Code Jam 2012 - Round 2

Analysis: Descending in the Dark

The main challenge in this problem is determining whether a cave is lucky. The set of all possible states is huge, so a complete search over all plans is simply not going to work. Fortunately, there is one observation we can use to greatly simplify the task.

Eliminating Backtracking

Fix a cave **C**, and recall that $\mathbf{S_C}$ denotes the set of squares from which **C** can be reached. We will build up our plan one instruction at a time.

Let **X** be the set of squares that you can be in if you start in $\mathbf{S_C}$ and follow the plan so far. If **X** contains a square not in $\mathbf{S_C}$, we know the plan cannot work for every starting position. Otherwise, we can be 100% sure that the plan so far is fine! This is because the set of possible squares we are in has either stayed the same or gotten strictly smaller. In particular, if there was a plan that worked from the starting position, we can append it to what we have done so far, and it will still work.

So what does this mean? As long as we don't do a move that adds a square not in $\mathbf{S_C}$ to the set of possible squares **X**, we can go ahead and do that move, and it will still be possible to finish. In particular, we can always move left and right whenever we want, since moving left or right can never move you out of $\mathbf{S_C}$.

All You Need Is Down!

Even with the previous observation, we still have work to do. We now know all the moves that can be done safely, but the state space is still huge. We can't find just any move; we need one that makes progress.

Here is where it is important that you cannot go up the mountain. Suppose you can add a Down move to the plan, satisfying the following two properties:

- There is at least one position in **X** from which you can actually move down. (Without this, the Down move will never do anything, and so is useless!)
- There is no position in **X** from which a down move will take you outside of $\mathbf{X_C}$.

If you add this Down move to the plan, then the sum of the heights over all squares in **X** will have gone down, and it can never go up again, because you can never climb the mountain!

Since the sum of the heights is a positive integer, you will eventually have to stop making Down moves. At that point, you are stuck in one or more horizontal intervals. If there is just one interval and it contains the cave, the plan can be finished successfully. Otherwise, you're screwed! And remember, since this set of squares is a subset of what you started with, you were in fact screwed from the start.

Can You Go Down?

Only one question remains: given a set of positions **X** reachable from the start, can you come up with a valid plan that includes at least one Down move?

X must be contained in a set of horizontal intervals, bounded by impassable squares to the left and right. Since it is always safe to move left and right, we can keep moving left until **X** is actually just the leftmost square in each of these intervals. If we cannot make progress from that situation, we have already shown we are lost.

As we perform left and right moves from there, our position within each interval might change. However, note that if two intervals have the same length, our relative horizontal positions within them will always be the same. Therefore, let's define x_j to be our relative horizontal position within all intervals of length j. (In particular, $x_j = 0$ if we are in the leftmost position, and $x_j = j - 1$ if we are in the rightmost position.)

Lemma: It is possible to reach position $(x_1, x_2, ...)$ using left and right moves if and only if $x_i \le x_j \le x_i + j - i$ for all $i < j$.

Proof: First we show $x_i \le x_j$. This is true initially. If it ever failed after some sequence of moves, it would be because x_i and x_j were equal, and then either:

- We moved left, and only x_j was able to move.
- We moved right, and only x_i was able to move.

However, both of these scenarios are impossible. Therefore, x_i is indeed at most x_j, or in other words, the distance from x_i to the left wall is no larger than the distance from x_j to the left wall. The same argument can be applied for the right wall, which gives us the other half of the inequality: $x_j \le x_i + j - i$.

Conversely, any set of positions with $x_i \le x_j \le x_i + j - i$ really can be reached via the following algorithm:

- Start with each $x_i = 0$.

- Loop from i = largest interval length down to 2.
- Move i-2 + x_i - x_{i-1} times to the right, and then i-2 times to the left.

Try it yourself and you will see why it works!

We are now essentially done. We need to determine if there is set of positions {x_i} that can be reached for which it is safe to move down. Once you have gotten this far, you can finish it off with dynamic programming. Here is some pseudo-code that determines whether there is a set of positions from which it is possible to move down and make progress:

```
old_safety = [SAFE] * (n+1)
for length in {n, n-1, ..., 1}:
  for i in [0, length-1]:
    pos_safety[i] = best(old_safety[i], old_safety[i+1])
    if moving down leaves $S_C$:
      pos_safety[i] = UNSAFE
    elif moving down is legal and pos_safety[i] == SAFE:
      pos_safety[i] = SAFE_WITH_PROGRESS
  old_safety = pos_safety
return (pos_safety[0] == SAFE_WITH_PROGRESS)
```

This is a good starting point, but you would still have to tweak it to actually record what the right x_i is for all i.

Putting it all together, we repeatedly use the above algorithm to see if it is possible to make a down move. If so, we do it and repeat. Otherwise, we stop. If the only remaining interval is the one containing the cave, then that cave is lucky. Otherwise, it is not!

Since there are no correct submissions for the large input during the contest, we provide here a full java solution (by Petr Mitrichev) so everyone can use it to generate correct outputs for various inputs.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Dark {
 static class Segment {
  int len;
  long goodExitMask;
  long badExitMask;
 }

 public static void main(String[] args) throws IOException {
  BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
  int numTests = Integer.parseInt(reader.readLine());
  for (int testId = 0; testId < numTests; ++testId) {
   String[] parts = reader.readLine().split(" ", -1);
   if (parts.length != 2) throw new RuntimeException();
   int rows = Integer.parseInt(parts[0]);
   int cols = Integer.parseInt(parts[1]);
   String[] field = new String[rows];
   for (int r = 0; r < rows; ++r) {
    field[r] = reader.readLine();
    if (field[r].length() != cols) throw new RuntimeException();
   }
   System.out.println("Case #" + (testId + 1) + ":");
   for (char caveId = '0'; caveId <= '9'; ++caveId) {
    int cr = -1;
    int cc = -1;
    for (int r = 0; r < rows; ++r)
     for (int c = 0; c < cols; ++c)
      if (field[r].charAt(c) == caveId) {
       cr = r;
       cc = c;
      }
    if (cr < 0) continue;
    boolean[][] reach = new boolean[rows][cols];
    reach[cr][cc] = true;
    int nc = 1;
    while (true) {
     boolean updated = false;
     for (int r = 0; r < rows; ++r)
      for (int c = 0; c < cols; ++c)
```

```
       if (reach[r][c]) {
        if (r > 0 && field[r - 1].charAt(c) != '#' && !reach[r - 1][c]) {
         reach[r - 1][c] = true;
         ++nc;
         updated = true;
        }
        if (c > 0 && field[r].charAt(c - 1) != '#' && !reach[r][c - 1]) {
         reach[r][c - 1] = true;
         ++nc;
         updated = true;
        }
        if (c + 1 < cols && field[r].charAt(c + 1) != '#' && !reach[r][c + 1]) {
         reach[r][c + 1] = true;
         ++nc;
         updated = true;
        }
       }
     if (!updated) break;
    }
    List<Segment> segments = new ArrayList<Segment>();
    for (int r = 0; r <= cr; ++r)
     for (int c = 0; c < cols; ++c)
      if (reach[r][c] && (c == 0 || !reach[r][c - 1])) {
       int c1 = c;
       while (reach[r][c1 + 1]) ++c1;
       Segment s = new Segment();
       s.len = c1 - c + 1;
       for (int pos = c; pos <= c1; ++pos) {
        if (r + 1 < rows && field[r + 1].charAt(pos) != '#') {
         if (reach[r + 1][pos])
          s.goodExitMask |= 1L << (pos - c);
         else
          s.badExitMask |= 1L << (pos - c);
        }
       }
       segments.add(s);
      }
    while (true) {
     int maxLen = 0;
     for (Segment s : segments)
      maxLen = Math.max(maxLen, s.len);
     long[] badByLen = new long[maxLen + 1];
     for (Segment s : segments) {
      badByLen[s.len] |= s.badExitMask;
     }
     long[] possible = new long[maxLen + 1];
     possible[1] = 1;
     for (int len = 1; len <= maxLen; ++len) {
      possible[len] &= ~badByLen[len];
      if (len < maxLen) {
       possible[len + 1] = possible[len] | (possible[len] << 1);
      }
     }
     for (int len = maxLen; len > 1; --len) {
      possible[len - 1] &= possible[len] | (possible[len] >> 1);
     }
     List<Segment> remaining = new ArrayList<Segment>();
     for (Segment s : segments)
      if ((s.goodExitMask & possible[s.len]) == 0) {
       remaining.add(s);
      }
     if (remaining.size() == segments.size()) break;
     segments = remaining;
    }
    System.out.println(caveId + ": " + nc + " " + (segments.size() == 1 ? "Lucky" : "Unlucky"));

   }
  }
 }
}
```