Problem Statement 1: "Sparse Matrix Compression Check"

You're given a square matrix of integers. You need to determine if this matrix can be **"sparsely compressed"** based on a specific rule. A matrix can be sparsely compressed if, for every row, the number of **non-zero elements** is less than or equal to a given **threshold** `K`. If it can be compressed, you also need to find the row with the maximum number of non-zero elements.

Constraints:

- Matrix dimensions `N x N`, where `1 <= N <= 10`.
- Elements of the matrix are integers between 0 and 100.
- Threshold `K` is an integer between 1 and `N`.

Input Format:

- First line: Integer `N` (dimension of the matrix).
- Second line: Integer `K` (threshold for non-zero elements).
- Next `N` lines: `N` integers representing each row of the matrix.

Output Format:

- If the matrix can be sparsely compressed, print "YES" followed by a newline, and then print the maximum number of non-zero elements found in any row.
- If the matrix cannot be sparsely compressed, print "NO" followed by a newline, and then print the first row index (0-indexed) that violates the compression rule.

Sample Input 1:

```
3
2
1 0 5
0 0 2
3 1 0
```

Sample Output 1:

```
YES
2
```

Explanation 1:

- Row 0: 2 non-zero elements (1, 5). `2 <= 2`.
- Row 1: 1 non-zero element (2). `1 <= 2`.
- Row 2: 2 non-zero elements (3, 1). `2 <= 2`. All rows satisfy the condition. The maximum number of non-zero elements in any row is 2.

Sample Input 2:

```
4
1
10 0 0 5
0 0 2 0
3 1 0 0
0 0 0 0
```

Sample Output 2:

```
NO
0
```

Explanation 2:

- Row 0: 2 non-zero elements (10, 5). `2 > 1`. This row violates the condition. The matrix cannot be compressed.

Problem Statement 2: "Cipher Sequence Validator"

You're given a sequence of characters (a string, represented as a character array) and a **key value**. You need to check if the sequence is a **valid "cipher sequence"** based on these rules:

1. Each character in the sequence, when its ASCII value is shifted by the `key` value, must result in an **uppercase English alphabet character** ('A' to 'Z').
2. The shifted character must be **unique** within the entire sequence of shifted characters.

If the sequence is a valid cipher sequence, print "VALID". Otherwise, print "INVALID".

Constraints:

- The length of the character sequence is between 1 and 20.
- The sequence contains only lowercase English alphabet characters ('a' to 'z').
- The key value is an integer between 1 and 25.

Input Format:

- First line: A character array representing the sequence (e.g., `char sequence[21];`).
- Second line: An integer `key`.

Output Format:

- "VALID" or "INVALID"

Sample Input 1:

```
abc
1
```

Sample Output 1:

```
VALID
```

Explanation 1:

- 'a' + 1 = 'b' (ASCII 97 + 1 = 98) -> 'B' (ASCII 66)
- 'b' + 1 = 'c' (ASCII 98 + 1 = 99) -> 'C' (ASCII 67)
- 'c' + 1 = 'd' (ASCII 99 + 1 = 100) -> 'D' (ASCII 68) All shifted characters 'B', 'C', 'D' are uppercase and unique.

Sample Input 2:

```
abz
1
```

Sample Output 2:

```
INVALID
```

Explanation 2:

- 'a' + 1 = 'B'
- 'b' + 1 = 'C'
- 'z' + 1 = '{' (ASCII 122 + 1 = 123). This isn't an uppercase English alphabet character. So, it's invalid.

Sample Input 3:

```
aba
1
```

Sample Output 3:

```
INVALID
```

Explanation 3:

- 'a' + 1 = 'B'
- 'b' + 1 = 'C'
- 'a' + 1 = 'B' The shifted character 'B' isn't unique (it appears twice). So, it's invalid.

Problem Statement 3: "Alternating Sign Array Sum"

You're given an array of integers. You need to calculate a **special sum** based on an "alternating sign" rule. For each element in the array:

- If its index is **even** (0, 2, 4...), it contributes positively to the sum.
- If its index is **odd** (1, 3, 5...), it contributes negatively to the sum.
- **However, there's a twist**: if an element is a **prime number**, its contribution is **always positive**, regardless of its index.

Constraints:

- Array size `N`, where `1 <= N <= 15`.
- Elements of the array are integers between -100 and 100.

Input Format:

- First line: Integer `N` (size of the array).
- Second line: `N` integers representing the elements of the array.

Output Format:

- The final calculated special sum.

Sample Input 1:

```
5
10 3 7 12 5
```

Sample Output 1:

```
13
```

Explanation 1:

- Index 0 (even): 10 (not prime). Contribution: +10
- Index 1 (odd): 3 (prime). Contribution: +3 (due to prime rule)
- Index 2 (even): 7 (prime). Contribution: +7 (due to prime rule)
- Index 3 (odd): 12 (not prime). Contribution: -12
- Index 4 (even): 5 (prime). Contribution: +5 (due to prime rule) Total Sum: `10 + 3 + 7 - 12 + 5 = 13`.

Problem Statement 4: "Balanced Bracket Sequence Search"

You're given an array of characters, which represents a sequence. You need to find the **longest *contiguous* sub-sequence** within this array that is a **"balanced bracket sequence"**. A balanced bracket sequence is defined as follows:

1. It contains only '(', ')', '[', ']', '{', '}'.
2. Every opening bracket has a corresponding closing bracket of the same type.
3. The brackets are properly nested.

You only need to output the **length** of the longest such sub-sequence. If no such sub-sequence exists, output 0.

Constraints:

- The length of the character array is between 1 and 30.
- The array can contain any ASCII characters (not just brackets).

Input Format:

- First line: An integer `N` (length of the character array).
- Second line: `N` characters (e.g., `char sequence[31];`).

Output Format:

- The length of the longest balanced bracket sub-sequence.

Sample Input 1:

```
10
( ( ) [ ] { } ) )
```

Sample Output 1:

```
6
```

Explanation 1: The sub-sequence { }) starting from index 2 to 7 has a length of 6 and is balanced.

Sample Input 2:

```
7
[ ( ] ) { } [
```

Sample Output 2:

```
2
```

Explanation 2:

- The sub-sequence `()` from index 1 to 2 has length 2.

- The sub-sequence `{}` from index 4 to 5 has length 2. The longest balanced sub-sequences have length 2.

Sample Input 3:

```
5
a b c d e
```

Sample Output 3:

```
0
```

Explanation 3: No brackets are present, so no balanced bracket sequence.

Problem Statement 5: "Matrix Element Constraint Check"

You're given a `3 x 3` integer matrix and a **target value** `T`. You need to determine if the element at a specific position `(R, C)` within the matrix satisfies a given condition based on its row and column parities. The condition is:

- If `R` is even and `C` is even, the element at `(R,C)` must be **greater than** `T`.
- If `R` is even and `C` is odd, the element at `(R,C)` must be **less than** `T`.
- If `R` is odd and `C` is even, the element at `(R,C)` must be **equal to** `T`.
- If `R` is odd and `C` is odd, the element at `(R,C)` must be a **positive multiple of** `T` (assuming `T` is positive; if `T` is negative or zero, this condition is false).

Constraints:

- Matrix elements are integers between -100 and 100.
- Target value `T` is an integer between -100 and 100.
- Row `R` and Column `C` are integers between 0 and 2.

Input Format:

- First line: Integer `R` (target row).
- Second line: Integer `C` (target column).
- Third line: Integer `T` (target value).
- Next 3 lines: 3 integers representing each row of the matrix.

Output Format:

- "YES" if the element at `(R,C)` satisfies the condition, "NO" otherwise.

Sample Input 1:

```
0
0
5
10 2 3
4 5 6
7 8 9
```

Sample Output 1:

```
YES
```

Explanation 1:

- `R=0` (even), `C=0` (even).
- Condition: Element at `(0,0)` must be greater than `T`.
- Element at `(0,0)` is 10. `10 > 5`. Condition satisfied.

Sample Input 2:

```
0
1
5
10 2 3
4 5 6
7 8 9
```

Sample Output 2:

```
YES
```

Explanation 2:

- `R=0` (even), `C=1` (odd).
- Condition: Element at `(0,1)` must be less than `T`.
- Element at `(0,1)` is 2. `2 < 5`. Condition satisfied.

Sample Input 3:

```
1
0
5
10 2 3
4 5 6
7 8 9
```

Sample Output 3:

```
NO
```

Explanation 3:

- `R=1` (odd), `C=0` (even).
- Condition: Element at `(1,0)` must be equal to `T`.
- Element at `(1,0)` is 4. `4 != 5`. Condition not satisfied.

Sample Input 4:

```
1
1
2
10 2 3
4 6 6
7 8 9
```

Sample Output 4:

```
YES
```

Explanation 4:

- `R=1` (odd), `C=1` (odd).
- Condition: Element at `(1,1)` must be a positive multiple of `T`.
- Element at `(1,1)` is 6. `T=2`. 6 is a positive multiple of 2 (`6 = 3 * 2`). Condition satisfied.