Problem A. Depth of Interval

The number of possible pairs corresponding to the top two positions in a contiguous subarray with all distinct elements is linear in the length of the array. More concretely, the following lemma holds.

> **Lemma.**
>
> Let N be a positive integer and let P be an arbitrary permutation of $(1, 2, \ldots, N)$. For each $1 \leq L \leq R \leq N$, define integers a and b such that among $P_L, P_{L+1}, \ldots, P_R$, the smallest element and the second smallest element are P_a and P_b, respectively. Then, the total number of pairs (a, b) that can arise in this way is $\Theta(N)$.

This follows from the fact that the set of half-open intervals $[\min(a, b), \max(a, b))$ over all pairs (a, b) with $a \neq b$ is laminar.

First, enumerate all possible pairs (a, b). Starting from $(L, R) = (1, N)$, we can recursively shrink the interval (L, R) so that the corresponding pair (a, b) changes, and process these intervals recursively. This can be done in $O(N \log N)$ time using a segment tree or similar data structures.

Next, when computing $f(L, R)$ recursively according to its definition, note that at recursion depths of at least 1, the number of possible (L, R) pairs is $O(N)$. Therefore, it is effective to memoize already computed values of $f(L, R)$ to avoid redundant computations. Using this approach, we precompute $f(\min(a, b) + 1, \max(a, b) - 1) + 1$ for all possible pairs (a, b).

Then, for each possible pair (a, b), we count how many intervals (L, R) correspond to it. This can be done by counting the number of closed intervals $[L, R]$ that contain the closed interval $[\min(a, b), \max(a, b)]$ and that do not contain any element smaller than P_b other than P_a. This can be computed efficiently using binary search on a segment tree or similar techniques.

Finally, for $k = f(\min(a, b) + 1, \max(a, b) - 1) + 1$, we add the number of such intervals (L, R) to the answer for k. By doing this for all possible (a, b), we obtain the distribution of $f(L, R)$ over all (L, R), which is exactly what we need to compute.

The overall time complexity of this algorithm is $O(N \log N)$.

Problem B. Increasing Swaps

Hereafter, let left_rotate denote the operation that takes the first element of a sequence and appends it to the end of the sequence. Conversely, let right_rotate denote the operation that takes the last element of a sequence and prepends it to the beginning.

Then, a single operation consists of applying left_rotate to each disjoint contiguous subarray.

Moreover, for any t, if an element belongs to the same contiguous subarray in the t-th operation, it must belong to the same contiguous subarray in the $(t + 1)$-th operation as well.

Let $Q = P^{-1}$, and consider performing the operations in reverse order, aiming to sort Q in ascending order. Then, each operation consists of applying right_rotate to each of several disjoint contiguous subarrays.

We introduce the following notation:

- If a sequence a is obtained by concatenating sequences a_1 and a_2 in this order, we write $a = a_1 + a_2$.
- Let $f(a)$ denote the solution value for sequence a.
- A sequence $a = a_1 + a_2 + \cdots + a_\ell$ is said to be splittable into $a_1, a_2, \ldots, a_\ell$ if the following conditions hold:
  - a_1 is a permutation of $(1, \ldots, |a_1|)$,
  - a_2 is a permutation of $(|a_1| + 1, \ldots, |a_1| + |a_2|)$,
  - $\vdots$

- a_ℓ is a permutation of $(\sum_{i=1}^{\ell-1} |a_i| + 1, \ldots, \sum_{i=1}^{\ell} |a_i|)$.

Under this definition, the following greedy strategy is optimal (the proof is given later):

- Repeatedly apply right_rotate until the sequence becomes splittable into at least two parts for the first time.
- Split the sequence so that the number of parts after splitting is maximized, and then solve each resulting subsequence independently.
  - That is, if $a = a_1 + a_2 + \cdots + a_\ell$ is splittable into $a_1, a_2, \ldots, a_\ell$, then $f(a) = \max\big(\{f(a_1), f(a_2), \ldots, f(a_\ell)\}\big)$.

Therefore, the problem can be solved by repeatedly applying right_rotate until the sequence becomes splittable, and then recursively solving each subproblem.

Moreover, the number of right_rotate operations required until the sequence first becomes splittable can be determined in linear time with respect to the sequence length using the algorithm described below, and the actual splitting can also be computed in linear time. Since the recursion depth is at most N, and at each depth we process a total of N elements, the overall time complexity is $O(N^2)$.

Algorithm to Find the Number of right_rotate Operations Until the Sequence First Becomes Splittable into Two or More Parts

Let $x = (x_1, x_2, \ldots, x_N)$ be a permutation of $(1, 2, \ldots, N)$.

We process the values in increasing order. Since the set of values is $\{1, 2, \ldots, N\}$, this can be done in $O(N)$ time. Prepare a binary array $A = (A_1, \ldots, A_N)$, which is considered circular, i.e., the right neighbor of A_N is A_1.

- Initially, all values in A are set to 0.
- For $i = 1, \ldots, N-1$, do the following:
  - Look at all positions j such that $Q_j = i$ (processing values in increasing order), and set $A_j \leftarrow 1$.
  - If all the positions with value 1 in A form a single contiguous interval, then the sequence is splittable with the first segment having length i. By maintaining the indices where 0 and 1 are adjacent, we can determine in $O(1)$ time whether the sequence is splittable and how many times right_rotate must be applied.
- Among the values of the required number of right_rotate operations obtained for $i = 1, \ldots, N-1$, return the minimum.

Proof That the Greedy Strategy Is Optimal

We first prove the following lemma.

Lemma

For any sequence a and its subsequence a', we have $f(a') \leq f(a)$.

- When performing a right_rotate operation, extracting a subsequence results either in a right_rotate operation on that subsequence or in no change.

- Operations that do nothing can be eliminated by delaying the start of operations on that interval by one step.
- Therefore, from a sorting sequence for a, we can construct a sorting sequence for a' that uses no more operations.

Based on this lemma, we proceed with the proof.

For a sequence $a = a_1 + a_2$ of length n, it suffices to show that if a is splittable into a_1 and a_2, then $f(a) = \max(f(a_1), f(a_2))$.(If a is not splittable, we must apply right_rotate to the entire sequence anyway, so it suffices to consider the splittable case.)

The cases $n = 1, 2$ are trivial. Assume that the statement holds for all lengths $m \leq n - 1$.

Let t be a splittable sequence of length n, and let t' be the sequence that becomes splittable for the first time after applying right_rotate to t one or more times. We distinguish cases based on the relationship between the length of the leading contiguous segment obtained by splitting t and that obtained by splitting t'.

Case 1: The lengths are equal

In this case, the leading segments obtained from t and t' are permutations of the same set of elements. Since t' is obtained by repeated right_rotate operations, this can only happen when $t = t'$. Thus, the statement holds trivially.

Case 2: The leading segment length of t is smaller than that of t'

Let

$$t = t_1 + t_2 + t_3 + t_4, \quad t' = t_4 + t_1 + t_2 + t_3,$$

and assume that the leading segment obtained by splitting t is t_1, while that obtained by splitting t' is $t_4 + t_1 + t_2$. We may assume $|t_1|, |t_3|, |t_4| > 0$ and $|t_2| \geq 0$.

Then:

- Splitting t directly yields a solution of

$$\max(f(t_1), f(t_2 + t_3 + t_4)).$$

- Applying right_rotate to t until it becomes t' and then splitting yields a solution of

$$|t_4| + \max(f(t_4 + t_1 + t_2), f(t_3)).$$

Consider $f(t_2+t_3+t_4)$. After $|t_4|$ applications of right_rotate, the sequence can be split into t_4+t_2 and t_3. If it were splittable with fewer operations, this would contradict the definition of t'. Since $|t_2+t_3+t_4| < n$, we have

$$f(t_2 + t_3 + t_4) = |t_4| + \max(f(t_4 + t_2), f(t_3)).$$

Therefore,

$$\begin{aligned}
&\max(f(t_1), f(t_2 + t_3 + t_4)) \\
&\quad = \max\big(f(t_1), |t_4| + \max(f(t_4 + t_2), f(t_3))\big) \\
&\quad \leq |t_4| + \max(f(t_1), f(t_4 + t_2), f(t_3)) \\
&\quad \leq |t_4| + \max(f(t_4 + t_1 + t_2), f(t_3)).
\end{aligned}$$

Thus, the claim holds.

Case 3: The leading segment length of t is larger than that of t'

Let

$$t = t_4 + t_1 + t_2 + t_3, \quad t' = t_1 + t_2 + t_3 + t_4,$$

and assume that the leading segment obtained by splitting t is $t_4+t_1+t_2$, while that obtained by splitting t' is t_1. Then we have:

- Splitting t directly yields a solution of

$$\max(f(t_4 + t_1 + t_2), f(t_3)).$$

- Applying right_rotate to t until it becomes t' and then splitting yields a solution of

$$|t_1| + |t_2| + |t_3| + \max(f(t_1), f(t_2 + t_3 + t_4)).$$

Since the sequence $t_4 + t_1 + t_2$ becomes splittable into t_1 and $t_2 + t_4$ after $|t_1| + |t_2|$ operations, we have

$$f(t_4 + t_1 + t_2) \le |t_1| + |t_2| + \max(f(t_1), f(t_2 + t_4)).$$

Therefore,

$$\begin{aligned}
&\max(f(t_4 + t_1 + t_2), f(t_3)) \\
&\quad \le \max\big(|t_1| + |t_2| + \max(f(t_1), f(t_2 + t_4)), f(t_3)\big) \\
&\quad \le |t_1| + |t_2| + |t_3| + \max(f(t_1), f(t_2 + t_4), f(t_3)) \\
&\quad \le |t_1| + |t_2| + |t_3| + \max(f(t_1), f(t_2 + t_3 + t_4)).
\end{aligned}$$

Thus, the claim is proven.

Hence, if a sequence t is splittable, it is optimal to split it immediately. By mathematical induction, this holds for sequences of any length.

Problem C. Sum of Three Inversions

Let $Z = N-X-Y$. Then Z represents the number of occurrences of 3 in A. Also, define $D_i = (A_i, B_i, C_i)$.

For integers i, j with $1 \le i < j \le N$, define the **contribution** of the pair (i, j) as the number of inequalities among the following three that hold: $A_i > A_j$, $B_i > B_j$, and $C_i > C_j$. With this definition, the problem can be rephrased as counting the number of sequences D such that the total contribution over all pairs (i, j) is exactly K.

Now, partition the permutations of $(1, 2, 3)$ into the following two groups:

- Group P: $(1, 2, 3), (2, 3, 1), (3, 1, 2)$
- Group Q: $(1, 3, 2), (2, 1, 3), (3, 2, 1)$

Then, the following fact holds:

- For any i, j with $1 \le i < j \le N$, if D_i and D_j belong to permutations from different groups, then the contribution of (i, j) is exactly 1.

This is because when (A_i, B_i, C_i) and (A_j, B_j, C_j) are permutations from different groups, exactly one of the three equalities $A_i = A_j$, $B_i = B_j$, and $C_i = C_j$ holds. If $A_i = A_j$, then exactly one of the inequalities $B_i > B_j$ and $C_i > C_j$ holds, so the contribution of (i, j) is 1. The cases $B_i = B_j$ and $C_i = C_j$ can be argued in the same way.

Using this observation, the total sum of contributions over all pairs (i, j) can be expressed as

$$\begin{aligned}&(\text{total contribution within group } P)\\ &+ (\text{total contribution within group } Q)\\ &+ (\text{number of permutations from group } P) \times (\text{number of permutations from group } Q).\end{aligned}$$

Define $dp[x][y][z][k]$ as the number of sequences consisting only of permutations from group P such that the permutation $(1, 2, 3)$ appears x times, $(2, 3, 1)$ appears y times, $(3, 1, 2)$ appears z times, and the total contribution is k. This DP table can be computed in $O(N^5)$ time. An analogous DP can be computed in the same way for group Q.

Suppose that in D, the numbers of $(1, 2, 3)$, $(2, 3, 1)$, and $(3, 1, 2)$ are x, y, and z, respectively, and that the total contribution within group P is k. Then the following quantity contributes to the answer:

$$\binom{N}{x+y+z} \times dp[x][y][z][k] \times dp[X-x][Y-y][Z-z][K-k-(x+y+z)(N-x-y-z)].$$

By summing this value over all x, y, z, k satisfying $0 \le x \le X$, $0 \le y \le Y$, $0 \le z \le Z$, and $0 \le k \le K$, the final answer can be computed in $O(N^5)$ time.

Problem D. Grid Path Tree

The cases where $N = 1$, $M = 1$, or $k = 1$ can be handled easily, so we omit them. From now on, we assume that $N \neq 1$, $M \neq 1$, and $k \neq 1$.

A Slow Solution

First, we describe a method to compute the answer in time complexity $O(N + M)$ for a fixed value of k.

There is a one-to-one correspondence between trees and paths that move from the top-left corner to the bottom-right corner of an $N \times M$ grid, using only right moves and down moves.

Now, impose the additional condition $(a_2, b_2) = (2, N + 1)$ on the tree. Under this condition, such trees are in one-to-one correspondence with sequences of positive integers

$$A = (A_1, A_2, \ldots, A_{|A|})$$

that satisfy the following:

$$\begin{aligned}N - 1 &= A_1 + A_3 + \cdots,\\ M - 1 &= A_2 + A_4 + \cdots.\end{aligned}$$

This corresponds to a path that moves down A_1 times, then right A_2 times, then down A_3 times, and so on.

For a tree represented by a sequence A, the number of pairs (i, j) such that $\mathrm{dist}(i, j) = k$ (for $k \neq 2$) can be written as

$$\sum_{t=1,\ldots} A_t A_{t+k-2}.$$

When k is odd, the total sum of the above value over all sequences A with $|A| = L$ is given by

$$\max(0, L + 2 - k) \binom{N-1}{\lfloor \frac{L+1}{2} \rfloor} \binom{M-1}{\lfloor \frac{L}{2} \rfloor}.$$

When k is even, it can be expressed as

$$\max\left(0, \left\lfloor \frac{L+1}{2} \right\rfloor + 1 - k\right) \binom{N}{\lfloor \frac{L+1}{2} \rfloor + 1} \binom{M-2}{\lfloor \frac{L}{2} \rfloor - 1} + \max\left(0, \left\lfloor \frac{L}{2} \right\rfloor + 1 - k\right) \binom{N-2}{\lfloor \frac{L+1}{2} \rfloor - 1} \binom{M}{\lfloor \frac{L}{2} \rfloor + 1}.$$

The same formula is also valid when $k = 2$.

Therefore, once k and $|A|$ are fixed, the answer can be computed in $O(1)$ time. As a result, for a fixed k, the total answer can be obtained in $O(N + M)$ time.

When $(a_2, b_2) = (1, N + 2)$, the problem can be solved in the same way by swapping N and M.

Full-Score Solution

For any $2 \le k \le N + M - 5$, the value

$$ans_{k+4} - 2 \cdot ans_{k+2} + ans_k$$

can be computed in $O(1)$ time, assuming binomial coefficients are precomputed.

Therefore, after computing $ans_2, ans_3, ans_4, ans_5$ directly using the method described above, the remaining values can be derived appropriately using this fact.

In this way, all values ans_k for arbitrary k can be computed in total time complexity $O(N + M)$, which is sufficiently fast to solve this problem.

Problem E. Max Twice Subsequences

$O(|B|)$-time solution per query

First, let us establish how to compute the answer in the case $Q = 1$. Let $n := |B|$, and suppose we are given $B = (B_1, B_2, \ldots, B_n)$.

Assume that there exists a subsequence that appears at least twice, and take two distinct index sequences $(i_1, i_2, \ldots, i_k)$ and $(j_1, j_2, \ldots, j_k)$ that realize it. We must have $k \ge 1$, and there exists some p $(1 \le p \le k)$ such that $i_p \ne j_p$. Without loss of generality, assume $i_p < j_p$. In this case, $(i_1, i_2, \ldots, i_{p-1}, i_p, j_{p+1}, j_{p+2}, \ldots, j_k)$ and $(i_1, i_2, \ldots, i_{p-1}, j_p, j_{p+1}, j_{p+2}, \ldots, j_k)$ are also distinct index sequences that produce the same subsequence. Therefore, when we look for distinct index sequences that give a subsequence which appears at least twice, it suffices to consider only those pairs of index sequences that differ in exactly one position.

Now, let us denote such distinct index sequences by $(i_1, i_2, \ldots, i_a, l, j_1, j_2, \ldots, j_b)$ and $(i_1, i_2, \ldots, i_a, r, j_1, j_2, \ldots, j_b)$. Here, we have

$$1 \le i_1 < i_2 < \cdots < i_a < l < r < j_1 < j_2 < \cdots < j_b \le n.$$

Since $l \ne r$ and $B_l = B_r$, the sequence B must contain duplicate elements. More concretely, we can argue as follows: take the smallest s $(1 \le s \le n)$ such that the suffix $(B_s, B_{s+1}, \ldots, B_n)$ contains no duplicate elements. Then we must have $l < s$, and if $s = 1$ no good sequence exists.

Next, consider choosing an index sequence $(i_1, i_2, \ldots, i_a, l, r, j_1, j_2, \ldots, j_b)$ that satisfies the conditions, deciding its indices from the front. If the first index is i, then from $l < s$ we obtain

$$B_i = \max_{1 \le t < s} B_t.$$

Since we may choose the subsequence greedily from the front, we must have

$$i = \min(\operatorname*{argmax}_{1 \le t < s} B_t).$$

At this point, we have the following two options:

1. Fix $a \ge 1$ and set $i_1 \leftarrow i$. This is possible when $i < s - 1$.

2. Fix $a = 0$ and set
$$l \leftarrow i, \quad r \leftarrow \min\{\, t > i \mid B_t = B_i \,\}.$$
This is possible when $\{\, t > i \mid B_t = B_i \,\} \neq \emptyset$.

In either case, noting that $\{\, t \geq s \mid B_t = B_{s-1} \,\} \neq \emptyset$, we can choose the remaining indices appropriately to construct index sequences satisfying the conditions. When both of the above choices are possible, we must decide which one to take. Since the first element $C_1 = B_i$ of the subsequence has already been fixed, the next quantity to maximize is C_2. For each option, C_2 is given as follows:

1. If we choose option 1: we are effectively solving the same problem on $B \leftarrow B(i, n]$, so
$$C_2 = \max_{i<t<s} B_t.$$

2. If we choose option 2: we will greedily choose the maximum subsequence from indices greater than r, hence
$$C_2 = \max_{r<t\leq n} B_t.$$

If the two candidates for C_2 are different, we should take the option that gives the larger value. If they are the same, we should actually choose option 1. The reason is that, if option 2 is chosen, and the maximal subsequence taken from indices after r is X, then by choosing option 1 instead, and setting
$$a' = 1, \quad i'_1 = i, \quad l' = \min(\operatorname*{argmax}_{i<t<s} B_t), \quad r' = \min(\operatorname*{argmax}_{r<t\leq n} B_t),$$
we can obtain the same subsequence X. Here, we can see that $l' < r'$ as follows: first, if $s \leq r'$, this is clear from $l' < s$. If $r' < s$, then by the definitions of r and r' we have $i < r < r' < s$, and combining the maximality of B_i on $[1, s)$ with $B_i = B_r$, we get
$$\max_{i<t<s} B_t = B_r.$$
Comparing this with the definition of l', we obtain $l' \leq r$, and thus $l' \leq r < r'$, which gives the desired inequality.

We have now obtained the optimal strategy for choosing the index sequence from the front. In the implementation, we precompute

- s,
- $\mathrm{tolim}_i = \min(\operatorname{argmax}_{i<t<s} B_t)$,
- $\mathrm{toend}_i = \min(\operatorname{argmax}_{i<t\leq n} B_t)$,
- $\mathrm{nxt}_i = \min\{\, t > i \mid B_t = B_i \,\}$,

and whenever the two options arise, we compare the next values (the candidates for C_2) and decide which option to take. If we choose option 1, we continue with the same procedure; if we choose option 2, we can greedily take the maximum subsequence from indices greater than r. All of this can be done in $O(|B|)$ time.

The following is an implementation in pseudocode. Input, preprocessing, and obtaining the rolling hash are omitted.

```
# B[1],...,B[m] have been processed
m = 0
ans = []
while true :
    i = tolim[m]
    ans.append(B[i])
    if i == s-1 :
        m = nxt[i]
        break
    if nxt[i] != infty && B[tolim[i]] < B[toend[nxt[i]]] :
        m = nxt[i]
        break
    m = i
while m < n :
    i = toend[m]
    ans.append(B[i])
    m = i
```

Speeding it up

Now consider applying the above $O(|B|)$ solution to $B = (A_{L_i}, A_{L_i+1}, \ldots, A_{R_i})$ defined from a query (L_i, R_i). We see that $s, \mathrm{tolim}, \mathrm{toend}$ appearing in the solution depend heavily on R_i, and barely on L_i. Therefore, we will read all queries in advance and process them in ascending order of R_i.

Assume that $s, \mathrm{tolim}, \mathrm{toend}, \mathrm{nxt}$ have been computed for $B = (A_1, A_2, \ldots, A_R)$. Suppose $L < s$. If we rewrite the above pseudocode so that it works for a query (L, R), it becomes:

```
# A[L],...,A[m] have been processed
m = L-1
ans = []
while true :
    i = tolim[m]
    ans.append(A[i])
    if i == s-1 :
        m = nxt[i]
        break
    if nxt[i] != infty && A[tolim[i]] < A[toend[nxt[i]]] :
        m = nxt[i]
        break
    m = i
while m < R :
    i = toend[m]
    ans.append(A[i])
    m = i
```

Assume that we can obtain, in $O(\log N)$ time, the value of i at which the `break` happens in this pseudocode; denote this value by $i_{L,R}$. Then the lexicographically largest good sequence has the following structure:

- Among the indices i that appear in tolim, list in ascending order those that satisfy $L \leq i < i_{L,R}$, and call this sequence $(i_1, i_2, \ldots, i_a)$.
- Among the indices j that appear in toend, list in ascending order those that satisfy $\mathrm{nxt}_{i_{L,R}} < j \leq R$, and call this sequence $(j_1, j_2, \ldots, j_b)$.

- The lexicographically largest good sequence is exactly the concatenation of $(A_{i_1}, A_{i_2}, \ldots, A_{i_a})$, $(A_{i_{L,R}})$, and $(A_{j_1}, A_{j_2}, \ldots, A_{j_b})$ in this order.

Therefore, if we prepare segment trees that store the rolling hash of the single-element sequence (A_i) at each index i that appears in tolim or toend, respectively, we can obtain the answer for each L in $O(\log N)$ time.

Next, note that the `if` condition in the pseudocode does not depend on L, so $i_{L,R}$ can be characterized as follows. Fix R, and let $L < s$ be arbitrary. Among all indices that appear in tolim, only those that satisfy at least one of the `if` conditions in the pseudocode can be candidates for $i_{L,R}$. Let I_R be the set of such indices. Then $i_{L,R}$ is the smallest element of I_R that is at least L. By managing I_R with a balanced binary search tree such as `std::set`, we can obtain $i_{L,R}$ in $O(\log N)$ time.

The remaining task is to update $s, \mathrm{tolim}, \mathrm{toend}, \mathrm{nxt}$ when R is increased by 1, and to perform the corresponding updates to the segment trees with rolling hashes and to I_R.

First, s is monotonically non-decreasing as R increases. Both tolim and toend change as s and R increase, but the sets of indices that appear in them can be maintained using stacks. When R increases by 1, only up to one entry of nxt changes. Up to this point, ignoring the updates to I_R, all required updates can be performed in a total of $O(N \log N)$ time.

Let S be the set of indices that appear in tolim; then $I_R \subseteq S$. We will update I_R as follows. By defining min and max appropriately on the empty set and adding sentinels to A, we can rewrite the `if` used for the `break` so that it is unified into a single condition `A[tolim[i]] < A[toend[nxt[i]]]`. For readability, we will also write A_* as $A[*]$.

- Any index i that is popped from the stack for tolim will never again be a member of I_R.
- For $i \in S$, define
$$d_i = A[\mathrm{toend}_{\mathrm{nxt}_i}] - A[\mathrm{tolim}_i].$$
Then the condition for i to be an element of I_R is $d_i > 0$.
- When nxt is updated, at most one index $i \in S \setminus I_R$ (namely, the one with $\mathrm{nxt}_i = R$) has its d_i increased and newly becomes an element of I_R.
- When tolim is updated, at most one index $i \in S$ (the top of the stack) has its d_i decreased and ceases to be an element of I_R.
- When toend is updated, for those $i \in S \setminus I_R$ whose nxt_i falls into a certain interval, d_i increases and all but at most one of them newly become elements of I_R. In particular:
  - If, for an index i in that interval, nxt_i is not the top of the stack for toend, then $A[\mathrm{nxt}_i] < A[\mathrm{toend}_{\mathrm{nxt}_i}]$ holds. Moreover, since $i \in S$, we have $A[i] \geq A[\mathrm{tolim}_i]$. Thus,
  $$d_i = A[\mathrm{toend}_{\mathrm{nxt}_i}] - A[\mathrm{tolim}_i] > A[\mathrm{nxt}_i] - A[i] = 0,$$
  and therefore such i becomes an element of I_R.

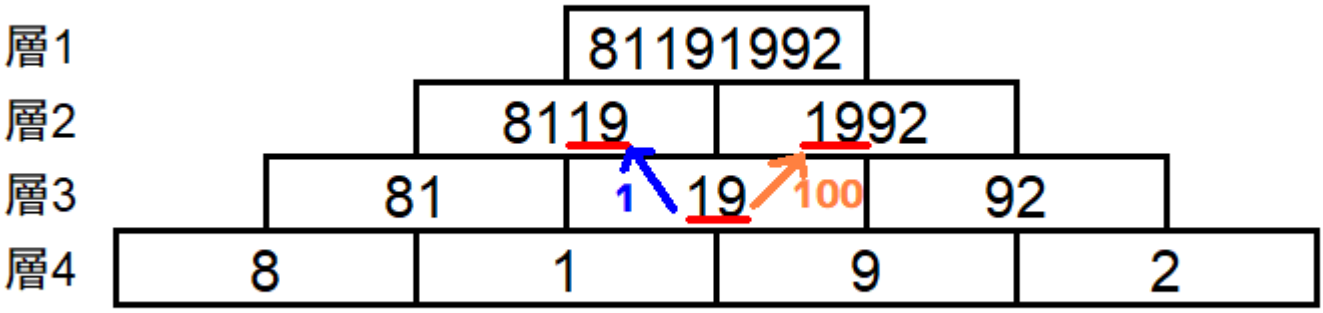
Since the number of times an element of I_R ceases to belong to I_R can be bounded by $O(N)$ in total, the overall complexity is not worsened even if we scan the relevant part of $S \setminus I_R$ naively in the last item. For example, we can store $S \setminus I_R$ in a balanced binary search tree such as `std::set` keyed by nxt_i, and, for a given range, simply traverse all elements in that range.

By carefully implementing these updates with sufficient attention to the order in which they are applied, we can solve this problem. The time complexity of this solution is $O((N + Q) \log N)$.

Problem F. Decimal Pyramid

Let us consider computing the contribution of each block $C_{N,i}$ in layer N (the bottom layer) to $C_{1,1}$.

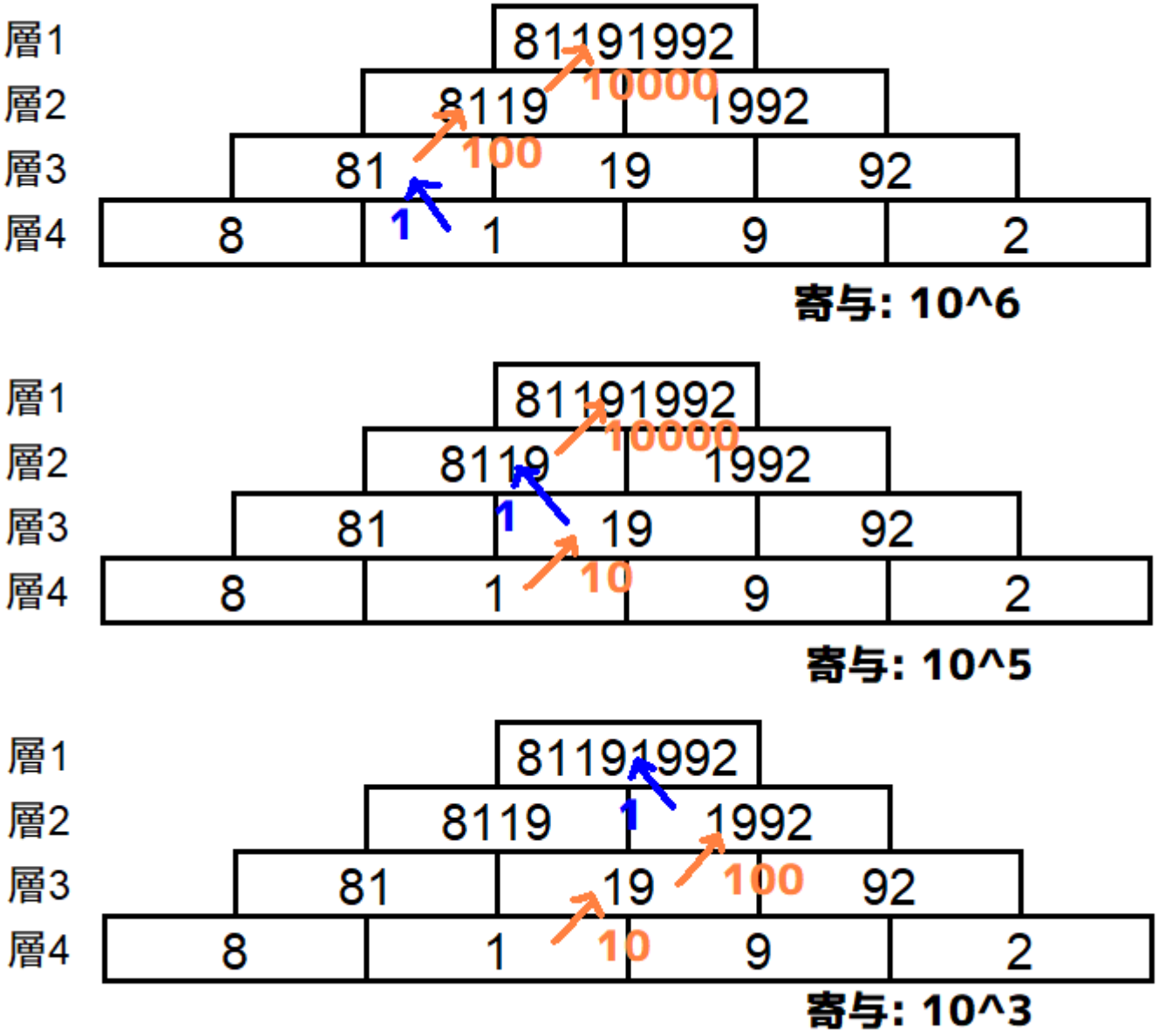
In general, a block $C_{i,j}$ affects two blocks in layer $i-1$: $C_{i-1,j-1}$ and $C_{i-1,j}$. In this process, it contributes to $C_{i-1,j-1}$ with coefficient 1, and to $C_{i-1,j}$ with coefficient $10^{2^{N-i}}$.



When $C_{i,j}$ contributes to layer $i-1$, we can think of it as having two choices: contributing either to $C_{i-1,j-1}$ or to $C_{i-1,j}$. We call the former choice "moving to the upper left," and the latter choice "moving to the upper right."

By repeatedly making contributions of the form "moving to the upper left" or "moving to the upper right," we will eventually reach $C_{1,1}$. It can be seen that, for the block $C_{N,i}$, the number of times it moves "to the upper left" is exactly $i-1$, and the number of times it moves "to the upper right" is exactly $N-i$.

Conversely, any way of moving that consists of exactly $i-1$ moves to the upper left and $N-i$ moves to the upper right represents a valid contribution from $C_{N,i}$ to $C_{1,1}$.



Now, let us express this using polynomials. Suppose that every time we "move to the upper right," we multiply by x. Then, the total contribution from $C_{N,i}$ to $C_{1,1}$ can be written as

$$C_{N,i}[x^{N-i}] \prod_{t=1}^{N-1} (1 + 10^{2^{t-1}} x).$$

To compute this for all i, it suffices to expand the polynomial $\prod_{t=1}^{N-1}(1 + 10^{2^{t-1}} x)$. This can be achieved in $O(N \log^2 N)$ time by using polynomial merging techniques (divide and conquer).

Problem G. Don't Make Zero

In fact, a sequence of length X of the form $(-(N-X-1), -(N-X-2), \ldots, -A, A+1, \ldots, N-1, N)$ is always a good sequence.

If, in a subsequence, the number of positive integers is greater than or equal to the number of negative integers, then the sum of the subsequence is positive. If the number of positive integers is less than the number of negative integers, then the sum is negative.

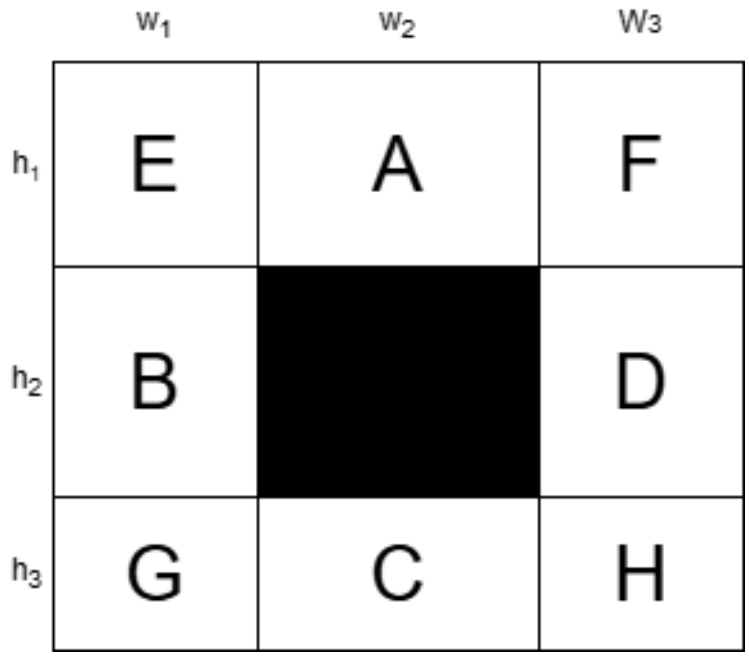
Therefore, when the sign is -, we can append $-(N-X-1), -(N-X-2), \ldots$ in order, and when the sign is +, we can append $N, N-1, \ldots$ in order. By doing so, we can always construct a good sequence.

Problem H. Akari Counting

Let $h_1, h_2, h_3, w_1, w_2, w_3$ be the values defined as follows.

- $h_1 = A - 1$
- $h_2 = B - A + 1$
- $h_3 = H - B$
- $w_1 = C - 1$
- $w_2 = D - C + 1$
- $w_3 = W - D$

We divide the grid into eight regions A, B, C, D, E, F, G, H as shown in the figure below, and let a, b, c, d, e, f, g, h denote the numbers of lights placed in each region, respectively. (Note that from this point on, A, B, C, D refer to region names.)



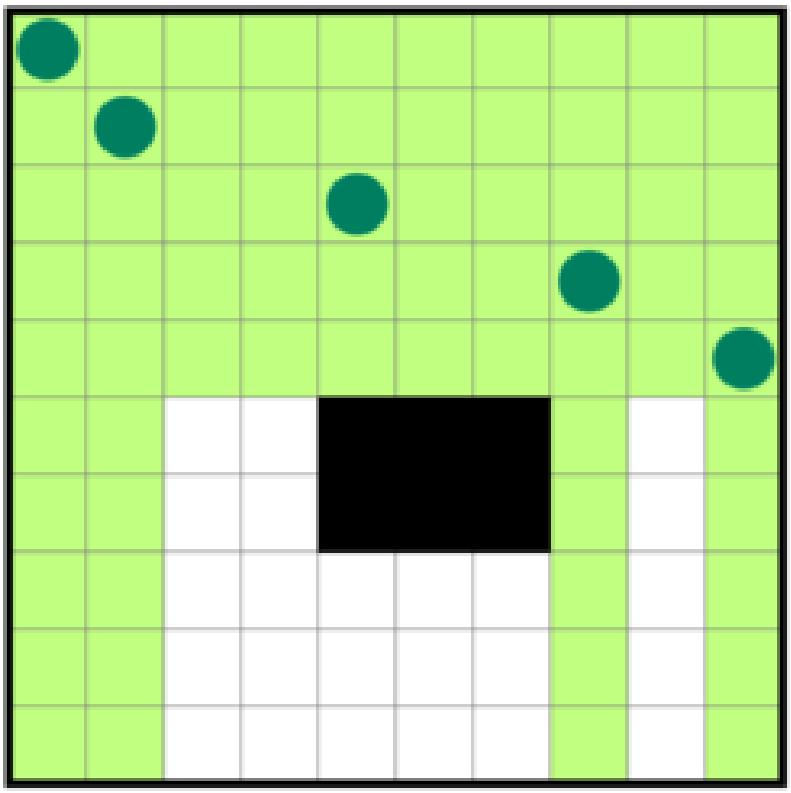
Conditions for Illuminating All White Cells

Region A is illuminated only by lights placed in regions A, E, and F. Therefore, in order to illuminate all cells in region A, one of the following conditions must hold:

- $a + e + f = h_1$
- $a + e + f \neq h_1$ and $a = w_2$

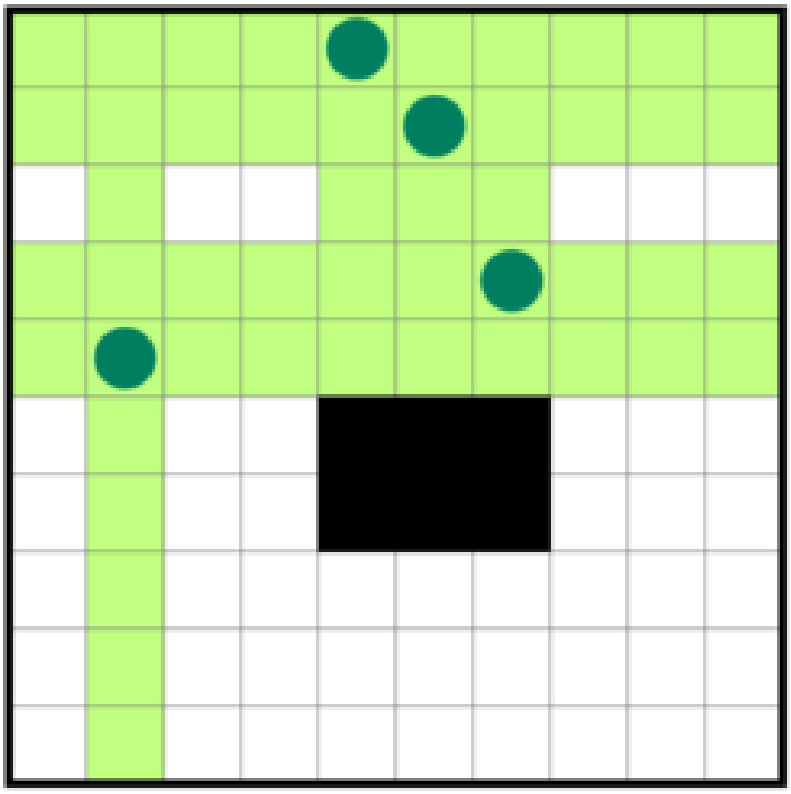
Case $a + e + f = h_1$

All cells in regions A, E, and F are illuminated.



Case $a = w_2$ (and $a + e + f \neq h_1$)

All cells in region A are illuminated, but some cells in regions E and F remain unilluminated. These remaining cells must be illuminated vertically by lights placed in other regions.



Similarly, considering the illumination of all white cells in region B, one of the following must hold:

- $b + e + g = w_1$
- $b + e + g \neq w_1$ and $b = h_2$

Next, consider illuminating all white cells in region E.

Suppose there exists a row among the top h_1 rows with no light placed, and also a column among the leftmost w_1 columns with no light placed. Then the white cell at the intersection of this row and column is not illuminated by any light, violating the condition. Taking this into account, in order to illuminate all white cells in region E, at least one of the following must hold:

- $a + e + f = h_1$
- $b + e + g = w_1$

Applying the same reasoning to regions F, G, and H, we see that in order to illuminate all white cells in regions E, F, G, and H, one of the following conditions must hold:

- $a + e + f = h_1$ and $d + g + h = h_3$
- $b + e + g = w_1$ and $d + f + h = w_3$

Based on these observations, we proceed to solve the problem.

Counting

First, we partially fix a placement of lights that satisfies the conditions for illuminating regions A, B, C, and D. Then, we determine the remaining placements so that the conditions for illuminating regions E, F, G, and H are also satisfied.

To illuminate all cells in region A, one of the following must hold:

- $a + e + f = h_1$
- $a + e + f \neq h_1$ and $a = w_2$

Let $X_0[i]$ be the number of ways to place lights in region A such that $a + e + f = h_1$ and $e + f = i$. This value is given by $\binom{w_2}{h_1 - i} \frac{h_1!}{i!}$.

Next, consider the case where $a + e + f \neq h_1$, $a = w_2$, and $e + f = i$. We count the number of pairs consisting of a placement of lights in region A, and a set of rows in regions E and F that contain at least one light. This number is $\binom{h_1 - w_2}{i} \cdot \frac{h_1!}{(h_1 - w_2)!}$, and we denote it by $X_1[i]$.

Applying the same computation to regions C, G, and H, we define $Y_0[i]$ and $Y_1[i]$.

We then define $P_0[i]$ and $P_1[i]$ as follows:

- $P_0[i] = \sum_{j+k=i} X_0[j] \cdot Y_0[k]$
- $P_1[i] = \sum_{j+k=i} \big(X_0[j] \cdot Y_1[k] + X_1[j] \cdot Y_0[k] + X_1[j] \cdot Y_1[k]\big)$

The meanings of $P_0[i]$ and $P_1[i]$ are as follows:

- $P_0[i], P_1[i]$: the number of ways in which the white cells with lights in regions A and D, and the set of rows containing lights in regions E, F, G, and H, are fixed, and the size of the row set is i
- $P_0[i]$: both $a + e + f = h_1$ and $d + g + h = h_3$ hold
- $P_1[i]$: at least one of $a + e + f \neq h_1$ or $d + g + h \neq h_3$ holds

Similarly, we perform the same analysis for columns and define $Q_0[i]$ and $Q_1[i]$ as follows:

- $Q_0[i], Q_1[i]$: the number of ways in which the white cells with lights in regions B and C, and the set of columns containing lights in regions E, F, G, and H, are fixed, and the size of the column set is i
- $Q_0[i]$: both $b + e + g = w_1$ and $c + f + h = w_3$ hold
- $Q_1[i]$: at least one of $b + e + g \neq w_1$ or $c + f + h \neq w_3$ holds

Now consider placing exactly i lights in regions E, F, G, and H. Once the sets of rows and columns to be used are fixed, the number of ways to place the lights is $i!$.

Therefore, the final answer can be expressed as:

- $\sum_i \big(P_0[i] \cdot Q_0[i] + P_0[i] \cdot Q_1[i] + P_1[i] \cdot Q_0[i]\big) \cdot i!$

The bottleneck of the computation lies in evaluating P_0, P_1, Q_0, and Q_1. By using convolution, this can be done in time complexity $O\big((H + W) \log(H + W)\big)$.

Problem I. Subgrid Connected Components

Analysis

A naive approach that counts the number of connected components directly would take $O(N^2Q)$ time, which is far too slow. Instead, we observe that this graph has a grid structure and look for useful structural properties.

First, note that this grid forms a planar graph. Therefore, it has several regions enclosed by edges (we also regard the unbounded outer area as a region).

If we remove one edge that encloses a region, the number of connected components does not change, while the number of regions decreases by one. By repeating this process, if the original number of regions is r, we can remove $r-1$ edges to make the number of regions equal to 1.

Next, if we add one edge between two different connected components (which is possible because the graph is grid-shaped), the number of regions does not change, while the number of connected components decreases by one. Thus, if the original number of connected components is a, we can add $a-1$ edges to make the graph connected.

The resulting graph is connected and has no cycles, i.e., it is a tree. Therefore, if the number of vertices is v, the number of edges must be $v-1$.

Letting e be the original number of edges, we obtain

$$e-(r-1)+(a-1)=v-1,$$

which can be rearranged as

$$a=v-e+r-1.$$

This formula can also be understood as an instance of **Euler's polyhedron theorem**.

From this observation, we see that each query can be answered by counting three quantities: **the number of vertices, the number of edges, and the number of regions**.

The number of vertices is, as stated in the problem, $((D_i-U_i+2)/2)\times((R_i-L_i+2)/2)$. The number of edges can be computed using prefix sums of edges, allowing $O(N^2)$ preprocessing time and $O(1)$ time per query. The difficult part is counting the **number of regions**.

Counting the Number of Regions

To handle this, we use the concept of the **dual graph**. In the dual graph, each region of the original graph corresponds to a vertex. Consider the points represented by even coordinates (x,y). Two such points are considered to belong to the same region if they are adjacent and there is no edge between them. Moreover, if a point is connected to the outside, it is regarded as belonging to the outer region.

For a subgrid given by a query, any closed cycle that is entirely contained in the subgrid remains a closed region within that subgrid as well.

The behavior of regions changes only when the original grid has a cycle, but the subgrid contains only part of that cycle, causing the region to connect to the outside. Such cases occur only along the **outside boundary of the subgrid**, and there are at most $4N$ such points.

To answer the question "How many cycles does the subgrid contain?" efficiently, we assign **one representative point to each region**.

This approach works as follows:

- If the subgrid fully contains a cycle, it must include the representative point of that region.
- If the subgrid contains no cycle at all, it contains none of the representative points.

- If the subgrid contains only part of a cycle, the representative point may or may not be included. However, by exhaustively checking the parts that connect to the outside along the boundary, we can remove all representative points corresponding to regions that are connected to the outside. As noted above, we need to examine only $4N$ points.

In this way, we can count the number of cycles accurately. By preprocessing the prefix sums of representative points for all regions in $O(N^2)$ time, we can answer each query in $O(N)$ time.

Overall, this yields an algorithm with $O(N^2)$ preprocessing time and space, and $O(N)$ time per query.

Problem J. Divide Polygon

Correspondence Between Polygon Dissections and Trees

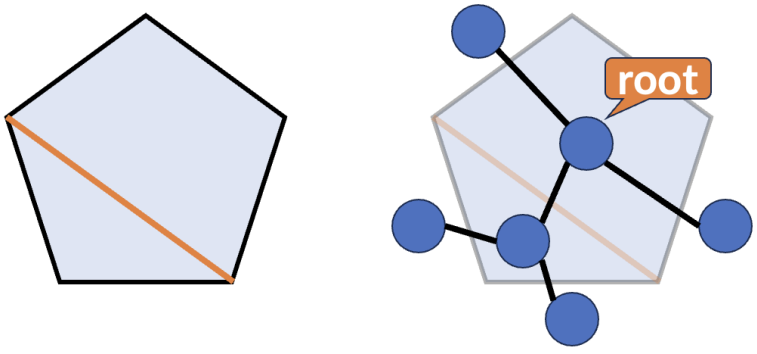
Ways to dissect a polygon are in one-to-one correspondence with rooted trees that satisfy all of the following conditions. The vertices are unlabeled, but the children of each vertex are ordered.

- There are exactly $N-1$ leaves.
- There are exactly $k+1$ non-leaf vertices, including the root.
- For a non-leaf vertex whose number of children is c, the value $c+1$ is contained in S.

One possible way to establish the correspondence between dissections and trees is as follows. Here, let segment i denote the edge connecting vertex i and vertex $i+1$, with segment N denoting the edge connecting vertex N and vertex 1.

- For each resulting polygon in the dissection, associate one non-leaf vertex.
- For each segment i $(1 \leq i \leq N-1)$, associate one leaf.
- Connect two vertices by an edge if and only if the corresponding polygons share the same diagonal.
- Connect the corresponding vertices if and only if a polygon contains segment i $(1 \leq i \leq N-1)$.
- Choose the vertex corresponding to the polygon containing segment N as the root.
- Order the children so that the leaves appear from left to right in the order corresponding to segments $1, 2, \ldots, N-1$.

The figure below shows an example for $N=5$ and $k=1$, where the polygon is dissected by drawing the diagonal connecting vertices 2 and 4, together with the corresponding tree.

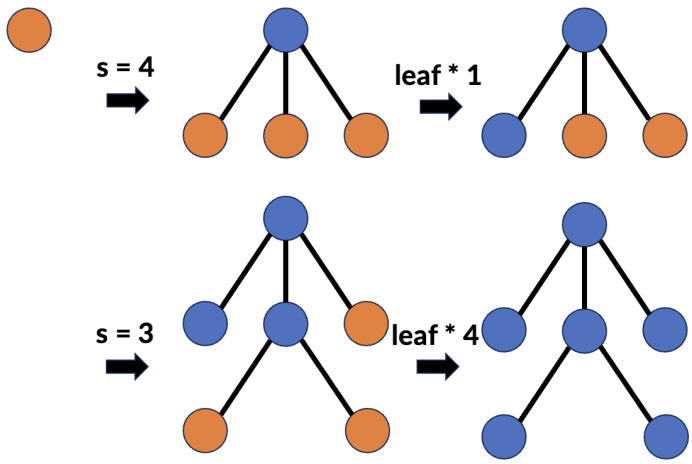


Correspondence Between Trees and Sequences

The tree described above can be constructed by starting from a tree consisting of a single root and performing the following operation exactly $N + k$ times.

- Choose and perform one of actions 1 or 2. Action 1 must be performed exactly $N - 1$ times in total.
  - Action 1: Select the leftmost leaf and fix it as a leaf.
  - Action 2: Choose $s \in S$, and for the leftmost vertex that is not yet fixed as a leaf, add $s - 1$ children to it.

For example, the tree corresponding to the pentagon dissection shown earlier can be constructed by the following sequence of operations.



We associate the sequence of operations that constructs the tree with an integer sequence A as follows.

- Start with A being empty, and scan the operations from left to right, modifying A as below.
  - When Action 1 is performed, append -1 to the end of A.
  - When Action 2 is performed, append $s - 2$ to the end of A.

For the example shown in the figure, this corresponds to $A = (2, -1, 1, -1, -1)$.

The final sequence A satisfies all of the following conditions if and only if the sequence of operations is valid.

- $|A| = N + k$
- Each element a in A satisfies either $a = -1$ or $(a + 2) \in S$
- The sum of all elements in A is -1
- For any $1 \leq i < N + k$, we have $A_1 + A_2 + \cdots + A_i \geq 0$

Therefore, the number of valid sequences A satisfying all of the above conditions is exactly the answer.

Counting the Sequences

If we ignore the last condition, the number of valid sequences can be written using $f(x) = \sum_{s \in S} x^{s-2}$ as $\binom{N+k}{k+1}[x^{N-2}]f(x)^{k+1}$. Let this value be denoted by C_k.

Even after adding the last condition, by applying Bertrand's ballot theorem, we can see that the answer becomes $\dfrac{C_k}{N+k}$.

More concretely, this can be shown by establishing a one-to-one correspondence between the following two sets:

- Pairs (A, x), where A satisfies all conditions including the last one, and x is an integer with $0 \le x < N + k$
- Sequences B that satisfy all conditions except the last one

The correspondence is defined by taking B to be the sequence obtained by cyclically shifting A to the left by x positions. For example, the pair $A = (2, -1, 1, -1, -1)$ and $x = 3$ corresponds to $B = (-1, -1, 2, -1, 1)$.

Thus, for each $0 \le k \le N - 3$, it suffices to compute $\dfrac{C_k}{N+k}$.

The bottleneck of the computation lies in evaluating C_k for all $0 \le k \le N - 3$. This problem is known as **Power Projection**, and it can be solved in time complexity $O(N \log^2 N)$.

Therefore, the answer to this problem can also be computed within the same time complexity.

Problem L. Colorful Quadrilateral

0. Definitions

- For two 2D vectors $\overrightarrow{u}, \overrightarrow{v}$, define the cross product as
$$\overrightarrow{u} \times \overrightarrow{v} := u_x v_y - v_x u_y.$$
- For a 2D vector $\overrightarrow{v}$, define its 90° counterclockwise rotation as
$$\mathrm{rot}(\overrightarrow{v}) := (-v_y, v_x).$$
- For a 2D vector $\overrightarrow{v}$, define $\mathrm{top4}(\overrightarrow{v})$ as the following set of four points: For each color c, pick the point of color c whose inner product with $\overrightarrow{v}$ is maximized. Among all selected points, take the four points with the largest inner products with $\overrightarrow{v}$.

1. Properties of the Optimal Quadrilateral

Let the optimal quadrilateral be $ABCD$. Consider the situation where only two points A and C are known. Then points B and D can be determined as follows:

1. Ignore points of the same colors as A and C.
2. Let $\overrightarrow{v} := \mathrm{rot}(\overrightarrow{AC})$. Choose the point with the minimum inner product with $\overrightarrow{v}$ as B, and the point with the maximum inner product as D.
3. If B and D have the same color, replace one of them with the minimum/maximum point among the remaining colors (whichever yields the larger area).

4. If the quadrilateral is simple (non-self-intersecting), its area is

$$\frac{1}{2}(\overrightarrow{AC} \times \overrightarrow{BD}).$$

If it is self-intersecting, one can reorder A, B, C, D to form a simple quadrilateral with larger area, so using $\frac{1}{2}(\overrightarrow{AC} \times \overrightarrow{BD})$ is acceptable.

From steps 2–3 we have:

- For each color c, let P_c be the set of points of color c. Any point not on the convex hull of P_c will never be chosen in the answer.
- For every chosen point X, there exists some direction $\overrightarrow{v}$ such that $X \in \text{top4}(\overrightarrow{v})$.

2. $O(N^3)$ Solution

1. Enumerate all pairs (A, C).
2. Compute B and D in $O(N)$ time by taking the minimum/maximum inner products with $\text{rot}(\overrightarrow{AC})$.

3. Fast Query: Maximum Inner Product in a Color Range

Using a segment-tree-like structure, we can find, in $O((\log N)^2)$ time, the point of colors in $[L, R)$ whose inner product with direction $\overrightarrow{v}$ is maximized.

Each node of the segment tree stores the convex hull of the points whose colors are in that interval. Given $O(\log N)$ nodes and $O(\log N)$ time per hull to query the max inner product, we obtain $O((\log N)^2)$ per query.

If the directions $\overrightarrow{v}$ are sorted, we can precompute the breakpoints where the maximizing vertex changes and reduce the cost to $O((N + Q) \log N)$.

4. $O(N^2 \log N)$ Algorithm

1. Enumerate all $\overrightarrow{AC}$ and sort them by polar angle in $O(N^2 \log N)$ time.
2. For each $\overrightarrow{AC}$, find B, D of colors different from A and C using the data structure in amortized $O(\log N)$ time.
3. If B and D have the same color, reselect one to maximize the area.

5. Case of Exactly Four Colors

Assume the four colors in the quadrilateral are $1, 2, 3, 4$ in order for A, B, C, D. Let the direction of $\overrightarrow{AC}$ be $\overrightarrow{v}$. Then:

- For color 2, the point maximizing $-\text{rot}(\overrightarrow{v})$ is B.
- For color 4, the point maximizing $\text{rot}(\overrightarrow{v})$ is D.

Note: It is **incorrect** to say:

- For color 1, the point maximizing $-\overrightarrow{v}$ is A
- For color 3, the point maximizing $\overrightarrow{v}$ is C

Hence the problem can be solved in $O(N \log N)$ time:

1. Sort points by x-coordinate.
2. Compute convex hulls of each color and discard interior points.
3. Fix A of color 1 and enumerate C of color 3.
4. Rotate $\overrightarrow{v}$ over $[0, 360^\circ)$ and enumerate candidates for $\overrightarrow{BD}$. This corresponds to taking the Minkowski sum of -1 times the hull of color 2 and the hull of color 4, which takes $O(N)$ time.
5. Similarly enumerate candidates for $\overrightarrow{AC}$.
6. Using two-pointer scanning over sorted directions, find the $\overrightarrow{BD}$ that maximizes $|\overrightarrow{AC} \times \overrightarrow{BD}|$.

6. Randomized Approach

If there are more than four colors, randomly map the colors to $\{1, 2, 3, 4\}$ and run the above algorithm. With probability $1/32$, the largest-area quadrilateral is preserved. Repeating yields high success probability.

Because all repeatable steps take $O(N)$, the complexity is $O(N \log N + RN)$ for R repetitions.

Since the problem contains multiple test cases, a purely randomized solution may not achieve AC, but combining with the $O(N^3)$ method for small N may work.

7. Number of Distinct $\mathrm{top4}(\overrightarrow{v})$

We study how many different $\mathrm{top4}(\overrightarrow{v})$ arise as $\overrightarrow{v}$ rotates through 360°.

When colors are all distinct, the number is $O(N)$ due to results on order-k Voronoi diagrams:

$$O(k(N-k)) \quad \text{with } k = 4 \text{ or } k = N - 4.$$

Reference: Lee, Der-Tsai. "On k-nearest neighbor Voronoi diagrams in the plane."IEEE Transactions on Computers 100.6 (1982): 478–487.

For general colored setting, the number is still $O(N)$. The essential idea:

If we can show that the possible sets of colors appearing in $\mathrm{top4}(\overrightarrow{v})$ are at most $7N$, then the number of top4 types is at most $8N$.

Let p be the 4th-ranked point in some $\mathrm{top4}(\overrightarrow{v})$. Such a configuration corresponds to a half-plane whose boundary contains p and which contains exactly four colors.

For each p, the number of such possible color sets is at most 7 (or often 4 if p's color appears elsewhere). Thus the total number is $O(N)$.

Detailed reasoning: Consider rotating a half-plane around p. If other points of the same color exist and must be excluded, rotation is limited to $\leq 180^\circ$, and the number of candidate color sets is ≤ 4. If full 360° rotation is allowed, the number is ≤ 7.

8. Near-Linear Algorithm

We propose the following:

1. Compute convex hulls of each color and discard interior points.
2. Rotate $\overrightarrow{v}$ through 360° and enumerate candidates for A and C from top4($\overrightarrow{v}$) and top4($-\overrightarrow{v}$). The number of candidates is $O(N)$.
3. Sort all $\overrightarrow{AC}$ candidates by angle.
4. For each $\overrightarrow{AC}$, compute B and D from top4($\overrightarrow{v}$) and top4($-\overrightarrow{v}$) where $\overrightarrow{v} = \mathrm{rot}(\overrightarrow{AC})$.

The total time is $f(N) + O(N \log N)$, where $f(N)$ is the time to enumerate all top4($\overrightarrow{v}$).

9. Enumerating top4($\overrightarrow{v}$)

We describe an $O(N(\log N)^2)$ method.

In a segment tree over the color range:

Each node $[l, r)$ stores:

- `top4`: the top4($\overrightarrow{v}$) for colors in $[l, r)$,
- `v_next`: the next direction where this node's `top4` or any child's `top4` changes.

The monoid operation:

- Merge children's `top4` sets (8 points) to produce the parent's 4-point `top4`.
- Compute `v_next` as the earliest breakpoint among child nodes or from changes in the merged hull.

During rotation, advance $\overrightarrow{v}$ to the root's `v_next`, then update all nodes whose `v_next` matches it.

Each node updates $O(n)$ times where n is its number of points. Thus the total is $O(N(\log N)^2)$.

Problem M. Many Approaches

We will refer to the squares and the park as *cells* and the *grid*, respectively. Also, we call a **march** a *scenario*.

Offline Solution

We first describe a solution that is allowed to read all queries in advance and answer them offline.

We extend both the people and the cells by M to the left and right. That is, assume the cells are indexed $-M, -M+1, \ldots, N+M-1$, and initially person i stands on cell i. When answering how many people are on a given cell, we count only people $0, 1, \ldots, N-1$.

First, we simulate the scenario over the entire array A. We then design a data structure that supports the following queries:

1. Return the cell index where person x currently stands.
2. Return the minimum index of any person standing on cell x.

3. Return the maximum index of any person standing on cell x.
4. Move every person not on cell x one step closer to x.

Because movements never change the relative order of people, the difference between the positions of person $i+1$ and person i is always 0 or 1. Moreover, once this difference becomes 0, it stays 0 forever. Thus, we maintain these differences d_i using a segment tree with range sums. Queries 1, 2, and 3 can be implemented via binary search on the segment tree, and Query 4 reduces to updating at most two points in the segment tree using Queries 1–3.

During the full simulation over A, we issue Query 4 for $x = A_1, A_2, \ldots, A_M$ in order.

We can answer interval-scenario queries by issuing appropriate data-structure queries at the correct times during this simulation. We denote the scenario for $(A_L, A_{L+1}, \ldots, A_R)$ by $A[L, R]$.

Determining the destination of person P under scenario $A[L, R]$

Immediately before processing $x = A_L$, we query (using Query 2) for some person standing on cell P; call this person y. Because we extended the number of people and cells, such a y always exists. We can then identify the movement of person y with that of person P under scenario $A[L, R]$. Thus, immediately after processing $x = A_R$, Query 1 gives the final cell of y, which is the answer.

Counting how many people end up on cell P after scenario $A[L, R]$

Immediately after processing $x = A_R$, Queries 2 and 3 give the minimum and maximum person indices on cell P, say l and r. Immediately before processing $x = A_L$, we use Query 1 to find the positions l' and r' of persons l and r, respectively. The set of people who end up on cell P under scenario $A[L, R]$ corresponds to the set of people who were initially within positions $[l', r']$ before A_L. In the actual scenario, only people 1 through N exist at the beginning, each occupying one position, so the answer is the size of the intersection of intervals $[l', r']$ and $[1, N]$.

Online Conversion

These queries can also be processed online. Interval-scenario queries do not modify the data structure, so it suffices to simulate the entire scenario for A in advance, and to access the data structure at specific times. Persistence is all that is required; the above method works with a persistent segment tree. This solution runs in $O((N + M + Q) \log(N + M))$ time and space.

Remark

The reason for requiring online queries is to distinguish this problem from a naive approach using a balanced binary tree, where one would insert "relevant" people into the structure per query without extending people or cells.