Problem Set Developers

- Bryce Sandlund (NCNA Chief Judge)
- Antonio Molina
- Finn Lidbetter
- Anindya Das
- Nalin Bhardwaj
- Tomas Rokicki
- Marc Furon (SoCal)
- Ed Skochinski (SoCal)
- Ronqi Qiu (SoCal)
- Bob Logan (SoCal)
- Nalin Bhardwaj
- Yinzhan Xu
- Pasha Kazatsker
E - Curve Speed

Problem

Given $R$ and $S$, evaluate the equation:

$$V = \sqrt{(R \times (S + .16)))/.067.}$$
Problem

Given \( R \) and \( S \), evaluate the equation:

\[
V = \sqrt{\frac{R \times (S + 0.16))}{0.067}}.
\]

Solution

Straightforward.
Problem

Given $R$ and $S$, evaluate the equation:

$$V = \sqrt{(R \times (S + .16))/.067}.$$  

Solution

Straightforward.

Pitfalls

- Not rounding properly.
Problem

Given $R$ and $S$, evaluate the equation:

$$V = \sqrt{(R \times (S + .16))/.067}.$$ 

Solution

Straightforward.

Pitfalls

- Not rounding properly.
- Not reading to end-of-input properly.
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    while (scan.hasNext()) {
        double R = scan.nextDouble();
        double S = scan.nextDouble();
        double V = Math.sqrt(R * (S + 0.16) / 0.067);
        long ans = Math.round(V);
        System.out.println(ans);
    }
}
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    while (scan.hasNext()) {
        double R = scan.nextDouble();
        double S = scan.nextDouble();
        double V = Math.sqrt(R * (S + 0.16) / 0.067);
        long ans = Math.round(V);
        System.out.println(ans);
    }
}
Problem
Given two lists of names and emails, determine which records do not match on either first name and last name or email with any record in the other list.
Problem

Given two lists of names and emails, determine which records do not match on either first name and last name or email with any record in the other list.

Solution

Straightforward.
Problem
Given two lists of names and emails, determine which records do not match on either first name and last name or email with any record in the other list.

Solution
Straightforward.

Pitfalls
- Missing matches.
Problem
Given two lists of names and emails, determine which records do not match on either first name and last name or email with any record in the other list.

Solution
Straightforward.

Pitfalls
- Missing matches.
- TLE on some $O(n^2)$ solutions (sorry).
Problem
Given two lists of names and emails, determine which records do not match on either first name and last name or email with any record in the other list.

Solution
Straightforward.

Pitfalls
- Missing matches.
- TLE on some $O(n^2)$ solutions (sorry).

Statistics: 248 submissions, 27 accepted.
Problem
Given a list of speeds, round them according to the given rules.

Solution
Straightforward.

Pitfalls
Finding the "most recent preceding value for $s$ outside of range $[i + t_f, i + t_r]$" is prone to bugs. You are better off iterating backwards to ensure you do it correctly. This is worst-case $O(n^2)$, though with the input bounds this should be fine. I also didn’t create any cases to force $O(n^2)$ runtime on look-back solutions (oops).

Confusion on inclusivity of "falls between" (it doesn’t matter since $t_f$ and $t_r$ are of the form $0.x5$, where $x \in \{0, \ldots, 9\}$, and speed is given to the first decimal place).
Problem
Given a list of speeds, round them according to the given rules.

Solution
Straightforward.

Pitfalls
Finding the "most recent preceding value for \( s \) outside of range \( \lfloor i + t \rfloor \)" is prone to bugs. You are better off iterating backwards to ensure you do it correctly. This is worst-case \( O(n^2) \), though with the input bounds this should be fine. I also didn't create any cases to force \( O(n^2) \) runtime on look-back solutions (oops).

Confusion on inclusivity of "falls between" (it doesn't matter since \( t \) and \( t' \) are of the form 0.x5, where \( x \in \{0, \ldots, 9\} \), and speed is given to the first decimal place.)

Problem Author: Ed Skochinski
Problem
Given a list of speeds, round them according to the given rules.

Solution
Straightforward.

Pitfalls
- Finding the “most recent preceding value for s outside of range \([i + t_f, i + t_r]\)” is prone to bugs. You are better off iterating backwards to ensure you do it correctly. This is worst-case \(O(n^2)\), though with the input bounds this should be fine. I also didn’t create any cases to force \(O(n^2)\) runtime on look-back solutions (oops).
Problem
Given a list of speeds, round them according to the given rules.

Solution
Straightforward.

Pitfalls
- Finding the “most recent preceding value for $s$ outside of range $[i + t_f, i + t_r]$” is prone to bugs. You are better off iterating backwards to ensure you do it correctly. This is worst-case $O(n^2)$, though with the input bounds this should be fine. I also didn’t create any cases to force $O(n^2)$ runtime on look-back solutions (oops).
- Confusion on inclusivity of “falls between” (it doesn’t matter since $t_f$ and $t_r$ are of the form $0.x5$, where $x \in \{0, \ldots, 9\}$, and speed is given to the first decimal place.).
Pitfalls, cont.

- It is possible $t_f > 0.5$ or $t_r < 0.5$, so avoid library rounding.
Pitfalls, cont.

- It is possible $t_f > 0.5$ or $t_r < 0.5$, so avoid library rounding.

Statistics: 339 submissions, 40 accepted.
Problem

For each string in input, determine the number of unique proper contiguous substrings that have the same set of characters as the input string and contain no proper substrings also containing the same set of characters as the input string.
Problem
For each string in input, determine the number of unique proper contiguous substrings that have the same set of characters as the input string and contain no proper substrings also containing the same set of characters as the input string.

Solution
One straightforward solution:

1. Given a starting index $i$, find the smallest $j$ such that the substring from $i$ to $j$ has all unique characters; call it $ending(i)$. 
Problem
For each string in input, determine the number of unique proper contiguous substrings that have the same set of characters as the input string and contain no proper substrings also containing the same set of characters as the input string.

Solution
One straightforward solution:

1. Given a starting index \( i \), find the smallest \( j \) such that the substring from \( i \) to \( j \) has all unique characters; call it \( \text{ending}(i) \).
2. Add into a set/hashset all proper substrings \([i, j]\) where \( \text{ending}(i + 1) \neq \text{ending}(i) \).
Problem

For each string in input, determine the number of unique proper contiguous substrings that have the same set of characters as the input string and contain no proper substrings also containing the same set of characters as the input string.

Solution

One straightforward solution:

1. Given a starting index \( i \), find the smallest \( j \) such that the substring from \( i \) to \( j \) has all unique characters; call it ending(\( i \)).
2. Add into a set/hashset all proper substrings \([i, j]\) where ending(\( i + 1 \)) \( \neq \) ending(\( i \)).
3. Report the number of strings in the set.
Pitfalls

- I/O. Again. Many submissions did not pass sample data.
D - Substring Characters

Pitfalls

- I/O. Again. Many submissions did not pass sample data.

Statistics: 105 submissions, 36 accepted.
Problem

Given a set of “facts”, resembling function calls, and a set of queries, determine how many facts are matched by each query, following the stated rules for matching.
Problem
Given a set of “facts”, resembling function calls, and a set of queries, determine how many facts are matched by each query, following the stated rules for matching.

Solution
Given a query, check each fact to determine if it matches.
Problem

Given a set of “facts”, resembling function calls, and a set of queries, determine how many facts are matched by each query, following the stated rules for matching.

Solution

Given a query, check each fact to determine if it matches.

Pitfalls

Improper handling of spaces between tokens. Testcase handmade2.in:

\[
\text{fact}(a,b,c) \\
\text{fact} ( \ a \ , \ b,c) \\
\text{fact}(\_,\_,\_) \ // \ should \ be \ 2
\]
Inproper logic. Testcase `handmade1.in`:

- `test(arg1, arg1)`
- `test(arg1, arg2)`
- `test(__arg2, __arg2) // 1`
- `test(__arg1, __arg2) // 2`
- `test2(_, __) // 0`
- `test(_, _) // 2`
- `test(_, _) // 2`
Pitfalls

Inproper logic. Testcase handmade1.in:

test(arg1, arg1)
test(arg1, arg2)

test(__arg2, __arg2) // 1
test(__arg1, __arg2) // 2
test2(_, __) // 0
test(_, __) // 2
test(_, __) // 2

Statistics: 96 submissions, 24 accepted.
Problem

Given specifications for a set of points on a 2D plane representing shoelace holes, determine the number of valid symmetric shoelace patterns that result in shoelace lengths between a lower bound $f_{\text{min}}$ and upper bound $f_{\text{max}}$. 

Problem Author: Marc Furon
Problem

Given specifications for a set of points on a 2D plane representing shoelace holes, determine the number of valid symmetric shoelace patterns that result in shoelace lengths between a lower bound \( f_{\text{min}} \) and upper bound \( f_{\text{max}} \).

Solution

- The first task is to get an upper bound on the number of possible valid lacing patterns.
Since hole 0 must go to 1 or 1 to 0, and we must start and end at $2N-2$ and $2N-1$, we can figure out the pattern from $2N-2$ to 0 or 1, then the rest of the holes are a reflection.
Since hole 0 must go to 1 or 1 to 0, and we must start and end at $2N - 2$ and $2N - 1$, we can figure out the pattern from $2N - 2$ to 0 or 1, then the rest of the holes are a reflection.

We can only choose one of $\{2, 3\}, \{4, 5\}, \ldots, \{2N - 4, 2N - 3\}$ on the way from $2N - 2$ to 0 or 1 so that we may complete the reflection.
Solution

A loose upper bound can be determined as follows. Consider the pattern between $2N - 2$ and whichever of 0 or 1 the lace goes through first.

- Of each pair $\{0, 1\}, \{2, 3\}, \ldots, \{2N - 4, 2N - 3\}$ the lace goes through either the left hole, the right hole, or neither hole (for $\{0, 1\}$ it must go through either the left or right hole).
Solution

A loose upper bound can be determined as follows. Consider the pattern between \(2N - 2\) and whichever of 0 or 1 the lace goes through first.

- Of each pair \(\{0, 1\}, \{2, 3\}, \ldots, \{2N - 4, 2N - 3\}\) the lace goes through either the left hole, the right hole, or neither hole (for \(\{0, 1\}\) it must go through either the left or right hole).

- We can bound the number of possible orders of pairs the lace travels through at \((N - 2)!\).
Solution

A loose upper bound can be determined as follows. Consider the pattern between \(2N - 2\) and whichever of 0 or 1 the lace goes through first.

- Of each pair \(\{0, 1\}, \{2, 3\}, \ldots, \{2N - 4, 2N - 3\}\) the lace goes through either the left hole, the right hole, or neither hole (for \(\{0, 1\}\) it must go through either the left or right hole).
- We can bound the number of possible orders of pairs the lace travels through at \((N - 2)!\).
- Thus we get a loose upper bound of \(3^{N-1} \cdot (N - 2)!\). With \(N = 9\), this is \(\approx 33\,000\,000\).
Solution

A loose upper bound can be determined as follows. Consider the pattern between $2N - 2$ and whichever of 0 or 1 the lace goes through first.

- Of each pair $\{0, 1\}$, $\{2, 3\}$, $\ldots$, $\{2N - 4, 2N - 3\}$ the lace goes through either the left hole, the right hole, or neither hole (for $\{0, 1\}$ it must go through either the left or right hole).
- We can bound the number of possible orders of pairs the lace travels through at $(N - 2)!$.
- Thus we get a loose upper bound of $3^{N-1} \cdot (N - 2)!$. With $N = 9$, this is $\approx 33\,000\,000$.
- Tighter upper bounds can show the number of valid patterns is more like 30 000, but the loose bound should suffice to show a brute force solution is fast enough.
Final Solution

Brute force all valid patterns, then count how many patterns result in shoelace length between $f_{min}$ and $f_{max}$. 
J - Ada Loveslaces

Final Solution
Brute force all valid patterns, then count how many patterns result in shoelace length between $f_{\text{min}}$ and $f_{\text{max}}$.

Pitfalls
Recomputing patterns for each shoelace length rather than storing them. In a slow language this can TLE on big cases.
Final Solution

Brute force all valid patterns, then count how many patterns result in shoelace length between $f_{\text{min}}$ and $f_{\text{max}}$.

Pitfalls

Recomputing patterns for each shoelace length rather than storing them. In a slow language this can TLE on big cases.

Statistics: 41 submissions, 6 accepted.
Problem

Given a tree, find the value of a maximum value path that visits each edge at most $k$ times, where the value of a path is the weight of every distinct edge in the path.
Problem
Given a tree, find the value of a maximum value path that visits each edge at most \( k \) times, where the value of a path is the weight of every distinct edge in the path.

Solution
- The first observation is to see that since the graph is a tree, if \( k \geq 2 \), we can take all edges, since it is always possible to visit every edge of a tree while visiting no edge more than twice; see for example: https://en.wikipedia.org/wiki/Maze_solving_algorithm#Wall_follower.
**Problem**

Given a tree, find the value of a maximum value path that visits each edge at most $k$ times, where the value of a path is the weight of every distinct edge in the path.

**Solution**

- The first observation is to see that since the graph is a tree, if $k \geq 2$, we can take all edges, since it is always possible to visit every edge of a tree while visiting no edge more than twice; see for example: [https://en.wikipedia.org/wiki/Maze_solving_algorithm#Wall_follower](https://en.wikipedia.org/wiki/Maze_solving_algorithm#Wall_follower).

- Otherwise, if $k = 2$, we are looking for the longest path in a weighted tree, also known as the *diameter* of the tree.
Solution

This is a classic problem; solutions can be found online. Two are:

1. Run a Dijkstra (DFS) from a node $u$ to find the farthest away node $v$. Run dijkstra again from $v$ to find the farthest away node $t$. The path from $v$ to $t$ is the longest in the tree (not too hard to prove yourself!).
Solution

This is a classic problem; solutions can be found online. Two are:

1. Run a Dijkstra (DFS) from a node $u$ to find the farthest away node $v$. Run dijkstra again from $v$ to find the farthest away node $t$. The path from $v$ to $t$ is the longest in the tree (not too hard to prove yourself!).

2. Root the tree arbitrarily, and run a dynamic program to find the maximum length path. Details of which are also straightforward; try yourself or consult the internet.
This is a classic problem; solutions can be found online. Two are:

1. Run a Dijkstra (DFS) from a node \( u \) to find the farthest away node \( v \). Run dijkstra again from \( v \) to find the farthest away node \( t \). The path from \( v \) to \( t \) is the longest in the tree (not too hard to prove yourself!).

2. Root the tree arbitrarily, and run a dynamic program to find the maximum length path. Details of which are also straightforward; try yourself or consult the internet.

Statistics: 22 submissions, 6 accepted.
Problem

Given a number $N \leq 10^{16}$ and $t \leq 100$, determine the number of representations $N$ has in a form of binary where each digit can be 0, 1, ..., or $t$. 
Consider the possible length-three prefixes to valid representations of $N$ with $t = 2$. Here they are listed from smallest to largest, with equal representations on the same line.

111
110  022  102
101  021
100  012  020
011
010  002
001
000
Consider the possible length-three prefixes to valid representations of $N$ with $t = 2$. Here they are listed from smallest to largest, with equal representations on the same line.

111
110  022  102
101  021
100  012  020
011
010  002
001
000

How many can be valid prefixes? The binary notation takes exactly one. Because we can represent larger values with less digits using digit 2, we could also take the line below it. But we can’t take two lines below it! If you fill the remainder of the digits with 2’s, the value represented will be less than the binary representation prefix with 0’s for the remaining digits.
Example: say the first three digits of $N$ in binary starts as 010. Then we can also potentially represent $N$ starting with 001, since $0012\ldots2 = 00202\ldots2 = 01011\ldots0$, which is only one less than the maximum number representable starting with 010 in binary.
Example: say the first three digits of $N$ in binary starts as 010. Then we can also potentially represent $N$ starting with 001, since 0012...2 = 00202...2 = 010111...0, which is only one less than the maximum number representable starting with 010 in binary.

However, if we start 000, then even if all remaining digits are 2, the maximum number we can represent is 0002...2 = 00102...2 = 00111...0, which is less than any binary number starting with 010.
Solution

- Example: say the first three digits of $N$ in binary starts as 010. Then we can also potentially represent $N$ starting with 001, since $0012 \ldots 2 = 00202 \ldots 2 = 01011 \ldots 0$, which is only one less than the maximum number representable starting with 010 in binary.

- However, if we start 000, then even if all remaining digits are 2, the maximum number we can represent is $0002 \ldots 2 = 00102 \ldots 2 = 00111 \ldots 0$, which is less than any binary number starting with 010.

- This is true if we list all the prefixes of any length! Furthermore, generalizing to general $t$ instead of 2, we can show that less than $t$ lines below the binary representation will allow a valid representation.
Final Solution

- We can brute force the start of the number, counting the number of ways to fill the rest in.
C - Redundant Binary Notation

Final Solution

- We can brute force the start of the number, counting the number of ways to fill the rest in.
- Let

\[
\text{Ways}(i, d) := \text{number of ways to represent } i \text{ in } d \text{ digits.}
\]

We can try all digits 0, \ldots, t in the leading position d and then recurse on the remaining number. We prune the DP by returning 0 if \( i > t(2^d - 1) \) or \( i < 0 \).
Final Solution

- We can brute force the start of the number, counting the number of ways to fill the rest in.
- Let

\[
\text{Ways}(i, d) := \text{number of ways to represent } i \text{ in } d \text{ digits}.
\]

We can try all digits 0, \ldots, t in the leading position d and then recurse on the remaining number. We prune the DP by returning 0 if \( i > t(2^d - 1) \) or \( i < 0 \).

- The previous analysis shows there will be at most \( O(t \log N) \) explored states of the DP when called with \( \text{Ways}(N, \lceil \log_2(N) \rceil + 1) \).
Final Solution

- We can brute force the start of the number, counting the number of ways to fill the rest in.
- Let

  \[ \text{Ways}(i, d) := \text{number of ways to represent } i \text{ in } d \text{ digits.} \]

We can try all digits 0, \ldots, t in the leading position \( d \) and then recurse on the remaining number. We prune the DP by returning 0 if \( i > t(2^d - 1) \) or \( i < 0 \).

- The previous analysis shows there will be at most \( O(t \log N) \) explored states of the DP when called with \( \text{Ways}(N, \lceil \log_2(N) \rceil + 1) \).

Statistics: 12 submissions, 6 accepted.
Problem

Given parameters of a track, determine the staggered starting locations of a $D$-meter race.
Problem
Given parameters of a track, determine the staggered starting locations of a $D$-meter race.

Solution
Draw triangles and use trigonometry to determine the locations, according to spec.
Problem
Given parameters of a track, determine the staggered starting locations of a $D$-meter race.

Solution
Draw triangles and use trigonometry to determine the locations, according to spec.

Statistics: 41 submissions, 12 accepted.
Problem

Given a schedule of $k$ trips that must be completed in a graph, determine the minimum number of drivers necessary to complete all trips.

Solution

This is a classic matching problem. The solution is to match trips to trips. A trip $i$ can be matched to trip $j$ if it is possible for a driver to complete trip $i$, then arrive at the start of trip $j$ at or before the start of trip $j$.

A maximum matching determines the number of drivers necessary. A driver can start at any unmatched trip, then follow the chain of matchings for next trips. The answer is thus $k$ minus the maximum matching.
Problem
Given a schedule of \( k \) trips that must be completed in a graph, determine the minimum number of drivers necessary to complete all trips.

Solution
- This is a classic matching problem. The solution is to match trips to trips. A maximum matching determines the number of drivers necessary. A driver can start at any unmatched trip, then follow the chain of matchings for next trips. The answer is thus \( k \) minus the maximum matching.
Problem
Given a schedule of $k$ trips that must be completed in a graph, determine the minimum number of drivers necessary to complete all trips.

Solution
- This is a classic matching problem. The solution is to match trips to trips.
- A trip $i$ can be matched to trip $j$ if it is possible for a driver to complete trip $i$, then arrive at the start of trip $j$ at or before the start of trip $j$. 

Problem

Given a schedule of $k$ trips that must be completed in a graph, determine the minimum number of drivers necessary to complete all trips.

Solution

- This is a classic matching problem. The solution is to match trips to trips.
- A trip $i$ can be matched to trip $j$ if it is possible for a driver to complete trip $i$, then arrive at the start of trip $j$ at or before the start of trip $j$.
- A maximum matching determines the number of drivers necessary. A driver can start at any unmatched trip, then follow the chain of matchings for next trips. The answer is thus $k$ minus the maximum matching.
The small number of locations can be exploited to create a smaller general maximum flow problem.
Comments

- The small number of locations can be exploited to create a smaller general maximum flow problem.

- Hopcroft-Karp can also be used instead of Ford-Fulkerson as the matching algorithm for improved speed.
The small number of locations can be exploited to create a smaller general maximum flow problem.

Hopcroft-Karp can also be used instead of Ford-Fulkerson as the matching algorithm for improved speed.

Unfortunately, we were unable to separate solutions based on graph representation and flow algorithm (without disadvantaging Java), so the time limits are set to accept them all.
The small number of locations can be exploited to create a smaller general maximum flow problem.

Hopcroft-Karp can also be used instead of Ford-Fulkerson as the matching algorithm for improved speed.

Unfortunately, we were unable to separate solutions based on graph representation and flow algorithm (without disadvantaging Java), so the time limits are set to accept them all.

Statistics: 20 submissions, 3 accepted.
Problem

Given the board of a codenames game and a list of possible hint words with which board words they are associated, determine the probability of winning the game, given that guessers pick uniformly at random from available board words associated with the given hint word.

Solution

Straightforward subset DP to evaluate all possible game states. Let $DP(board, turn, hint, guesses) = \text{probability player whose turn it is wins, given the live cards, the hint word, and how many guesses remain.}$

Board is an $N$-bit bitmask where a 1 represents a live card. The recurrence is as follows: if this is a guessing state ($guesses > 0$), then we make a uniformly-random choice among live cards associated with hint, then recurse on the remaining state.
Problem
Given the board of a codenames game and a list of possible hint words with which board words they are associated, determine the probability of winning the game, given that guessers pick uniformly at random from available board words associated with the given hint word.

Solution
- Straightforward subset DP to evaluate all possible game states.
**Problem**

Given the board of a codenames game and a list of possible hint words with which board words they are associated, determine the probability of winning the game, given that guessers pick uniformly at random from available board words associated with the given hint word.

**Solution**

- Straightforward subset DP to evaluate all possible game states.
- Let $\text{DP(board, turn, hint, guesses)} := \text{probability player whose turn it is wins, given the live cards, the hint word, and how many guesses remain.}$
Problem

Given the board of a codenames game and a list of possible hint words with which board words they are associated, determine the probability of winning the game, given that guessers pick uniformly at random from available board words associated with the given hint word.

Solution

- Straightforward subset DP to evaluate all possible game states.
- Let $\text{DP}(\text{board}, \text{turn}, \text{hint}, \text{guesses}) :=$ probability player whose turn it is wins, given the live cards, the hint word, and how many guesses remain.
- Board is an $N$-bit bitmask where a 1 represents a live card.
Problem
Given the board of a codenames game and a list of possible hint words with which board words they are associated, determine the probability of winning the game, given that guessers pick uniformly at random from available board words associated with the given hint word.

Solution
- Straightforward subset DP to evaluate all possible game states.
- Let $DP(\text{board}, \text{turn}, \text{hint}, \text{guesses}) := \text{probability player whose turn it is wins, given the live cards, the hint word, and how many guesses remain.}$
- Board is an $N$-bit bitmask where a 1 represents a live card.
- The recurrence is as follows: if this is a guessing state ($\text{guesses} > 0$), then we make a uniformly-random choice among live cards associated with $\text{hint}$, then recurse on the remaining state.
Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.
Solution

- Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.

- Number of states: $2^N \cdot 2 \cdot M \cdot N$. Time to compute each state:
Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.

- Number of states: $2^N \cdot 2 \cdot M \cdot N$. Time to compute each state:
  - $O(N)$ on a guessing state. ($2^N \cdot 2 \cdot M \cdot N$ guessing states.)
Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.

Number of states: $2^N \cdot 2 \cdot M \cdot N$. Time to compute each state:
- $O(N)$ on a guessing state. ($2^N \cdot 2 \cdot M \cdot N$ guessing states.)
- $O(MN)$ on a clue-giving state. ($2^N \cdot 2$ clue-giving states.)
Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.

Number of states: $2^N \cdot 2 \cdot M \cdot N$. Time to compute each state:
- $O(N)$ on a guessing state. ($2^N \cdot 2 \cdot M \cdot N$ guessing states.)
- $O(MN)$ on a clue-giving state. ($2^N \cdot 2$ clue-giving states.)

Final complexity: $O(2^N \cdot M \cdot N^2) \approx 700\,000\,000$ iterations, which is a lot, but bit operations are fast and a 15 second time limit generous.
Solution

- Otherwise, it is a clue-giving state, and we consider all possible remaining hint words and all possible values for $K$, taking the maximum.

- Number of states: $2^N \cdot 2 \cdot M \cdot N$. Time to compute each state:
  - $O(N)$ on a guessing state. ($2^N \cdot 2 \cdot M \cdot N$ guessing states.)
  - $O(MN)$ on a clue-giving state. ($2^N \cdot 2$ clue-giving states.)

- Final complexity: $O(2^N \cdot M \cdot N^2) \approx 700,000,000$ iterations, which is a lot, but bit operations are fast and a 15 second time limit generous.

Statistics: 45 submissions, 1 accepted.
Problem

Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.
Problem
Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.

Solution
- State-based Dijkstra.
Problem

Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.

Solution

- State-based Dijkstra.
- State: street and direction, lane, number of left turns taken so far, number of right turns taken so far.
Problem

Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.

Solution

- State-based Dijkstra.
- State: street and direction, lane, number of left turns taken so far, number of right turns taken so far.
- Graph should have about $O(NMKXY)$ vertices and slightly more edges.
Problem
Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.

Solution
- State-based Dijkstra.
- State: street and direction, lane, number of left turns taken so far, number of right turns taken so far.
- Graph should have about $O(NMKXY)$ vertices and slightly more edges.
- Following all rules correctly is the crux of the problem.
**Problem**

Given a road map of a city grid, including lane layouts, determine the shortest amount of time to fulfill a trip using at most $X$ left turns and $Y$ lane changes.

**Solution**

- State-based Dijkstra.
- State: street and direction, lane, number of left turns taken so far, number of right turns taken so far.
- Graph should have about $O(NM^2KXY)$ vertices and slightly more edges.
- Following all rules correctly is the crux of the problem.

Statistics: 0 submissions, 0 accepted.