Solution Description

1 Problem Highest

Author: Matei-Octavian Stănescu, UNSTPB

1.1 Subtask 1: $N \leq 300, M \leq 500\,000$

For this subtask, we can calculate the dp (dynamic programming) that holds the lowest cost from any floor to any floor: $d[i][j]$ = the minimum travel cost from i to j. Initialize every $d[i][i]$ to 0, and the recurrence relation is

$$d[i][j] = \min\left(\min_{\substack{k \\ i+v[k]\geq b}} (d[i][k]+1),\ \min_{\substack{k \\ i+w[k]\geq b}} (d[i][k]+2)\right).$$

Total time complexity: $O(N^3 + M)$. If using a bottom-up dp approach, the complexity becomes $O(N^2 \cdot v_{max} + M)$.

1.2 Subtask 2: $N \leq 3000, M \leq 3000$

To prove that Dijkstra yields $O(M \cdot N \cdot log(N))$ complexity, we need to make the observation that, using last subtask's notation, the sequence $d[i]_j$ is ascending (in other words, $d[i][j]$ increases as j increases, meaning that the further you have to go from i, the higher the cost).

At every query (A, B), we start from A, adding it to the priority queue. The priorities are sorted firstly after cost (lowest cost $\rightarrow$ highest priority), then after floor index (higher index $\rightarrow$ higher priority).

From floor i, we add to the priority queue the floors descending *next* from index $(i+v[i]) \ldots (i+1)$, with $cost = cost[i]+1$, but if we find a floor that has

$cost[j] \leq cost[i] + 1$, then we break. This is possible precisely thanks to the monotony of the sequence $d[A]_i$. We do the same thing for $(i+w[i]) \dots (i+1)$, with $cost = cost[i] + 2$. As a consequence, we will not visit the same floor more than twice.

Total time complexity: $O(M \cdot N \cdot log(N))$

1.3 Subtask 3: $N \leq 20\,000, M \leq 20\,000$

The road to the full solution starts here. The only observation made so far is that starting from floor A, the further you have to go, the higher the cost will be. Firstly, we convert v and w from jump sizes to pointers to the highest reachable floor: `v[i] += i`; `w[i] += i`.

Then, w needs to be updated for the case when two cost-1 jumps are made one after the other, which can sometimes be better than just making one cost-2 jump. However, note that it is not enough to attribute $w[i] = max(w[i], v[v[i]])$, since on the road from i to $v[i]$, we can encounter a node that can take you even further than $v[i]$ could. For example:

$v : 4\ 8\ 1\ 1\ 1\ 1 \dots$

Here, it is optimal to jump from the first floor to the second, then make an 8-high jump.

So we use a rmq, defined as $rmq[0][i][j] =$ the floor from the interval $[j, j+2^i)$ that takes you the highest after making a cost-1 jump. The query looks like this:

`int query_rmq(a, b, 0)` $=$ the floor i, among $[a, b]$, that has the highest $v[i]$. Similarly, we define:

`int query_rmq(a, b, 1)` $=$ the floor i, among $[a, b]$, that has the highest $w[i]$, which we will use later.

Coming back, $w[i]$ will be

$$v[\text{whichever } i \text{ from } [i, v[i]] \text{ takes you the furthest in one jump}]$$

$$\Rightarrow w[i] = v[\text{query\_rmq}(i, v[i], 0)], \quad \forall i \in \{1, \dots, n\}$$

To answer the query (A, B), it seems, at first glance, enough to greedily make 2-long jumps until you reach B, since we accounted for 2 cost-1 jumps being better than 1 cost-2 jump. However, take the example:

$$v : 1\ 1\ 1\ 1\ 1\ 1 \tag{1}$$
$$w : 1\ 6\ 1\ 1\ 1\ 1 \tag{2}$$

Starting from position 0, if we greedily jump cost-2, we will skip the 6. So it is better to calculate the dp $d[i] = \{\text{the highest you can go if you pay } i\}$. But we will only need the last two positions from it: *curr* and *prev*, since:

$$d[i+1] = max(v[\text{best up until } d[i]], w[\text{best up until } d[i-1]]),$$

more precisely:

$$next = max(v[query\_rmq(A, curr, 0)], w[query\_rmq(A, prev, 1)]).$$

Total time complexity: $O(N \cdot M)$

1.4 Subtask 4: $N \leq 200\,000, M \leq 200\,000$

Define the binary lifting DP:

$highest[k][i]$ = The highest floor you can get to from floor i, for a cost of at most 2^k

Base cases:

$$highest[0][i] = v[i] \tag{3}$$
$$highest[1][i] = w[i] \tag{4}$$

For understanding the formulas better, check out Subtask 5, as they are easier to visualize thanks to v and w being ascending.

Precalculation: In order to calculate $highest[k+1][i]$, we need to merge two $highest[k][i, j]$, for which there are 2 cases:

Case 1:

$$\texttt{|------------------|------------------|} \qquad 2^k + 2^k \tag{5}$$
$$\texttt{(---)}\ (2) \tag{6}$$

$$highest[k+1][i] = highest[k][\text{query\_rmq}(i, highest[k][i], k)]$$

where `int query_rmq(a, b, k)` = the floor i, among $[a, b]$, that has the highest $highest[k][i]$. Note: `(---)` represents the cost-2 jump that must be considered, and cannot be split between the two 2^k jumps.

Case 2:

$$\texttt{|-------------|---|-------------|} \qquad (2^k-1)+2+(2^k-1) \tag{7}$$

$$\texttt{(---)}\ (2) \tag{8}$$

For this case, we need another DP:

$highest\_m1[k][i]$ = the highest floor you can get to from floor i, for a cost of at most 2^k-1

We will return later for how to calculate this one.

Precalculation: In order to calculate $highest[k+1][i]$, we need to merge two $highest[k][i, j]$, for which there are 2 cases:

Case 1:

$$\texttt{|-----------------|-----------------|} \qquad 2^k + 2^k \tag{9}$$

$$\texttt{(---)}\ (2) \tag{10}$$

$$highest[k+1][i] = highest[k][\text{query\_rmq}(i, highest[k][i], k)]$$

where `int query_rmq(a, b, k)` = the floor i, among $[a, b]$, that has the highest $highest[k][i]$.

Case 2:

$$\texttt{|--------------|---|--------------|} \qquad (2^k-1)+2+(2^k-1) \tag{11}$$

$$\texttt{(---)}\ (2) \tag{12}$$

For this case, we need another DP:

$$highest\_m1[k][i] = \text{the highest floor you can reach from floor } i$$

for a cost of at most $2^k - 1$

We will return later for how to calculate this one.

So for the second case, $highest[k+1][i]$ will be:

$$highest\_m1[k][\text{best up to } highest[1][\text{best up to } highest\_m1[k][i]]]$$

where:

1. best up to $highest[1][I] = \text{query\_rmq}(A, I, 1)$
2. best up to $highest\_m1[k][I] = \text{query\_rmq\_m1}(A, I, k)$

This means that we have to hold log rmq_m1's and log rmq's.

Now, to calculate $highest\_m1[k+1][i]$, there are also two cases:

Case 1: $2^{k+1} - 1 = 2^k + (2^k - 1)$

```
|------------------|--------------|      2^k + (2^k - 1)      (13)
              (---) (2)                                       (14)
|----------------|----------------|      (2^k - 1) + 2^k      (15)
```

$highest\_m1[k+1][i] = highest[k][\text{query\_rmq}(i, highest\_m1[k][i], k)]$

Case 2: $2^{k+1} - 1 = (2^k - 1) + 2^k$

```
|------------------|--------------|      2^k + (2^k - 1)      (16)
                (---) (2)                                     (17)
|----------------|----------------|      (2^k - 1) + 2^k      (18)
```

$highest\_m1[k+1][i] = highest\_m1[k][\text{query\_rmq\_m1}(i, highest[k][i], k)]$

Now that we've calculated the dynamics, it's time to handle the queries.

We binary search the result by keeping the best positions $curr$ and $prev$ for num_jumps and $num\_jumps - 1$ respectively. For every k, starting from position $curr < B$, we test if we overshoot by calculating the highest floor reachable for another jump of cost 2^k. The calculation for $next\_curr$ is similar to the calculation of $highest[k+1][i]$, except we use $curr$ instead of $highest[k][i]$, $prev$ instead of $highest\_m1[k][i]$, and A instead of i in the query_rmq. The formulas are:

1. $next\_curr = highest[k][query\_rmq(A, curr, k)]$
2. $next\_curr = highest\_m1[k][query\_rmq\_m1(A, highest[1][query\_rmq(A, prev, 1)], k)]$

For $next\_prev$, it is similar to the calculation of $highest\_m1[k+1][i]$:

1. $next\_prev = highest\_m1[k][query\_rmq\_m1(A, curr, k)]$;
2. $next\_prev = highest[k][query\_rmq(A, prev, k)]$;

Total time complexity: $O(N \cdot log^2(N) + M \cdot log(N))$, memory complexity: $O(N \cdot log^2(N))$

Optimization: there is a way to reduce the memory complexity to $O(Nlog(N))$, but the time complexity still remains the same.

Firstly, notice that when calculating $highest[k+1][i]$ and $highest\_m1[k+1][i], \forall i$, we only use $query\_rmq(..., k)$. That means that we can calculate the rmq for each k and forget the previous one. Note that we will need, however, to keep the $rmq[1][i][j]$ separately, since we always use it for calculating the case where the cost-2 domino falls in the middle. Similarly, when answering the queries, instead of binary searching for every query separately, we do a parallel binary search, meaning that for every k, we **try** to add a jump of 2^k to every query. This also works because we only utilize $rmq[k][i][j]$ and then forget about it.

Total time complexity: $O(N \cdot log^2(N) + M \cdot log^2(N))$, memory complexity: $O(N \cdot log(N))$

1.5 Subtask 5: $N \leq 500\,000, M \leq 500\,000$ and v and w are ascending.

Because v and w are both ascending, there is no need to check what node from interval $a \dots b$ takes you the farthest, because it will always be b. So there is no need for all of these $rmq's$, leaving us with just $O(Nlog(N))$ memory used by $highest$ and $highest\_m1$. The formulas become:

For calculating $highest[k+1][i]$:

- $highest[k][highest[k][i]]$
- $highest\_m1[k][highest[1][highest\_m1[k][i]]]$

For calculating $highest\_m1[k+1][i]$:

- $highest\_m1[k][highest[k][i]]$;
- $highest[k][highest\_m1[k][i]]$;

For queries, calculating $next\_curr$:

- $highest[k][curr]$
- $highest[1][highest\_m1[prev]]$

For queries, calculating $next\_prev$:

- $highest[k][prev]$
- $highest\_m1[k][curr]$

Total time and memory complexity: $O((N+M) \cdot log(N))$

1.6 Subtask 6: $N \leq 500\,000, M \leq 500\,000$

Notice that in the $Nlog^2(N)$ solution, the only thing keeping the complexity high is the *rmq*, since we have to calculate *LOG* of them. What if we didn't need to? There needs to be done a greedy observation, which is:

"The floor that takes you the highest in a jump of cost 2^k, is either:

- the one that takes you the highest in a jump of cost 1, or
- the one that takes you the highest in a jump of cost 2.".

There can't be another better one, since in a sequence of jumps that add to a cost of 2^k, the first jump is the most significant, because the higher it takes you, the more floors you can reach, having an opening to floors that can take you even higher, and this propagates through every jump.

This is why only 2 floors have to be taken into consideration when making every jump, and these are the results of queries $query\_rmq(a, b, 0)$ and $query\_rmq(a, b, 1)$. Thus, we reduce the number of *rmq*'s that need to be calculated to 2.

Total time and memory complexity: $O((N+M) \cdot log(N))$