Collect all intersection points across every pair of boundaries (that are not parallel).
3. Remove the intersection points that are outside the "triangle of valid strategies".
4. The optimal solution lies in one of these intersection points.

The running time of this solution depends on the number of candidates to test, which we can bound as follows:

- For each of the $9 \cdot (9-1)/2$ pairs of possibilities, there are at most 4 candidates, since each possibility gives rise to two parallel half-plane boundaries, and so there may be $2 \cdot 2$ intersections among them. Overall, these give rise to up to $9 \cdot (9-1)/2 \cdot 4 = 144$ candidates.
- For each of the 9 possibilities, there are at most $3 \cdot 3$ more candidates, since each boundary only intersects the boundaries of the "triangle of valid strategies" at most 3 times. These give rise to up to $9 \cdot 3 \cdot 3 = 81$ candidates.
- There are 3 more candidates corresponding to the vertices of the triangle of valid strategies: $(0,0)$, $(1,0)$ and $(0,1)$.

Thus, overall, there are at most $144 + 81 + 3 = 228$ candidates! Each candidate can then be checked quickly, so the overall solution runs quickly.

As an aside, although 228 is already low enough to make us feel confident that the solution will pass, we can nonetheless still improve this bound by being more careful with our analysis:

- Three of the "9 possibilities" are actually the same, and they correspond to the ith dispenser malfunctioning into the ith dispenser, i.e., not actually malfunctioning! So we can replace 9 with 7.
- For each of the 7 possibilities, there are actually just $3 \cdot 2$ more candidates, not $3 \cdot 3$, since each boundary only intersects the triangle of valid strategies at most 2 times. The third point is outside the triangle and thus may be omitted.

[17]You may wonder if this is actually correct, since the boundary itself only belongs to the region in one of the sides of that boundary. But thankfully, the probabilities are nonnegative, and the boundary is always on the side where the inequality is true (since the inequality is not strict: "$\leq$"), so the boundary point is included in the "best" region.

These give a better bound of $7 \cdot 6/2 \cdot 4 + 7 \cdot 3 \cdot 2 + 3 = 147$ candidates!

We can quantify the running time in another way. Suppose there are n dispensers. Then there are $O(n^4)$ candidates to test, and each one can be tested in $O(n^2)$ time. Therefore, the running time is $O(n^6)$. Thankfully, in our case, $n = 3$, so this is very fast. But for bigger n, can you find a faster solution than $O(n^6)$?

Note: You should not use floats here, since the answer could belong to a convex region of area 0, so any imperfection will make the answer incorrect. In particular, if the coordinates of the strategy don't have terminating binary expansions, then the floats couldn't even represent them exactly!

J.2 Editorial

This problem was inspired by a broken water dispenser at Josh's office. We just came up with a probability problem around the premise of a "broken water dispenser".

Problem K: I Brook the Code!

Setters: Payton Yao

Testers: Kevin Atienza, Anton Rufino

Statement Authors: JD Dantes

Test Data Authors: Josh Quinto

Editorialists: JD Dantes

K.1 Solution

In this problem, you are asked to sort pairs of numbers. You could try to use simpler approaches like Bubble or Selection Sort. However, these run in $O(n^2)$ time and would result to TLE as the size of the input can be up to 10^5. As a rule of thumb, your solution's running time should be in the order of 10^8 steps; thus $O(n^2)$ sorts would pass if the size of the list to sort is in the order of 10^4.

For the given bound of 10^5, an $O(n \log n)$ sorting approach would be fine, such as Quicksort. You could implement such sorts yourself, but programming languages usually have such functionalities built-in already. For C/C++, there's '`qsort()`' or '`sort()`'. Python has '`sorted()`' or '`sort()`' which you can use on lists.

Sorting algorithms are typically demonstrated with single integers, not pairs. To extend it to the multiple number case, we just have to define when one pair is "less than" another pair. We can do so by creating *comparators*.[18] Alternatively, we note that some useful classes, such as '`pair`' or '`tuple`' in C++ or '`tuple`' in Python, conveniently evaluate comparisons by starting from the first elements and moving towards the last, and so we can use these directly when using '`sort()`'. For this problem, the heights are guaranteed to be distinct and sorting of the weight does not really matter, but otherwise the default comparators would probably be useful.

There also exist some data structures that are already internally sorted, such as C++'s '`map`' and '`set`' classes. Items are sorted as you insert, and you can then just traverse the entire '`map`' or '`set`' to print the items in the desired order.

If you'd like to learn more about the time complexities of other common data structures and sorting algorithms, check out Big-O Cheat Sheet.

[18]see C++ examples here: `http://www.cplusplus.com/reference/algorithm/sort/?kw=sort`

Problem L: Break the Pattern!

Setters: JD Dantes

Testers: TBA

Statement Authors: JD Dantes

Test Data Authors: JD Dantes

Editorialists: JD Dantes, Kevin Atienza

L.1 Solution

Given the ℓ integers $s_1, s_2, \ldots, s_\ell$, we want to find a polynomial where each s_i is a root.

Note that the polynomial $(x-s_1)(x-s_2)\ldots(x-s_\ell)$ will fit this description, assuming that it is within the maximum degree k. Furthermore, any multiple of this polynomial also fits the description, as long as the degree is $\leq k$.

We can then multiply any arbitrary term $(x-a)$ and the description will still hold (again, if the maximum degree allows for it). Alternatively, we can also just multiply a constant to produce a different polynomial. In fact, using constants, we can product at least 999983 distinct polynomials just by multiplying distinct nonzero constants, more than enough for the given bounds.

Also, amazingly, it turns out that this is the only way to produce valid polynomials. In other words:

Theorem L.1. *Let $s_1, s_2, \ldots, s_\ell$ be distinct values. Then any polynomial with all s_is as roots is a multiple of $(x-s_1)(x-s_2)\ldots(x-s_\ell)$.*

Note that it is crucial in the theorem that the s_is are distinct.

The task is then to multiply and produce the polynomials' expanded forms to get the desired coefficients. As the bounds are rather small, we could do this in a straightforward manner, such as representing the polynomial coefficients as vectors/lists and then repeatedly multiplying each $(x-s_i)$ until we get the final polynomial. Note that you also have to reduce the coefficients modulo 999983, otherwise, your coefficients will likely exceed the 10^9 bound on the coefficients, or even overflow.

Finally, note that there could be repeating roots in the sequence, possibly causing you to exceed the maximum degree k if unhandled. At the start, we should remove the duplicate numbers in the sequence as it suffices to use just one $(x-s_i)$ to satisfy the multiple occurrences of each s_i in the sequence.

Here's a proof of the theorem:

Proof. Let $p(x) = (x-s_1)(x-s_2)\ldots(x-s_\ell)$, and let $a(x)$ be any polynomial having all s_is as roots. We will show that $a(x)$ is a multiple of $p(x)$. Let $q(x)$ and $r(x)$ be the quotient and remainder, respectively, when we divide $a(x)$ by $p(x)$. In other words,

$$a(x) = p(x)q(x) + r(x).$$

Note that either $r(x) = 0$ or the degree of r is strictly less than the degree of p, which is ℓ.

But note that for each i, $r(s_i) = a(s_i) - p(s_i)q(s_i) = 0 - 0q(s_i) = 0$. Therefore, r is a polynomial with degree less than ℓ with at least ℓ distinct roots. But note that a nonzero polynomial with degree d has at most d roots. Therefore, the only possibility is that $r(x)$ is the zero polynomial. Therefore, $a(x) = p(x)q(x)$, i.e., $a(x)$ is a multiple of $p(x)$, as desired. □

Of course, this assumes that the statement "every nonzero polynomial with degree d has at most d roots" is true even in our case, where we consider 999983 to be equivalent to 0. This is true if and only if the number we chose is prime (and luckily, 999983 is prime). This also needs proof.

We use the notation $\mathbb{Z}/n\mathbb{Z}$ to denote the set of integers modulo n.

Theorem L.2. *Every nonzero polynomial $p(x)$ with coefficients in $\mathbb{Z}/n\mathbb{Z}$ have at most $\deg p$ roots if and only if n is prime.*

Proof. The forward direction is easy; we prove the converse, and assume n is composite and can be factorized nontrivially as $n = ab$. Then consider the polynomial ax. This degree-1 polynomial has at least 2 roots, 0 and b, which is what we wanted.

For the backward direction, assume n is prime. A key property of primes we will use is that if ab is divisible by (the prime) n, then either a is divisible by n, or b is divisible by n.

Let a_1 be a root of polynomial p. Let's divide $p(x)$ by $x - a_1$ using long division. We will get $p(x) = q(x)(x - a_1) + r(x)$, where $q(x)$ is some polynomial of degree $\deg p - 1$, and $r(x)$ is a polynomial of degree < 1, i.e., a constant. Then $r(a_1) = p(a_1) - q(a_1)(a_1 - a_1) = 0 - q(a_1) \cdot 0 = 0$, so $r(x)$ must in fact be the zero polynomial. Therefore, $p(x) = q(x)(x - a_1)$.

We can repeat this process on q, until we end up with some polynomial without any roots, or we end up factorizing p completely into linear factors, i.e., $p(x) = (x - a_1)(x - a_2) \cdots (x - a_k)q(x)$, where q is a nonzero polynomial such that $\deg q = \deg p - k$ without any roots in $\mathbb{Z}/n\mathbb{Z}$. Note that this also implies that the number of linear factors, k, is at most $\deg p$, since the degree of a nonzero polynomial is nonnegative.

Now, suppose a is a root of p. Then $0 = p(a) = (a - a_1) \cdot (a - a_2) \cdots (a - a_k) \cdot q(a)$, i.e., the latter product is divisible by n. But remember our key property of primes? It implies that one of $(a - a_1), (a - a_2), \ldots, (a - a_k), q(a)$ must be divisible by n. $q(a)$ is not divisible by n since q doesn't have any roots. Therefore, it must be the case that some $(a - a_i)$ must be divisible by n. But in this case, a must be equivalent to a_i. Therefore, we have shown that any root a of p must be one of $\{a_1, \ldots, a_k\}$, and thus, p has $k \leq \deg p$ roots. □

Problem M: Thin Ice

Setters: Kevin Atienza, Payton Yao

Testers: TBA

Statement Authors: Pio Fortuno III

Test Data Authors: Patrick Celon

Editorialists: Patrick Celon

M.1 Solution

The problem can be summarized into a single statement: *given a grid graph and a starting position s, find a path that travels through all cells*! As straightforward as this looks though, there are some things that we have to take note of. One is that the value $t \cdot r \cdot c$, at maximum, is much much larger than the 10 megabyte output constraint; there might be test files that have a lot of impossible scenarios. Therefore, we have to determine a way to find whether a given test case is possible quickly, say in constant time.

We can simplify the problem by noticing that *isomorphic cases are the same*; we just need to rotate them to match the path. From there, we can then start eliminating edge cases.

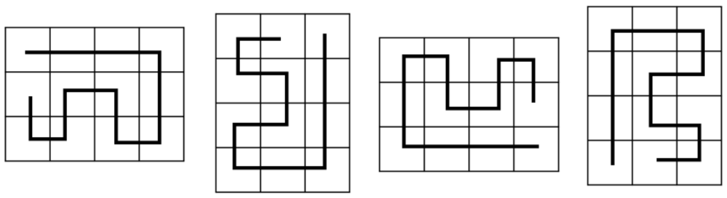
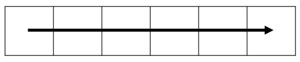
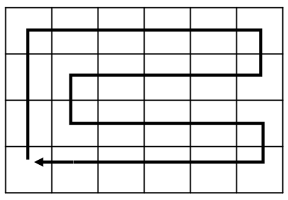


Figure 1: A solution string can be the same path, but rotated 90, 180, or 270 degrees.

Start with the easiest edge case: a $1 \times n$ grid. A path can only go in one direction, therefore it is impossible if you are tasked to start at the middle. See Figure 2a. Another insight to the problem lies in the fact that *a path is always possible if rc is even*. (We'll call a grid where rc is even an *even* grid, and *odd* otherwise.) This is because we can construct a cycle that goes through all cells for any type of even grid. See Figure 2b.



(a) $1 \times n$ edge case



(b) A generic solution for any even grid

Figure 2

In Figure 2b, the path zig-zags all the way to the bottom to return to its starting point. Therefore, to create a path from any point, we just have to create the cycle, and *offset the starting*

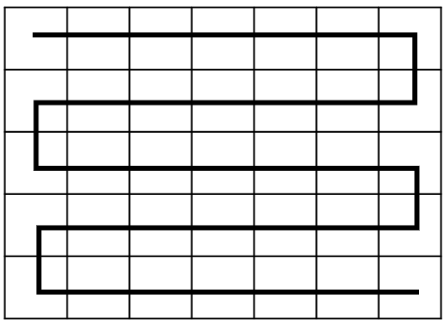


Figure 3: For odd grids, a zig-zag traversal is only valid from corners as it can't loop back to the start.

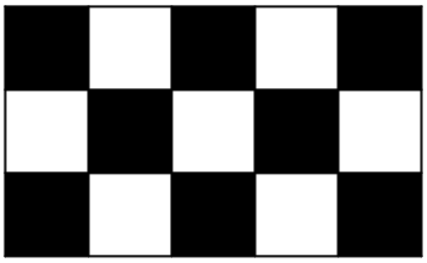


Figure 4: A grid can be represented by a checkerboard (alternating colors for adjacent cells)

point to the given point. Note that this is only possible if the number of rows in the zig-zag are even. Otherwise, the zig-zag attempt will look like Figure 3, and we can't loop back at the start. Thus, it is enough that one of the sides are even, as we can always rotate it such that it is similar to the one in Figure 2b.

The remaining cases are of odd grids (both sides are odd), and it turns out that it isn't always possible. We can determine if it is possible to create a path using the following properties:

- A grid can be colored *black* and *white* alternately (checkerboard).
- Any cell movement results in a change of color.
- The number of black cells *(majority)* $\neq$ the white cells *(minority)*.

Since $rc - 1$ is even (from rc being odd), the final destination cell should be the same color as the starting cell. If this starting cell is of the minority color, there are simply not enough cells left for it to traverse to — it needs $\frac{rc-1}{2}$ cells to transfer to, but there are only $\frac{rc-1}{2} - 1$ cells to transfer to as it already started on a cell of the same color. Thus, this case is impossible.

This property is called *color-incompatibility*. In other words, a grid is called *color-compatible* if and only if the starting cell rests on a *majority square*. If a grid is color-incompatible, then the above argument shows that there is no path. Amazingly, it turns out that if it is color-compatible, then a path exists!

We can now check to see whether a solution exists or not using these conditions:

```
// start_row and end_row are 1−indexed
def is_solvable(row, col, start_row, start_col):
   if row == 1 and (start_col != 1 and start_col != col) or
      col == 1 and (start_row != 1 and start_row != row):
      // 1xn / nx1 edge case
      return false
   elif row*col % 2 == 0:
      // even case
      return true
   else:
      // odd case
      // Check if the starting point falls into a majority square
      return (start_row + start_col) % 2 == 0
```

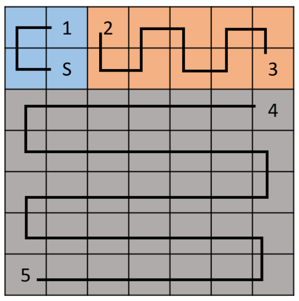
Let's now describe a solution for the color-compatible case.

To reduce the number of cases to solve, we can define a rotation function that rotates a given traversal string:

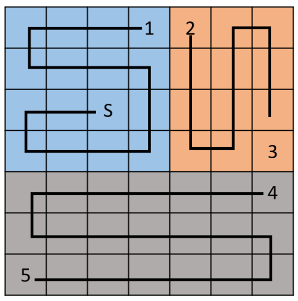
```
// start_row and end_row are 1−indexed
dir = "URDL"  // the directions while rotating clockwise

// Rotate a traversal string 'count' times to the right
def rotate_right(traversal, count):
   return string(dir[(dir.index(c) + count)%4] for c in
      traversal)
```

Thus, if needed, we can rotate a given grid into an isomorphic form that we can easily find a solution for, or at least into a form a bit more convenient for us.

One solution (out of many) would be to rotate it such that it is in the upper-left quadrant, and create an even grid (with even rows and columns) with the starting point near the bottom-right position. Then there would be three regions of interest: the even grid, the grid to the right, and the odd grid below. Then there are two cases, depending on whether the `start_row/col` is odd or not. This is shown in Figure 5.



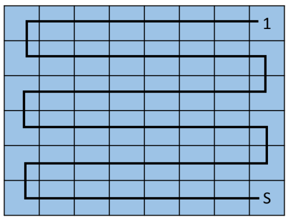
(a) when `start_row/col` is even



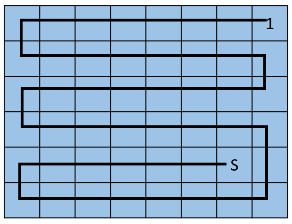
(b) when `start_row/col` is odd

Figure 5: Odd grid cases

In general, we can solve the odd-grid case by going from $S \to 1 \to 2 \to 3 \to 4 \to 5$. The paths $2 \to 3$ and $4 \to 5$ are similar to Figure 3 and are trivial. Thus we are left with how to make a path from the starting point S to point 1. If `start_row` is even, then we can do something similar to Figure 6a to end up at 1. If `start_row` is odd, then we can loop back under and behind S, then continue to do the same zigzag motion upwards (see Figure 6b).



(a) when `start_row/col` is even



(b) when `start_row/col` is odd

Figure 6: Traversal on the upper-left grid

Finally, we can define the full solution to be the combination of the solutions for the three cases:

1. The grid is a $1 \times n$ or $n \times 1$ edge case.
2. The grid is an even grid.
3. The grid is an odd grid.

For 1 and 2, we can use the solutions provided by Figure 2a and Figure 2b respectively. For 3, we use the solutions Figure 5a for starting positions at even cells (both row/col is even), and Figure 5b for starting positions at odd cells.

M.2 Editorial

A generalized version of the problem can be read about here: `https://doi.org/10.1155/2012/475087`.