Tunnels Analysis

Author: Emil Indzhev

Intro

The task in the problem is equivalent to trying to find the path of the Rabbit on the grid by repeatedly asking whether he's been through a cell. However, we have to do this level by level (i.e. we ask queries about the current level only and then "lock in" our answer for the level before proceeding). In addition, we have to do this optimally, i.e. we need to exactly minimize the depth of the decision tree for solving the given input grid. Since the grader is adaptive and we are looking for the exact minimum, we should not try to optimize some average case and hope we do well in the worst case (like we might on some other problem).

Subtask 1: $M = 2$ (5 points)

This subtask is quite trivial. For the current level, we just need to check whether the Rabbit has a choice at all (if there is only one non-blocked cell in this level or the next level, then there is no choice). Then, if there is a choice, we have to figure out where the Rabbit passed through. Wlog assume he entered at cell $S = 0$, now we just need to query $S' = 1$. If the answer is `true`, then he exited the level at cell 1, and if it is `false`, then he exited at cell 0. This is optimal, since it is actually the only way to check what happened at the current level.

Subtask 2: No blocked cells (21 points)

This subtask tests whether we understand the "tools" at our disposal for searching the exit point of the Rabbit. We can see that no matter where the Rabbit enters from, there are M possible places he could exit the level out of. One lower bound for how many queries we need per level is $\lceil \log_2(M) \rceil$, if it was possible to just binary search for the exit point. If that is possible, we see that it doesn't matter where he enters the next level, so by induction from the bottom to the top this would be optimal. Naively, we might think that we need to check the Rabbit's direction at the start of solving each level (e.g. ask the cell on the right first to see whether he passed through it) and then binary search. In that case this would be a lot more complicated. However, it should be fairly obvious that making such an initial query is very wasteful when the two intervals are not equal, so we should try to consider whether a direct binary search is possible.

Suppose we know the Rabbit entered at cell S and call his exit cell T. Now consider making a query for cell Q. If the answer to the query is `true`, then the Rabbit must have passed through it on his way to T, i.e. Q is between S and T (either $S \leq Q \leq T$ or $T \leq Q \leq S$). Similarly, if the answer is `false`, then T must be between S and Q (either $S \leq T < Q$ or $Q < T \leq S$). However, we know whether $Q < S$ or $S < Q$ since we know S (notice that querying $Q = S$ is useless), so we can simply interpret the answer to the query as splitting the interval of possible values of T around Q. Therefore, we can do the binary search as so:

```
// Invariant: T is in [L, R]
L = 0, R = M - 1
while L < R:
  X = (L + R) / 2
  if X >= S:
    if investigate(X + 1):
      R = X
    else:
      L = X + 1
  else:
    if investigate(X):
      L = X + 1
    else:
      R = X
T = L
```

Note that here we use X as the base thing we consider querying and then simply adjust our query (the Q from the above analysis), as once we understand how to query it is best to think in terms of the point where we split the interval into a $T \leq X$ case and a $T > X$ case. This will help with the rest of the problem.

Subtask 3: $N, M \leq 100$ (18 points)

This is the first subtask that tests actually solving the general version of the problem (though much slower than the optimal possible complexity). Let's start by getting a few trivial observations out of the way: First, we have to split each level into chunks of consecutive non-blocked cells. Then whatever we're doing, we'll be doing it per chunk. Each such chunk may have some useless cells that are above blocked cells or above cells with no path to the treasure. We should ignore these useless cells (there are many possible implementations, but it is best to do it implicitly, e.g. by skipping over them, instead of some "compression" of the grid, as that adds a lot of overhead).

One attempt at a solution would be to simply find the left and right bound at any given moment and do the naive binary search from above (possibly ignoring useless cells). This is rather optimistic, as it claims that nothing in the grid below matters. It is easy to see that this is not optimal in the worst case. Consider a chunk with 4 non-useless cells, the first 3 lead to straight vertical tunnels straight to the treasure (i.e. once we find that the rabbit entered through one of them, we need no further queries) and the last

one cell leads to a big open space, that will need many further queries (say 20). This naive solution would first do 2 queries to find which cell the Rabbit exited from and then proceed normally. I.e. the worst case is $2+20=22$ queries. However, we can see that we can do better, if we first ask a locally less optimal query that directly checks whether the Rabbit exited from the last cell. If yes, we proceed from there (so $1+20=21$ queries). In the other case, we do the naive solution on the remaining 3 cells, which needs 3 queries to find the exit (in the worst case) and then no further queries.

The above should tells us that we need to do a DP solution from the bottom to the top, where `dp[level][cell]` is the worst case cost to finish the solve once we figure out we entered the level at that cell. Notice that for all cells in the same chunk on a given level, this DP should have the same value. So we should solve level by level from the bottom to the top, solving chunk by chunk for the current level, using the values of the DP from the level below (below the current chunk) as continuation costs.

This will actually be the case for all solutions, as it is quite fundamental to the problem. Now the question is how to solve a given chunk. We know that all queries we do will leave us with a sub-interval of the current interval, so the natural idea is a left-right DP: `dpChunk[L][R]` is the cost of solving the chunk, if we know $L \leq T \leq R$. Then we populate this DP by iterating over all possible splits and finding the best one. The base case is when $L = R$, in which case, we simply use the value of the main DP for that cell, one level below. As for dealing with useless cells (or intervals), the way is probably to store `-1` in the DPs. Then in the outer loops (when iterating over L and R, we can simply "skip" them. So the final DP looks like so:

```
if L == R:
  dpChunk[L][R] = dp[Level + 1][L]
elif dp[Level + 1][L] == -1:
  dpChunk[L][R] = dpChunk[L + 1][R]
elif dp[Level + 1][R] == -1:
  dpChunk[L][R] = dpChunk[L][R - 1]
else:
  dpChunk[L][R] = Inf
  for X in [L, R):
    C_L = dpChunk[L][X]
    C_R = dpChunk[X + 1][R]
    C = max(C_L, C_R) + 1
    dpChunk[L][R] = min(dpChunk[L][R], C)
```

In the above code we omit the iteration over L and R (which is simply over all the possible subintervals of the chunk in some valid order), as well as the code storing the optimal splits, which we'll need later when we're actually performing the queries.

The time complexity of this solutions is $O(NM^3)$ and its space complexity is $O(NM^2)$.

Subtask 4: $N, M \leq 400$ (18 points)

This subtask simply asks of us to optimize the above (inner) DP.

We can see that, if we have some sequence of costs from the level blow and the optimal split point is X, and then we add some extra costs (expand the interval) to the right, the new optimal split point X' will have $X' \geq X$ (this is fairly trivial to show). Similarly for the left side, since the problem is symmetric. Therefore, Knuth's optimization works. I.e. instead of iterating X over $[L, R)$, we can iterate it over $[\text{opt}(L, R-1), \text{opt}(L+1, R)]$, where $\text{opt}(L, R)$ is the optimal split point for the interval. This is a standard optimization and the total run time amortizes to $O(NM^2)$.

Alternatively, using similar intuition, we can see that the cost of the split, as a function of X is convex, i.e. it decreases over some prefix and then increases over the suffix. Therefore, we can use two-pointers: for a fixed L, iterate R and keep track of the rightmost optimal X. When we increment R, we keep incrementing X, while this increment does not increase the cost of the split. This also amortizes to $O(NM^2)$. The actual code (if we don't explicitly use two-pointers), is very similar to the Knuth's optimization code (we actually use the same left bound for X), but instead of having a right bound for X from the DP we iterate until some condition. Note that, unlike with the other solutions, it is important to find the rightmost optimal X here, because of the amortization.

Both of these solutions have the same space complexity of $O(NM^2)$ but that is acceptable within the memory limit (and can easily be optimized to be divided by 2 by not allocating memory for the invalid $R < L$ cases, if needed).

Greedy approach

It should be fairly apparent that a left-right DP cannot scale to solve the full problem, as we're already computing each of its states in $O(1)$, but there is simply too many of them. Therefore, the natural next idea is a greedy approach. E.g. if there was a simple way to know the optimal split point as a function of the list of continuation costs (perhaps the point that most evenly balances their sums, or maximums or whatever), then we might be able to solve the problem more efficiently. However, to our knowledge, no such simple method works. Then, if we can't greedily split, the next natural idea is to greedily merge (after all merging is splitting in reverse).

If the list of costs was not ordered, i.e. if we could ask queries of the form "did the Rabbit exit from a cell in this subset of cells", the greedy merging algorithm is more obvious: we always merge the two smallest costs (this is known as Huffman merging) and make a new cost equal to their maximum plus 1. Actually, in that looser case, any merger that results in a minimal new cost is optimal. It turns out that something very similar works here as well.

If we consider the list of consecutive pairs of costs and the costs of their mergers, we can do the leftmost merge that results in a minimal new cost. There are multiple ways to prove this is optimal, but one way is to consider some decision tree and use a rebracketing argument on it to transform it to our decision tree, while showing we never increase the cost. The intuition behind why this works is that our "savings" come from merging equal costs, so since this is the minimal merge cost, we might as well directly do it (and it is leftmost in case there ends up being a sequence of equal costs – we want to avoid screwing up the parity). This idea gives an alternative (several times faster) solution for subtask 4, but its time complexity is still $O(NM^2)$, if implemented naively.

Subtask 5: $N, M \leq 1000$ (19 points)

It is conceptually fairly easy to optimize the above greedy solution. We simply need to put all the possible merges in a min-heap. Then we just extract the minimum (we key on pairs: new cost and position), perform the merge and add the new possible merge to the heap. Since we want to be inserting and erasing arbitrary costs in the middle of the list of costs, it is best to use a linked list. However, there are some difficulties with invalidating the no longer valid merges in the heap when we merge, since `std::priority_queue` does not support such an operation and `std::list` has no concept of invalidated nodes (they would just be deallocated). The author solution here uses a custom linked list, which does not deallocate the erased nodes, but simply marks them as invalid.

Finally, notice that while solving a chunk we implicitly compute the decision tree. The author solution explicitly constructs it and that is what it uses when making queries. For this subtask, it is okay to store in memory the decision tree for each chunk. The time complexity of this solution is $O(NM \log M)$ and its memory complexity is $O(NM)$.

Subtask 6: $N, M \leq 5000$ (19 points)

To solve the full problem we need one more observation. Instead of looking for the global minimal merger, we can find the first point where the sequence of costs is non-decreasing and then merge that locally minimal cost with the smaller of its two adjacent costs. This just means we can look for the leftmost local minimum merger instead of the leftmost global minimum merger. This can also be shown by a rebracketing argument (or shown to be optimal directly). The intuition is that this local minimum will always be a local minimum no matter what mergers we do, so it will eventually become a global minimum, and since it is already leftmost, we might as well do the merger now.

Interestingly enough, if we just look for this point (the leftmost local minimum) repeatedly, which sounds like it would lead to another $O(NM^2)$ solution, the solution ends up amortizing to $O(NM \log M)$, but that is not important for the full solution.

The full solution simply keeps a stack of strictly decreasing costs. Then whenever a new cost is processed, if it is smaller than the head of the stack, it is either pushed to the stack. If it isn't, first we need to consider merging the head of the stack backwards (if the penultimate element is smaller or equal to the new value), then we "resolve" the stack (merge the last two elements while they are equal). After this we can safely push the new value, merge it with the previous element, and again "resolve" the stack. Finally, once we are done processing the costs, we repeatedly merge the last two elements of the stack, until only a single element is left.

Again, we can construct the decision tree while doing these mergers (so the stack actually stores pointers to decision tree nodes). However, unless we optimize the storage a lot, this will not pass under the memory limit for the final subtask (but it will pass subtask 5). Therefore, we should not compute (or simply delete) the decision tree while computing the DP. Then we can recompute it just for the current chunk once we enter a new node. The time complexity is $O(NM)$, as is the space complexity (storing just one N by M array of integer costs).