**Problem Khans**

**Analysis**

The task Khans was intended to be one of the hard tasks of this competition. It seemed like a modification of the travelling salesman problem, but with some simplifications:

* You can visit again the nodes, but only after some time has passed
* The graph is very sparse (only up to four edges from each node)
* Very small range of possible "prices" for the nodes

Still, the problem would look exponential to most competitors, and indeed it is. There were three main directions in which it could be approached (ordered by runtime):

1. Bruteforce
2. Dynamic Programming
3. Cycle search

We'll look at each of those.

**Bruteforce**

The most obvious way to go was just try all possible paths and select the optimal one. This was doomed to fail due to being too slow, however we could add several improvements (pruning and fast state update) to make it at least fast enough to get us some points.

The pure bruteforce (with update of the whole board on each step) was the simplest to implement, but also the slowest in terms of runtime. It would get the competitors around 15 points.

If the contestants added a quick update of the board (since we only need to update a few cells: ~8, instead of 100 if N = 10, M = 10, as we'll see later) it got much faster and would yield around 25 points.

If they added some additional pruning (by using smart heuristics) they could get even more points (I haven't experimented enough to see how much exactly, but there are \*many\* possible heuristics).

**Dynamic Programming**

This was the intended solution and there are several ways to go here as well. Before getting to the possible ideas, the contestants should have noted the following:

* N and M were really low
* Aij was really low

In fact, the constraints on Aij being so low and the fact that the food doubles each year could have led to the following observation:

*Each cell replenishes to its maximum in no more than 8 years (i.e., it for sure will be okay to enter at the start of the 9-th year).*

The example in the statement was hinting this way – even if the maximum amount of food was at the upper limit 100, the 10 years after the khans have left it would yield the following values: 0, 1, 2, 4, 8, 16, 32, 64, 100, 100.

How can we exploit this? If there is a way to keep in the state which are the last 8 visited regions, we would have everything we need to deal with the re-visiting rule. And since the movement happens only through neighboring cells of the matrix, we can keep:

* The current cell
* The last seven directions which the khans used

Since each direction is one of four possibilities, this means we need [10][10] for the current cell and [47] = [16384] for the path.

That's not too much per se, but combined with the number of years K (which we must include in the state), the table becomes [K][N][M][47] which is a bit too much. We don't have the memory to keep such a large state; however, the fact that the years are consecutive comes in handy: we can do the DP iteratively by compressing the [K] dimension of the state into a [2] dimension (every time alternating between 0 and 1 for even and odd years, respectively). Now the table [2][10][10][16384] fits nicely into the memory we have; unfortunately, it is too slow.

An interesting observation is that some (many) of these states are left unused. This usually indicates that our state is not optimal, and has the side effect that the recursive approach is much faster than the iterative one (as it only visits the states we need). So, we have two choices:

1. Use vectors to create just as large DP table as we need (also we can make it of type short instead of int, as the answer is guaranteed to be less than or equal to K \* Aij, thus 100 \* 100 = 10,000) to use just as much memory as we need. This runs quite quickly, but gets around 45 points (gets ML on the rest of the tests).
2. Use iterative DP to have no problem with the memory, but now we don't know which states are actually used and which are not, so we have to compute them all. This also gets around 45-50 points (gets TL on the rest of the tests).

Argh! So, the only way to move forward is see \*why\* our state is not optimal and try to fix it.

What we still haven't used to our advantage? Maybe the most noticeable thing from the constraints are the strange limits of Aij (10 ≤ Aij ≤ 100). Why 10 and not 1? Well, one thing this guarantees is that we will never return to the cell we just came from (it will have food 1 which is sure to be less than the maximum (10 or more)). So, the actual number of directions we can take from each cell is limited to 3, not 4 as we claimed above. This reduces the last dimension of the DP table from 47 = 16384 to 37 = 2187. Actually, we still need one direction to be precise and all others to be offset by the previous one, so in fact instead of 37 we need 4 \* 36 = 2916, but that's close enough.

So, the final state we got is:

* [K] = [100] or [2] (for recursive / iterative, respectively): the current year
* [N][M] = [10][10]: The current cell
* [4] = the direction we must take to get to the previous cell
* [3][3][3][3][3][3] = the direction (excluding the previous direction) to each of the 6 cells before that.

This state is now sufficiently small to even have the full year in the table (thus becoming [100][10][10][4][36] = 29,160,000 cells. Since, as we noted above, we can use shorts instead of ints, this requires a little less than 60MB of memory.

Since our state only guarantees that we don't return into the previous region, but have no guarantee whatsoever that we don't exit the board or return into a not replenished region after a few moves, the actual number of visited states should still be less than the 29 million cells we keep. As we mentioned earlier, the recursive approach exploits this and is faster than the iterative (the best test I managed to create visits a little less than 5 million unique states).

If implemented well, this approach will require a constant number of operations inside the recursion (just trying the 4 possible directions, exploring recursively at most 3 of them). In terms of time complexity, this means it is equal to the number of cells in our DP table: O(N \* M \* K \* 4 \* 36). In theory we should ignore 4 \* 36, but let's not go there… This was the intended solution for 100 points.

**Cycle Search**

There was a totally different way to go: assume that a large portion of the path is a "cycle" – a sequence of regions which is repeated again and again. So, an optimal path will look like that:

1. Reach a cell of the cycle from the starting cell
2. Loop the cycle X times
3. Optionally visit some more cells

Although probably possible, this solution has \*many\* corner cases which have to be handled correctly in addition to the other work that has to be done (implement the cycle search and additional cell visiting). An upside here is that the solution is polynomial, thus much faster than the intended one, having no problems with TL or ML.

It is, however, most likely susceptible to WA. Assuming the task is given with full feedback, it may be possible to "cheat" this way, but most likely the solution wouldn't really be "correct" – most likely there are cases which are not handled correctly. There are again infinite number of ways to do it, so I didn't bother writing solutions of this type (just added some hand-crafted tests against some ideas I had).

*Author: Alexander Georgiev*