Problem A. Another Holiday Activity

First, let's learn how to solve the problem without change queries.

We will call a *cut* the division of the row (array) of people into a non-empty prefix and suffix. Let i be the number of people in the prefix, and we will call this cut the i-th. Thus, we have cuts numbered from 1 to $n-1$. Let s_i be the sum of the first i elements of the array a. For convenience, let $s = s_n$ be the total number of bags, which is divisible by n. Then we will define the cut value i as $s_i - i \cdot \frac{s}{n}$. This value intuitively represents the excess or deficiency of bags in the prefix of length i. We also note that in the array we want to achieve, where all elements are equal to $\frac{s}{n}$, this value is zero for all cuts. On the other hand, if all cut values are zero, then the array consists of equal numbers. Thus, it is necessary and sufficient for us to transform all s_i into zero using the operations described in the problem statement.

We note that in one operation, the value of s_i changes by at most one. If the i-th and $(i+1)$-th people do not give bags to each other, or if both give bags to each other, then the value does not change. Otherwise, the value decreases by -1 if the i-th gives a bag to the $(i+1)$-th, and increases by $+1$ in the remaining case. Let $m = \max\limits_{i=1}^{n-1}|s_i|$. From the reasoning above, the answer is at least m. We will show why the answer is exactly m. We will provide a constructive example for m minutes (where each minute this value will decrease by one).

Each minute, we will look at the positions i where $|s_i| = m$ and bring them closer to zero with the necessary bag transfers (we learned above how to subtract or add one). The only question that remains is why it will always be possible to make such a bag transfer for some i.

Let's analyze the case where $s_i = m$ and $|s_{i-1}| \neq m$. Then there will be no bag transfers between the $(i-1)$-th and i-th people, and the i-th must give one bag to the $(i+1)$-th. Thus, at that moment, it must hold that $a_i \geq 1$. Note that if $a_i = 0$, then $s_{i-1} = s_i + \frac{s}{n}$. That is, $s_{i-1} = m + \frac{s}{n} > m$, which contradicts the definition of m, and this could not have happened.

The case with the opposite sign for s_i and boundary cases can be analyzed similarly. Completely analogous reasoning can show that if the i-th person must give a bag to the person on the left and right, then they must have at least 2 bags at that moment. The logic behind this assertion is that if the operation cannot be performed, then we have not considered the maximum absolute value of all cut values, and for some neighboring cut, it will be greater.

Thus, we have learned to solve the problem when the array is fixed, and the value m can be easily computed in a linear pass. Now we will use a segment tree to optimize the solution. We will store the values of s_i in the leaves of the segment tree. Then, if we maintain the minimum and maximum in the segment tree, m will be either the maximum s_i or the negative of the minimum s_i. How do we update s_i during changes? We can represent the query as two operations of adding or subtracting a number from an array element. Thus, if we add a number x to a_i (which can also be negative), then all s_j with $j \geq i$ increase by x. Therefore, we need a segment tree with mass additions on subarrays and the ability to get the minimum/maximum in the array.

We obtain a solution in $\mathcal{O}(n + q \log n)$.

Problem B. Basic Problem from Little S.

Consider the graph G, whose vertices are the cells with positive height, and edges connect neighboring cells. This graph is **planar**, meaning it can be represented on a plane without crossing edges.

We can use Euler's formula for planar graphs: $c = V - E + \Gamma$, where c is the number of connected components, V is the number of vertices, E is the number of edges, and Γ is the number of faces in the plane. This formula is easy to prove by gradually adding edges to the graph. If there are no edges in the graph, the number of vertices equals the number of connected components. However, each added edge either connects two different connected components or splits one face into two, which is where the validity of the formula comes from.

Due to the condition on the monotonicity of heights, it turns out that each face in the graph is a square of size 2×2. Indeed, if there were a face of a different kind, there would strictly inside it be a non-empty set of cells U. Among these cells, we can consider the cell (i, j) with the minimum height; such a cell is not on the border, and for each of its neighbors, one of two conditions holds:

- $(i', j') \in U$ — neighbor (i, j), but then $h_{i',j'} \geq h_{i,j}$, because (i, j) has the minimum height among all cells in U;
- $(i', j') \notin U$ — neighbor (i, j), but then $h_{i',j'} \geq h_{i,j}$. Indeed, any face of graph G can be defined by a cycle that does not contain other cells from graph G inside it, but this cycle also separates the set of internal cells of the entire table U from all other cells. Thus, if $(i', j') \notin U$ is a neighbor of (i', j'), then cell (i', j') must lie in this cycle, and therefore belong to graph G, leading to $h_{i',j'} > 0$, while $h_{i,j} < 0$, which results in a contradiction.

Thus, the existence of a face that is not a 2×2 square clearly contradicts the problem's conditions.

Instead of a global change in heights, we will maintain a value called the *sea level* x and an array of heights $a_{i,j}$, where the actual height will be computed as $h_{i,j} = a_{i,j} - x$. That is, graph G after this reformulation is formed not by a set of cells with positive height, but by a set of cells with height strictly greater than x. We will learn to compute each of the terms V, E, Γ:

- V equals the number of cells in the entire table that have a height greater than x;
- E equals the number of pairs of neighboring cells, both of which have a height greater than x;
- Γ equals the number of 2×2 squares (as shown above, this is how all faces of graph G look) in which all cells have a height greater than x.

Since after the reformulation we only have point changes in heights, changing the height of one cell will affect at most four edges and at most four faces. To be able to quickly compute the value of x, we will maintain a set S consisting of certain numbers, each with a weight. We introduce the notation $w(x)$ — the weight of the number x, where $x \in S$. The weights will be formed as follows:

- For each cell in the table (i, j), the weight $w(h_{i,j})$ increases by 1;
- For each pair of neighboring cells in the table (i_1, j_1), (i_2, j_2), the weight $w(\min\{h_{i_1,j_1}, h_{i_2,j_2}\})$ decreases by 1;
- For each square consisting of cells (i_1, j_1), (i_2, j_2), (i_3, j_3), and (i_4, j_4), the weight $w(\min\{h_{i_1,j_1}, \ldots, h_{i_4,j_4}\})$ increases by 1.

With the introduced weights w, we can now say that $V - E + \Gamma$ is simply equal to the sum of the weights $w(y)$, taken over all numbers $y \in S$ that are greater than x. If the weights of the cells were small, the problem could already be easily solved using a segment tree or a Fenwick tree, but unfortunately, this is not the case. Therefore, it is proposed to take a few more actions:

- Introduce a set $\hat{S} = \{s_1 < s_2 < \ldots < s_k\}$ — all possible heights of cells (not the actual heights, but those we introduced along with the sea level) that occur at any moment in time;
- Compress the coordinates based on the array $s_1, \ldots, s_k$, meaning we now consider that increasing the weight $w(s_i)$ by Δ is actually increasing the i-th position in the array w' by Δ;
- To compute the sum of weights for numbers greater than x, we find the minimum index j such that $s_j > x$ and compute the sum $w'_j + w'_{j+1} + \ldots + w'_k$.

The final step, which involves using a segment tree that will definitely solve the problem, is left as an exercise for the readers. The author of this problem is a great lover of mathematics, and he couldn't help but leave this sarcastic comment in the analysis, sorry.

Problem C. Crossword

This problem can be solved using brute force. First, we will iterate through which word will be at the top, bottom, left, and right. There are a total of 24 possible arrangements. Then, for each arrangement, we will check if it is possible to construct a crossword.

To perform the check, we can iterate through positions i and j $(i < j)$ in each word where the given word will intersect with the two others. After that, it is necessary to check if the characters at the intersection of the words match, as well as whether the lengths of the opposite sides of the "rectangle" formed in the center of the crossword match. This check can be performed in $\mathcal{O}(n^8)$, where $n = \max\limits_{i=1}^{4}|s_i|$.

Thus, we obtain a solution in $\mathcal{O}(n^8)$ with a constant factor of order 24. This solution can fit within the limits with careful implementation.

The solution can also be easily optimized to achieve a complexity of $\mathcal{O}(n^6)$ with a constant factor of order 24, if for the bottom and right words we only iterate through one position (the second position is calculated using a simple formula, as the first position and the length of the side of the "rectangle" are known).

Problem D. Drinking Coffee is Hard

For convenience, let's sort the array a in non-increasing order. This will not change the answer to the problem.

We will learn to understand whether the athletes can drink the subsegment from the l-th to the r-th cups of coffee inclusive. If $r - l + 1 > n$, then the answer is obviously no. Otherwise, it is claimed that it is sufficient to check that for all i from 1 to $r - l + 1$ inclusive, the condition $b_{r-i+1} \le a_i$ holds. If this is satisfied, then the athletes can indeed drink all the coffee in the subsegment.

Assume that this is not the case; that is, there exists j such that $b_{r-j+1} > a_j$. Then on this subsegment, there are at least j cups of coffee with strength b_{r-j+1} (or more), and there are fewer than j athletes with such endurance, meaning they definitely cannot drink all the coffee from the subsegment.

Now we will use the two-pointer method. By iterating r in decreasing order, we will maintain a minimum boundary l_r such that for all $l \ge l_r$, the subsegments of coffee from l to r are feasible for the athletes. In this case, either $l_r = 1$, or the subsegment $[l_r - 1, r]$ is not feasible, but then all smaller l will also make the subsegment infeasible.

For a fixed r, we can decrease l_r while the above condition holds. We can check this condition with a single comparison of elements from two arrays. How do we transition from r to $r - 1$? Notice that if $[l_r, r]$ is feasible, then $[l_r, r - 1]$ is also feasible, since we simply do not need to drink one of the cups of coffee. Thus, we can initialize $l_{r-1} = l_r$.

It is clear that we have arrived at a fairly standard description of the two-pointer method. The answer to the problem will then be $\sum_{r=1}^{m} r - l_r + 1$.

Time complexity: $\mathcal{O}(n \log n + m)$.

Problem E. Enduring the Pokémon

Let's say we bought a Pokémon with strength x. It is easy to notice that battles should optimally be conducted in non-decreasing order of the opponents' strengths.

We will sort all the Pokémon by strength. Notice that if the i-th Pokémon can win the tournament, then the Pokémon $i + 1, i + 2, \ldots, n$ can also do so, as their strength is not less than that of the i-th Pokémon. Thus, we can use binary search to find the first Pokémon in sorted order that can win the tournament.

Now we know that the Pokémon $x, x+1, \ldots, n$ can win the tournament, while the Pokémon $1, 2, \ldots, x-1$ cannot. To find the answer, we will choose the minimum cost among the Pokémon $x, x+1, \ldots, n$.

Time complexity: $\mathcal{O}(n \log n)$.

Problem F. Fox on the Tree

We define a path from s to t, let its length be k, and its vertices along the path be $p_1, p_2, \ldots, p_k$ ($p_1 = s$, $p_k = t$).

Let d_v denote the number of children of vertex v. We define jump_v as the number of paths starting at vertex v, visiting some vertices in the subtree of v, and ending at an ancestor of vertex v. This value can be calculated as follows:

$$\text{jump}_v = 1 + f_{d_v - 1} \cdot \sum_{v \to u} \text{jump}_u,$$

where f_x is the number of ways to choose a subset from a set of x elements and arrange it in some order, and is calculated as follows:

$$f_x = \sum_{y=0}^{x} \binom{x}{y} \cdot y!$$

Indeed, such a path either goes directly up to the ancestor, or, if we look at this path from the end, it first visits some subset of the children of v, then moves to child u, somehow traverses the subtree of u, and ends at vertex v, meaning the number of such paths equals $f_{d_v-1} \cdot \text{jump}_u$.

Now consider the fox's route, which starts at vertex $p_1 = s$. Let's examine how a path from vertex p_i to vertex t can be structured.

We will denote the subtree of vertex p_i, excluding the subtree of vertex p_{i+1}, by the symbol S_i.

Then there are the following options for how the fox's path is structured:

1. $p_i \to (\varepsilon \mid S_i \mid S_{i-1}) \to p_{i-1} \to (\varepsilon \mid S_i) \to p_{i+1} \to \ldots \to t$;
2. $p_i \to (\varepsilon \mid S_i \mid S_{i+1}) \to p_{i+1} \to \ldots \to t$;
3. $p_i \to (\varepsilon \mid S_{i+1}) \to p_{i+2} \to \ldots \to t$.

Here, the symbol ε denotes skipping this step, and the symbol $\mid$ denotes the choice of one of several options.

We will calculate the number of paths dynamically as follows. The dynamic parameter will be the number i, denoting the vertex p_i from which we are building the path, and we will also keep a flag indicating whether we have already been at vertex p_{i-1}, as well as maintain the following. Notice that according to the structure of the fox's path described above, we enter the subtree S_i a certain number of times from vertex p_i, a certain number of times from vertex p_{i-1}, and a certain number of times from vertex p_{i+1}. In the dynamic state, we will keep track of the number of transitions of each type that we have already made for vertices p_{i-1} and p_i.

Now, when we make a transition in the dynamic state, we will multiply the answer by the number of ways to build a path through the subtree S_{i-1}. For this, we will need information about how many times we made each of the transitions at vertex p_{i-1}. Let's analyze how to calculate this.

Notice that we have entered and exited each subtree an even number of times, not exceeding four (this follows from all the options for constructing the path). We will denote by the triple (a, b, c) the number of times we have entered or exited vertices p_{i-1}, p_i, and p_{i+1}, respectively. Then the following options are possible for the subtree of vertex v:

1. $a = b = c = 0$;
2. $c = 0, a = b = 1$ or $a = 0, b = c = 1$;
3. $a = c = 1, b = 0$;
4. $a = 2, b = c = 1$ or $c = 2, a = b = 1$.

Unfortunately, the space of this analysis is too narrow to discuss each of these cases. Those interested are invited to independently calculate the number of ways in each of them. This is done similarly to the dynamics of jump_v and f_x described above.

We obtain a solution in time $\mathcal{O}(n)$, but with a large constant.

Problem G. Good Colorings 6

Let us define the distance between cells (i_1, j_1) and (i_2, j_2) as $|i_1 - i_2| + |j_1 - j_2|$.

Notice that among any three cells, there will be two cells whose distance is even. To prove this, it is enough to look at the chessboard coloring of the table and notice that among three cells, there will be two cells of the same color. The distance between them will be even.

Thus, among the initially known three cells, we can choose two whose distance will be even. Consider some shortest path (that is, a path with minimal distance) connecting the two chosen cells. The length of such a path will be no more than $n + m$. Let the chosen path pass through cells $(i_1, j_1), (i_2, j_2), \ldots, (i_{2k+1}, j_{2k+1})$ for some number k (the length of the path will be equal to $2k$). Consider the cells $(i_1, j_1), (i_3, j_3), (i_5, j_5), \ldots, (i_{2k+1}, j_{2k+1})$. It is claimed that in this sequence of cells, there will be two adjacent cells of different colors.

To find such cells, it is sufficient to use binary search. We will maintain the invariant that cells l and r have different colors. Initially, this is true since the colors of the known cells are pairwise different. Next, we will choose the midpoint of the segment m and compare the color of cell m with the colors of cells l and r. If the colors of cells l and m are the same, then the colors of cells m and r are different, and we can transition to the state (m, r). If the colors of cells m and r are the same, then the colors of cells l and m are different, and we can transition to the state (l, m). In the other case, we can transition to either state (l, m) or (m, r). After the binary search converges, two adjacent cells of different colors will be found. This action will be performed in $\log_2 \frac{n+m}{2}$ queries.

Now we need to consider different scenarios. If the two found cells are in the same row or the same column, we need to find out the color of the cell that is between them (denote it as (x, y)), and then find out the color of another cell adjacent to (x, y). After this, at least one corner of different colors will be guaranteed to be found.

If the cells are neither in the same row nor in the same column, we need to find out the color of any common neighbor of theirs.

Thus, no more than $\log_2 \left(10^9\right) + 2$ queries will be required, which fits within the limit.

Problem H. Hashing

We will compute the hash for each prefix of the array, denoting the hash of the i-th prefix as h_i. Then $h_1 = a_1$, and $h_i = (h_{i-1} \cdot k + a_i) \bmod p$.

Now we will iterate over the end of the subarray r and find the number of such numbers $l \le r$ such that the hash of the subarray $[l, r]$ equals the number x. It is easy to notice that the hash of the subarray $[l, r]$ can be expressed as follows:

$$(h_r - h_{l-1} \cdot k^{r-l+1}) \bmod p$$

Thus, the problem reduces to quickly finding the number of numbers $l \le r$ such that $h_r - h_{l-1} \cdot k^{r-l+1} \equiv x \pmod{p}$.

We multiply both sides of the equation by $k^{-r} \bmod p$ (this value can be found using Fermat's little theorem, since the number p is prime). We obtain the following equality:

$$h_r \cdot k^{-r} - h_{l-1} \cdot k^{-(l-1)} \equiv x \cdot k^{-r} \pmod p$$

Finally, we will move the terms to the appropriate sides of the equation:

$$h_r \cdot k^{-r} - x \cdot k^{-r} \equiv h_{l-1} \cdot k^{-(l-1)} \pmod p$$

Let $f_i = h_i \cdot k^{-i} \bmod p$. Then the equality takes the following form:

$$f_r - x \cdot k^{-r} \equiv f_{l-1} \pmod p$$

We will create a `map` to store for each y the number of indices i such that $f_i = y$. Then, while iterating over r, to compute the number of suitable l, we need to access the `map` with the key $(f_r - x \cdot k^{-r}) \bmod p$. After that, we need to increment the value for the key f_r by one.

We have obtained a solution in $\mathcal{O}(n \log n)$.

Problem I. Industrial Robots

Note that at the beginning of second t, the robots will be in columns that are $n - t$ apart. Let us denote $dp_{t,i}$ — can the robots be in columns i and $i + n - t$ at time t. This dynamic can be easily calculated in $\mathcal{O}(n^2)$. Now let's learn to calculate the answer faster.

We will iterate over t in increasing order and maintain the values of $dp_{t,i}$ for all i in the form of a set of segments $[l_j, r_j]$ such that $dp_{t,i} = 1$ for all $i \in [l_j, r_j - n + t]$ and $dp_{t,i} = 0$ for all other i. Initially (at $t = 0$), the set consists of a single segment $[1, n]$.

When transitioning from t to $t + 1$, the height of the ceiling in columns with $h_i = t + 1$ becomes zero, so all segments containing such values of i need to be split into segments that do not contain such i. After that, all segments with a length less than $n - t$ need to be removed. It is easy to see that this transition results in a set of segments for time $t + 1$.

We can maintain all segments in two copies of `std::set`, one of which has segments sorted by the left boundary, and the other by length. Splitting a segment at position i can be done by performing `lower_bound` in the first set, while removing segments of length less than $n - t$ corresponds to removing several smallest elements from the second set. This gives a solution in $\mathcal{O}(n \log n)$.

It can also be noted that the number of segments at time t does not exceed $\frac{n}{n-t}$, since all segments have a length of at least $n - t$, so from the estimate of the harmonic series, it follows that the total number of segments does not exceed $\frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{n} = \mathcal{O}(n \log n)$.

This means that segments can be maintained and recalculated directly without using a set, and the solution will work in $\mathcal{O}(n \log n)$.

Problem J. Just a Turtle Problem

Note that the table has an interesting property, namely for any cell (i, j) such that $i < n$ and $j < m$, the values in cells $(i + 1, j)$ and $(i, j + 1)$ are the same. This property holds because each subsequent row of the table is a cyclic left shift of the previous row of the table.

Thus, at any moment in time, regardless of which of the possible moves the turtle chooses, it will add the same number to the sum.

Initially, the sum is equal to a_1. Then a_2 will be added to the sum, then a_3, and so on. In total, the turtle will visit $2n - 1$ cells. Therefore, the answer will be equal to $a_1 + a_2 + \ldots + a_n + a_1 + a_2 + \ldots + a_{n-1}$.

Time complexity: $\mathcal{O}(n)$.

Problem K. Keeping the Medians Close 1

Consider an arbitrary number x. Let $d_i = +1$ if $a_i \geq x$ and $d_i = -1$ otherwise. Then it is easy to see that $\text{median}(a) \geq x \iff \sum d_i > 0$.

Consider all numbers $c_1 < c_2 < \ldots < c_{2n}$ that appear in arrays a and b. Notice that it cannot be the case that $\text{median}(a) \geq c_{n+1}$ and $\text{median}(b) \geq c_{n+1}$. If we replace all numbers $\geq c_{n+1}$ with $+1$, and the others with -1, then from the first observation it follows that the sum in array a and in array b must be positive. However, the sum of all numbers in arrays a and b is equal to 0, so such a situation is impossible.

Similarly, it can be proven that it cannot be the case that $\text{median}(a) \leq c_n$ and $\text{median}(b) \leq c_n$. Thus, without loss of generality, we can assume that:

$$\text{median}(a) \leq c_n < c_{n+1} \leq \text{median}(b)$$

This gives the following lower bound on the answer:

$$|\text{median}(a) - \text{median}(b)| \geq c_{n+1} - c_n$$

It turns out that such a bound is always achievable. To construct an example, we first perform operations so that the number c_n ends up in array a, and c_{n+1} ends up in array b.

Define $g(x)$ as follows:

$$g(x) = \begin{cases} -1 & \text{if} \quad x < c_n \\ 0 & \text{if} \quad x = c_n \text{ or } x = c_{n+1} \\ +1 & \text{if} \quad c_{n+1} < x \end{cases}$$

Then we want to achieve $\sum g(a_i) = \sum g(b_i) = 0$. It is easy to see that $\sum g(a_i) + \sum g(b_i) = 0$, so it is sufficient to achieve $\sum g(a_i) = 0$.

Consider all indices where a_i and b_i are not equal to c_n and c_{n+1}. We will apply operations to these indices in such a way that $a_i < b_i$. Notice that after this, $\sum g(a_i) \leq 0$. Now we will sequentially apply the operation to each of these indices. After this, it will hold that $\sum g(a_i) \geq 0$. Since after applying one operation this sum can increase by at most 1, at some point this sum must have been equal to 0, which is what we needed.

Time complexity: $\mathcal{O}(n \log n)$.

Problem L. Keeping the Medians Close 2

Before reading the solution to this problem, it is recommended to familiarize yourself with the solution to the easier version.

Assume that with an optimal choice of operations, the median of array a will be less than the median of array b. We will iterate over the median of array a in the order from c_n to c_1. We will maintain a pointer to the minimum value c_r of the median of array b that can improve the answer. Initially, $r = 2n$.

Let us fix the median of the first array c_l. We will decrease the value of c_r as long as we can perform no more than k operations such that $\text{median}(a) = c_l$ and $\text{median}(b) \leq c_r$. Each time we decrease the right boundary, we need to update the answer.

It remains to understand how to check if we can achieve $\text{median}(a) = c_l$ and $\text{median}(b) \leq c_r$. The operation with the index where the value c_l is located is determined unambiguously, while the operations with the other indices can either be done or not.

We will keep track of the *balances* of both arrays. To calculate the balance of array a, we need to replace each element greater than c_l with $+1$, and all other elements with -1, and then calculate the sum. Similarly, we will define the balance of array b, but we will replace elements greater than c_r with $+1$.

In terms of balance, we need to achieve that the balance of array a is equal to -1, and the balance of array b is less than 0. Considering the forced operation to place the element c_l in array a, we will find the balances of arrays a and b before applying all other operations. Let us denote these balances as x and y, respectively.

Let us consider how operations can change x and y. All operations can be divided into six types:

1. $x \to x \pm 1$, $y \to y$
2. $x \to x$, $y \to y \pm 1$
3. $x \to x \pm 1$, $y \to y \mp 1$

While moving the pointers, we can keep track of how many operations of each type there are.

Now we just need to find the minimum number of operations required to satisfy the conditions: $x = -1$ and $y < 0$. For this, the following greedy algorithm works:

- We maximize the use of the operation that brings x closer to -1 and decreases y.
- We maximize the use of the operation that brings x closer to -1 and does not change y.
- We maximize the use of the operation that brings x closer to -1 and increases y.
- If $y \geq 0$, we apply the operation that decreases y for $y + 1$ times.
- It may seem that other types of operations need to be used, but due to the correlation between x, y, and the number of operations of each type arising in this problem, the algorithm can be stopped here.

This solution can be implemented in $\mathcal{O}(n \log n)$, where $\log n$ arises only from sorting all elements.

Problem M. Magnification Matrix

You are given a rectangular grid with some cells already blocked. Determine the probability p of blocking the remaining (unblocked) cells such that the **expected number of paths** from the **top-left** corner to the **bottom-right** corner, moving only **right or down**, is exactly k.

Solution Outline

- Be very cautious with numerical calculations. Many sensible algorithms are **numerically unstable**.
- For this reason, the use of **arbitrary-precision arithmetic** (big integers) was permitted in the problem statement — this was actually a hint!
- One can observe that this problem lends itself well to a **binary search** on the value of the result (i.e., the desired probability p).
- Initially, check whether $p = 1$ yields an expected number of paths that is greater than or equal to the desired value k.
- Also, eliminate edge cases early, such as when the starting or ending cell is blocked.

Expected Value Computation

- We need a function that takes p as a parameter and returns the **expected number of valid paths**.
- Apply **dynamic programming** over the grid, using only the **special cells** (start, end, and blocked ones).
- Each cell stores the expected number of valid paths reaching it.

Efficient Calculation

- The result for each state can be calculated using the **inclusion-exclusion principle** and simple combinatorics (e.g., binomial coefficients).
- Due to the specific structure of the recurrence, the complexity of computing a state becomes **linear**, rather than exponential.