Problem Tutorial: "Young Salvador"

To begin with, it is not very difficult to calculate the height and width of the finished drawing: it will be a square with a side of $a + \lceil \frac{k}{2} \rceil \cdot 2b + \lfloor \frac{k}{2} \rfloor \cdot 2c$. Indeed, we start with a square of side a, and then with each move, we increase the side by $2b$ or $2c$.

Therefore, for some groups of test cases (1, 5, and 6), it was possible to write a solution of the type «create a board of known height and width, and color the cells on it according to the conditions, after which simply count the number of cells of each color». Of course, such a solution requires learning to transform the position of the cells relative to the center of the figure into a position relative to the origin of such a table, but this can be done with fairly simple transformations.

At the same time, in the first two groups of test cases, it was possible to score points with a simpler solution. Note that if the number of moves $k \leq 2$, then the length of the side of the figure will be equal to a, then $a + 2b$, then $a + 2b + 2c$. One could explicitly write down a formula that defines the number of cells added at each move: a^2, $2b \cdot (2a + 2b)$, $2c \cdot (2a + 4b + 2c)$. After that, it was only necessary to sum the number of cells of one color.

In groups 5 and 6, the solution described above worked, but in group 5, a slightly less implementation-complex solution also passed: instead of coloring the «frames» around the current figure, it was sufficient to make moves to the end and color the gradually decreasing squares entirely. In this case, the final drawing would not change, and to implement such a solution, it was enough to simply maintain the converging indices of the upper, lower, left, and right boundaries of the current square.

Test groups 3 and 4 were designed for a careful derivation of the formula: in the case of $a = 0$, the figure will consist of frames of width b or c of alternating colors. If we write the number of cells in each frame as the difference of the areas of the large and small squares, then if the side of the next square is d_i, the number of cells in the i-th layer will be equal to $d_i^2 - d_{i-1}^2$. By factoring out common factors,

- in group 3, we obtain a formula for the number of red cells

  $$(2b)^2 + (4b + 2c)^2 - (2b + 2c)^2 + (6b + 4c)^2 - (4b + 4c)^2 + \ldots = 2b \cdot (2b + (6b + 4c) + (10b + 8c) + \ldots),$$

  which can then be expanded using the formula for an arithmetic progression;
- in group 4, it is even simpler: black cells occupy only the initial square with side a, while the remaining $(a + 2bk)^2 - a^2$ cells are red.

Finally, the complete solution assumed the use of the same observation about the number of cells in each layer. We will simply iterate through all the moves, maintaining the current color, the length of the next frame, and the current length of the side of the figure. At each move, it is enough to add to the counter of cells of the current color the difference of the squares of the new side length and the old side length of the figure. Thus, the solution runs in time $\mathcal{O}(k)$ and scores full points.

It was also possible to solve the entire problem with a formula in $\mathcal{O}(1)$, but the formula reduces to summing several arithmetic progressions and looks quite complex, so you were not required to derive this formula completely in this problem.

Problem Tutorial: "Dutton's Patterns"

Solution Steps

1. Splitting the string into groups of identical characters:
   - We traverse the string and split it into sequences of identical characters, keeping the character and its length.

2. Iterating through all possible triples of groups:
    - We iterate through all possible triples of groups to find those that satisfy the condition $a_1 \neq a_2 \neq a_3$ and have the corresponding lengths k, i, j.
3. Checking length correspondence:
    - We check whether the lengths of the groups match the given k, i, j.
4. Counting the number of substrings:
    - If all conditions are met, then this triple of groups forms a valid substring.

Step-by-step Code Explanation in Python

1. Reading input data:

```
n = int(input())

s = input()

left, middle, right = map(int, input().split())
```

    - n — length of the string.
    - s — the string itself.
    - left, middle, right — counts of characters for substrings.
2. Splitting the string into groups of identical characters:

```
a = []
a.append((s[0], 1))
for i in range(1, n):
    if s[i] == a[-1][0]:
        a[-1] = (a[-1][0], a[-1][1] + 1)
    else:
        a.append((s[i], 1))
```

    - We start with the first group consisting of the first character of the string.
    - We traverse the string and combine consecutive identical characters into groups.
3. Iterating through all possible triples of groups and checking conditions:

```
cnt = 0
for i in range(1, len(a) - 1):
    if a[i - 1][1] >= left and a[i][1] == middle and a[i + 1][1] >= right:
        cnt += 1
```

    - We iterate through all groups, starting from the second and ending at the second-to-last.
    - We check that the number of characters in the neighboring groups meets the requirements left, middle, right.
4. Outputting the result:

```
print(cnt)
```

- We output the number of found substrings that meet the problem's conditions.

Example of Code Execution

For the string `llvvyywwww` and parameters left = 2, middle = 2, right = 2:

1. Splitting the string into groups:
   - `ll` $\rightarrow (\text{'l'}, 2)$
   - `vv` $\rightarrow (\text{'v'}, 2)$
   - `yy` $\rightarrow (\text{'y'}, 2)$
   - `wwww` $\rightarrow (\text{'w'}, 4)$
2. Iterating through triples of groups and checking conditions:
   - For $((\text{'l'}, 2), (\text{'v'}, 2), (\text{'y'}, 2))$, the conditions are satisfied.
   - For $((\text{'v'}, 2), (\text{'y'}, 2), (\text{'w'}, 4))$, the conditions are satisfied.

Result: 2 valid substrings.

Problem Tutorial: "Cows Check Dutton"

Solution to the Problem

1. Reading Input Data:

```
import sys
from collections import defaultdict

input = sys.stdin.read
data = input().splitlines()

n = int(data[0])
s = data[1]
```

2. Initializing Commands and Variables:

```
comand = {
    'L': (-1, 0),
    'R': (1, 0),
    'U': (0, 1),
    'D': (0, -1),
}

X = defaultdict(int)
Y = defaultdict(int)
x, y = 0, 0
X[0] = 1
Y[0] = 1
```

```
cell = {(0, 0): 1}
```

3. Processing Commands and Recording Values in Cells:

```
for i in range(n):
    c = s[i]
    x += comand[c][0]
    y += comand[c][1]
    tmp = cell.get((x, y), 0)
    cell[(x, y)] = i + 2
    X[x] -= tmp
    Y[y] -= tmp

    X[x] += i + 2
    Y[y] += i + 2
```

4. Processing Queries and Outputting Results:

```
q = int(data[2])
results = []
for it in range(q):
    tmp = list(map(int, data[3 + it].strip().split()))
    tp = tmp[0]
    if tp == 1:
        x = tmp[1]
        results.append(str(X[x]))
    else:
        y = tmp[1]
        results.append(str(Y[y]))

print("\n".join(results))
```

Detailed Explanation of the Solution

1. Reading Input Data:
   - We use the *sys* module to read input data, which allows for fast processing of large volumes of data.
   - We read the number of commands n and the command string s.
2. Initializing Commands and Variables:
   - We create a dictionary ‘comand‘, where each command (L, R, U, D) is associated with changes in coordinates on the field.
   - We create two dictionaries X and Y to store the sums of values in rows and columns, respectively. We use ‘defaultdict‘ for convenience.
   - We initialize the starting position $(0, 0)$ and record the value 1 in it.
3. Processing Commands and Recording Values in Cells:
   - We iterate through the command string and compute the new position after each command.
   - We get the current value in the cell and update it to the new value.

- We update the sum of values in the current row and column, taking into account the new value and subtracting the old value.

4. Processing Queries and Outputting Results:
   - We read the number of queries q.
   - We iterate through each query and, depending on the type of query (1 or 2), output the sum of values in the specified row or column.
   - We store the results and output them at the end.

This algorithm efficiently solves the problem, ensuring that commands and queries are processed in linear time, which is suitable for the problem's constraints.

Problem Tutorial: "Segments"

An important fact is that if a pair of integers (l, r) satisfies the conditions, then the pair $(l+1, r)$ also satisfies the conditions. Thus, there exists a d_r such that:

For a fixed r, the pair (l, r) satisfies the conditions if and only if l is an integer between d_r and r inclusive.

If we can find this d_r for each r, the desired answer can be found as:

$$\sum_{r=1}^{M}(r - d_r + 1).$$

Now let's consider how to find this d_r.

Suppose we already know d_{r-1} and now want to find d_r. If there is no i such that $R_i = r$, then there are no new constraints, so $d_r = d_{r-1}$. If such an i exists, then:

$$d_r = \max(d_{r-1}, L_{\max} + 1),$$

where $L_{\max}$ is the maximum value of L_i for i such that $R_i = r$.

By stepwise evaluation of these recurrence relations, we can find all d_r, starting from d_1 in order.

The problem can be solved by correctly implementing this procedure. The complexity is $O(N + M)$.

Problem Tutorial: "Basements and Mathematicians"

Solution Idea

The main goal is to find the maximum possible area of a regular polygon with k vertices that can be placed on a table of size $n \times m$.

Solution Steps

1. Input Processing

We read the dimensions of the table and the number of sides of the polygon:

$$n, m, k$$

where n and m are the dimensions of the table, k is the number of sides of the polygon.

For simplicity, let $n \leq m$.

2. Conditions for $k > 4$

If the number of sides of the polygon $k > 4$, then it is impossible to place such a polygon on the table, so the problem only considers regular polygons with 3 and 4 vertices:

$$\text{If } k > 4, \text{ output } -1.$$

3. Processing for a Regular Triangle ($k = 3$)

Lemma

In the optimal placement, we can assume that the regular triangle and the table share at least one common vertex.

Proof

Consider the topmost, bottommost, rightmost, and leftmost points on the regular triangle. In fact, there exists at least one such vertex for each of these points. Also, a vertex cannot simultaneously be the topmost and bottommost, or the rightmost and leftmost, so the triangle always satisfies at least one of the following statements:

1. There exists a vertex that is the topmost and rightmost point on the triangle.
2. There exists a vertex that is the topmost and leftmost point on the triangle.
3. There exists a vertex that is the bottommost and rightmost point on the triangle.
4. There exists a vertex that is the bottommost and leftmost point on the triangle.

For example, if the first statement holds, we can shift the triangle so that this vertex coincides with at least one of the vertices of the rectangle, and therefore we can assume that the optimal placement satisfies this assumption.

The area of a regular triangle depends on the length of its side, so we need to maximize the side length.

For the regular triangle, we fix the first vertex at the point $(0, 0)$, and the second vertex will be at the point (n, mid), where mid will be found using binary search.

As mid increases, the side length will increase, since the side length is $\sqrt{n^2 + \text{mid}^2}$.

The area of a regular triangle with a given side can be calculated using the formula:

$$S = \frac{\sqrt{3}}{4} \times a^2$$

where a is the length of the triangle's side.

Binary Search for the Triangle

During the binary search, we fix the second vertex at the point (n, mid). Since we know the coordinates of two points of the triangle, we can find the third.

The vector $v(n, \text{mid})$ represents the coordinates of the second vertex of the regular triangle. After that, we rotate vector v by an angle of $\frac{\pi}{3}$ (counterclockwise) to obtain the coordinates of the triangle's third vertex. We check whether this vertex fits on the table. Thus, we find the maximum value of mid.

After that, we calculate the area of the triangle with the found side length.

4. Processing for a Square ($k = 4$)

If $k = 4$, we can simply calculate the area of the square as the square of the minimum of the table's sides. The area of the square will be:

$$S = n^2 \quad \text{(where } n \text{ is the minimum side of the table).}$$

Brief Algorithm

1. Read the input data n, m, k.
2. If $k > 4$, output -1.
3. If $k = 3$, perform binary search to find the maximum possible side of the regular triangle.
4. If $k = 4$, calculate the area of the square that fits on the table.

Implementation

```
typedef long double ld;
const ld eps = 0.000'000'001;

ld pow2(ld x){
    return x * x;
}

struct pnt{
    ld x, y;

    pnt(){}

    pnt(ld _x, ld _y){
        this->x = _x;
        this->y = _y;
    }
};

pnt operator-(pnt a, pnt b){
    return pnt(a.x - b.x, a.y - b.y);
}

pnt operator+(pnt a, pnt b){
```

```
    return pnt(a.x + b.x, a.y + b.y);
}

pnt rot(pnt v, ld a){
    return pnt(
        cos(a)*v.x - sin(a)*v.y,
        sin(a)*v.x + cos(a)*v.y
    );
}

const ld pi = acos(-1.0);

int main() {
    ld n, m;
    int k;
    cin >> n >> m >> k;

    if(k > 4){
        cout << -1;
        return;
    }

    if(n > m)
        swap(n, m);

    if(k == 3){
        ld l = 0, r = m;
        for(int i = 0; i < 100; ++i){
            ld mid = (l + r) / 2.0;

            pnt v(n, mid);
            pnt u = rot(v, pi / 3.0);

            if(u.x >= 0 && u.y <= m)
                l = mid;
            else
                r = mid;
        }
        cout << fixed << setprecision(20) << (sqrt(3) * (r*r + n*n)) / 4.0;
    } else {
        cout << fixed << setprecision(20) << n*n;
    }
}
```

Problem Tutorial: "Ilnur and the Array"

Algorithm for Solution:

1. **Dirichlet's Principle**:
    - If the number of pairs of elements in the array $\frac{n(n-1)}{2}$ exceeds 1,000,000, then the array cannot be good, because there will be too many pairs for all of them to be unique. In this case, we

immediately output "NO".

- Since the value $P(i,j) = \lceil \sqrt{a_i \cdot a_j} \rceil$ is rounded up, there cannot be more than 1,000,000 distinct values of P.

2. **Calculating the value of $P(i,j)$:**
   - For each pair of indices i and j, calculate $P(i,j)$ using the formula $P(i,j) = \lceil \sqrt{a_i \cdot a_j} \rceil$.
   - Use rounding up to obtain an integer.
3. **Checking for uniqueness of values**:
   - Store the computed values of $P(i,j)$ in a flag array b.
   - If $P(i,j)$ is already present in the flag array, output «NO» and terminate the program.
   - If all values are unique, output «YES».

Implementation in Python:

```
import math

N = 1000000
n = int(input())

# If the number of pairs exceeds 1,000,000, then the array cannot be good
if n * (n - 1) // 2 > N:
    print("NO")
    exit(0)

# Reading the array
s = input().split()
a = [int(s[i]) for i in range(n)]

# Array to track the uniqueness of values P(i, j)
b = [0] * (N + 1)

for i in range(n):
    for j in range(i + 1, n):
        v = a[i] * a[j]
        u = int(math.sqrt(v))
        # Adjust u for an exact integer value
        while u * u > v:
            u -= 1
        while u * u < v:
            u += 1

        # Check for uniqueness
        if b[u] == 1:
            print("NO")
            exit(0)
        b[u] = 1

print("YES")
```

Explanation of Dirichlet's Principle:

Dirichlet's Principle states that if m objects are distributed among n boxes, and $m > n$, then at least one box will contain more than one object. In the context of this problem:

- The value $P(i, j) = \lceil \sqrt{a_i \cdot a_j} \rceil$ is rounded up, so there cannot be more than 1,000,000 distinct values of P.
- If the number of pairs exceeds 1,000,000, then by Dirichlet's Principle, there must be at least two pairs of numbers for which the value $P(i, j)$ coincides. Therefore, if $\frac{n(n-1)}{2} > 1,000,000$, the array cannot be good, as among these pairs there will be identical values of P.
- Thus, if the number of pairs exceeds 1,000,000, we immediately output «NO».

Problem Tutorial: "Dutton's Night Snack 7-8"

1. **Position Iteration**: For each position i from 0 to $n - k$:
   - If the bulb at position i is off (0), toggle k consecutive bulbs starting from position i.
   - Increase the action counter.
2. **Result Check**: After processing all positions, check if all bulbs are on. If yes, output the number of actions. If not, output -1.

Problem Tutorial: "Primitive Tetris"

In the first group of test cases, $n = 1$ and $m = 4$. Notice that on a field of height 1, the only figure that will not go beyond the boundaries of the field immediately is a horizontal strip of width 4. It is also worth noting that this figure can be easily distinguished from the others, even without considering the points and grids in the input data— for it, $r_i = 1$ and $c_i = 4$. Therefore, in this group of test cases, it was sufficient to check that the first figure is a strip 1×4, and in that case, output 2, otherwise output 1.

The second group of test cases was solved similarly, but it required significant case enumeration. If you were unable to submit a complete solution, you could try to pass at least this group of tests with an incomplete solution that had some errors. Alternatively, the case enumeration for this group could be reduced by noting that very few figures can overlap on such a low field without going beyond the boundary, and that two figures that do not go beyond the boundary can only be placed next to each other horizontally if they have heights $r_i = 2$ or $r_i = 3$, and there are not many such figures.

The third group guaranteed that all figures are «columns» 4×1. For such figures, the game can be processed quite simply: each column of the field will have a height of $4k$, where k is the number of figures that have fallen into that column. Therefore, it is sufficient to store the current height for each of the m columns of the field and monitor when the height of any of them exceeds n.

The fourth group is absolutely analogous to the third, with the only difference being that now the field is filled with squares 2×2. Since all squares fall on even columns, the filling of the field will be «dense», and it is also sufficient to maintain the height of each column of the field.

The remaining groups required the implementation of a more or less complete solution. The idea remains the same: for each column, we store its «height». An important distinction is that there can be "gaps"in the field, so we will store for each column h_j— the highest cell in it. Now let's learn how to determine at what height a figure will stop.

- In the fifth group of test cases, all figures are strips 1×4, so they will fall to a height of $\max(h_j, \ldots, h_{j+3}) + 1$;

- In the sixth and seventh groups, it was sufficient to move the figure explicitly one cell down and check for intersection with the current tiling of the field; in the sixth group, small constraints allowed performing all these actions arbitrarily with almost any running time;
- For a complete solution, it was necessary to find mn_j and mx_j for each figure and each of its columns— the lowest and highest cells in it; after which the row at which the figure will stop is determined as $R = \max(h_j - mn_0, \ldots, h_{j+3} - mn_3) + 1$, and recalculate the heights of the columns as $h_{j+0\ldots3} \leftarrow R + mx_{0\ldots3}$.

Accordingly, the complete solution processes each figure in $\mathcal{O}(1)$ and works in $\mathcal{O}(q)$ in total.

Problem Tutorial: "Primitive Tetris 7-8"

This problem is about a simple simulation of the process described in the statement. The easiest solution is to either draw a field of the required size on paper or open the test in a text editor, and then sequentially place the given shapes on the field according to the process described in the statement.

Of course, when solving the problem "manually", one could accidentally make a mistake, so alternatively, one could write code that solves this problem. The simplest solution stores the field as a two-dimensional array (matrix) and, when processing each shape, moves it down one row at a time until the next downward movement of the shape would cause it to intersect with the cells already placed on the field.