Chess Rush Solution

Topics

DP, math, combinatorics, implementation

Problem author

Zsolt Németh

Pawns, Rooks and Queens

If $c_1 = c_R$ then there is a unique shortest path, which takes 1 step for the rook and the queen, and $R - 1$ steps for the pawn. Otherwise, it is impossible getting to the exit square for the pawn, and it takes exactly 2 steps for both the queen and the rook. It is also obvious that the latter can always do it in two different ways.

Therefore, we only have to be careful about enumerating the number of ways to do the required 2 steps for the queen. She always has the same two paths as the rook available, plus we need to take into account all the options involving one or two diagonal moves. For these, we have to check if any path is blocked by the edges Chess Land, and also note that two diagonal moves are only possible if $1 + c_1$ and $R + c_R$ have the same parity.

Bishops

First note that the bishop can reach its destination iff $1+c_1$ and $R+c_R$ have the same parity, otherwise the answer is 0. For a small number of rows, a carefully implemented bruteforce evaluation can also solve the problem, but if R is large, we need to be more thorough. It is useful to note that we can count the number of steps and paths separately for the cases when the bishops leaves the first row using the left diagonal and the right diagonal, and in the end, if one is shorter than the other then choose that one, and if they take the same number of steps, just sum them to get the answer.

Now, for a given starting direction, we can use a combinatorial argument to find the answer. First, imagine that we move forward in a greedy manner, bouncing between the left and right edges of Chess Land until we reach the last row in some impact column c_I. This way, we can jump $C - 1$ rows forward before hitting a wall and having to move again, except the first and last moves, which are easy to handle. This gives us an initial guess for the number of steps. It is relatively easy to see that if $c_I = c_R$ then this shortest path is unique and the previously computed length is correct. Otherwise, there are two cases depending on how we arrive at the last row: for example, if we reach the last row while moving from left to right, and the target square is further to the right of c_I (i.e. $c_I < c_R$), then the previous step length is once again correct, as we could have chosen to not go all the way to the edges in one or more previous step, which would increase the vaule of c_I so we ensure $c_I = c_R$. However, if we reach the last row while moving from left to right, and the target square is to the left of c_I (i.e. $c_R < c_I$ and we effectively "jump over" the requred destination), then we need to include one additional step into our path somewhere along the way in order to ensure $c_I = c_R$.

This way, we obtain the number n of required steps, and the number f of diagonal movements we can spare by stopping before hitting an edge, and we need to distribute them arbitrarily

over $n-1$ steps, as we cannot stop during the last step. This is equivalent to the well-known combinatorial problem of distributing f balls into $n-1$ boxes, but is also relatively easy to figure out that it is the combination

$$\binom{f+n-2}{f}.$$

Regarding the implementation of the solution, one has to be very careful about handling corner cases such as starting from the first or last column or arriving there, the direction of arrival and so. Note that in the formula above, $n-2$ can be $O(R/C)$ large which makes the evaluation a bit tricky when C is small: notice that

$$\binom{f+n-2}{f} = \frac{(f+n-2)!}{f!(n-2)!} = \frac{(f+n-2)(f+n-3)\cdots(n-1)}{f!},$$

which only has $O(f)$ terms and $f < C$, so we can answer all queries with $O(QC)$ time complexity.

We remark that alternately, the answers to all possible queries could be precomputed in $O(C^2)$ time, using the fact that while cycling through all the possible values of c_R for a fixed c_1, the values n and f can change by at most 2, making it possible to adjust the answer in constant time, but this is much more difficult to implement and was not required to pass.

Kings

It is easy to see that the king has to make exactly $R-1$ bottom-up moves, and if $|c_R - c_1| \leq R-1$, we need to advance one row in each step in order to have a shortest path. The other case, when $|c_R - c_1| > R-1$ means we need to advance one column towards the destination each step and have some free maneuverability between rows, can be solved seperately in a similar manner as the easiest case of the bottom-up problem, since this can only occur when $R < C$.

From now on, we assume $R \geq C$, so the king moves one row forward in each step and takes $R-1$ steps total, so we just need to count the number of paths. First, we observe that the number of ways we can reach the j-th column of a row is initially `ways[c1]=1` and otherwise `ways[j]=0` for the first row, and for any further row can be computed dynamically as

```
next_ways[j] = ways[j - 1]+ ways[j] + ways[j + 1],
```
where $j = 1, \ldots, C$.

We can repeat this step for every row and then return the c_R-th entry, however this will take $O(RC)$ time per query.

Next, we have to notice that with the same technique, we can precompute and store the answer for every (c_1, c_R) pair, and answer each query in $O(1)$ time after, by adding another dimension to our dynamic programming: let `DP[i][j]` denote the number of ways to go from the i-th column of the first row to the j-th column of some current row, so initially we have `DP[i][i] = 1` and every other entry is 0. Now we just repeat

```
next_DP[i][j] = DP[i][j - 1]+ DP[i][j] + DP[i][j + 1]
```

for all $i, j = 1, \ldots, C$ indices $R - 1$ times to get all the answer for the R-th row. This precomputation has the time complexity $O(RC^2)$.

Our next observation is to notice that at each iteration, instead of advancing a single row $r \to r + 1$, we can choose to compute the answer for $r \to 2r - 1$ instead. Indeed, our current `DP` array stores all the numbers of ways to get from the columns of the first row to the columns of the r-th row, or in other words, to advance r rows forward. So if we want to count the number of ways to get from the i-th column of the first row to the j-th column of the $2r-1$-st row, we can enumerate all the possible paths going through the intermediate k-th column of the r-th row ($k = 1, \ldots, C$) by the formula

$$\texttt{double\_DP[i][j]} = \sum_{k=1}^{C} \texttt{DP[i][k]} \cdot \texttt{DP[k][j]}.$$

Combining the two DP formulas, we can precompute the answer for the R-th row based on the binary representation of R. Since the last formula takes $O(C^3)$ time to evaluate for all (i, j) pairs, the total time complexity is $O(C^3 \log R)$.

Notice that the cost of advancing a single row was $O(C^2)$, so if we could somehow speed up the computation of $r \to 2r - 1$ advancements too, then generating all answers for the king could be done in $O(C^2 \log R)$ time. There are multiple ways to do this step, but they all can be a bit tricky to find, so implementing a previous, less efficient approach and studying some of its outputs for small C and different R values could be very useful to figure out the main ideas and guess how a solution works.

First, the entries of `double_DP[i][1]`, i.e. going from any column of the first row to a fixed column $j = 0$, can be computed in $O(C^2)$ time, and since reversing the paths from the i-th to the j-th column gives exactly the paths from the j-th to the i-th column, `double_DP[i][j]` is symmetric and we obtain every `double_DP[1][j]` entry immediately.

Now suppose $1 < i < j < C/2$ holds. The key observation is that we don't have to use the values of `DP` anymore, as the paths going from the i-th to the j-th column are almost the same as the paths going from the $i - 1$-st column to the $j - 1$-st column, shifted by a single column to the right. In fact, the relation

```
double_DP[i][j] = double_DP[i-1][j-1] + double_DP[1][1+i+j]
```

holds, which we can use to compute missing entries based on the only single column we have evaluated beforehand. For other indices, we can use symmetry properties and even the similar relation

```
double_DP[i][j] = double_DP[i+1][j-1] + double_DP[1][1+i-j],
```

for the cases where $1 < j < C/2$ but $j \leq i \leq C$. We can verify these formulas by mathematical induction over the rows, but there are other, more beautiful combinatorial arguments to derive

them, which we are omitting for the sake of brevity, but nonetheless encourage the interested reader to try and figure them out for his/her own pleasure.