Problem 1:

Problem Statement: You are given an array of `n` integers. In one operation, you can remove **any** element from the array. Your task is to determine whether it's possible to perform some number of such operations so that **all the elements left in the array are equal**.

Solution: First, sort the array. Then, check each pair of consecutive elements. If every pair differs by at most 1, it's possible to remove elements in a way that makes all remaining numbers equal. However, if any pair differs by more than 1, it's **impossible** to achieve that goal.

Source Code :

```
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];

    sort(a.begin(), a.end());
    bool isOk = true;
    for (int i = 1; i < n; i++){
        if (a[i] - a[i - 1] > 1) {
            isOk = false;
            break;
        }
    }
    if (isOk) cout << "YES\n";
    else cout << "NO\n";
}

int main(void) {
    int t;
    cin >> t;
    while (t--) solve(t);
}
```

Input :

```
3
4
1 2 2 3
3
1 5 9
1
42
```

Output :

```
YES
NO
YES
```

Discussion:
Initially considered complex removal strategies, but realized that sorting simplifies the problem. The key insight is: if all consecutive elements differ by at most 1, we can remove outliers to make the rest equal. A gap greater than 1 makes it impossible.

Problem - 02 :

Problem Statement: Given two integers `a` and `b`, determine the minimum number of operations needed to make `a` equal to `b`. In each operation, you can add or subtract any number between 1 and 10 (inclusive) to/from `a`.

Solution: Calculate the absolute difference: `diff = |a - b|`. Each operation can change `a` by up to 10. So, the minimum number of operations is `ceil(diff / 10)`.

Source Code :

```cpp
#include<bits/stdc++.h>
using namespace std;

void solve(int tc){
    int a, b;
    cin >> a >> b;

    if (a == b){
        cout << 0 << '\n';
        return;
    }

    int cnt = 0;
    if (a > b){
        long long diff = a - b;
        cnt += diff / 10;
        a -= (cnt * 10);
        if (a != b) cnt++;

        cout << cnt << '\n';
        return;
    }

    cnt = 0;
    long long diff = b - a;
    cnt += diff / 10;
    a += (cnt * 10);
    if (a != b) cnt++;
    cout << cnt << '\n';
}

int main(void){
    int t;
```

```
    cin >> t;
    while (t--) solve(t);
}
```

Input :

```
4
5 5
13 42
18 4
1337 420
```

Output :

```
0
3
2
92
```

Discussion: The problem becomes simple once you notice each operation can change `a` by up to 10. To minimize operations, we just need the ceiling of the difference divided by 10. Used the trick `(d + 9) / 10` to compute ceiling without floating point. It's efficient and avoids precision issues.

Problem - 03 :

Problem Statement: Given three integers `a`, `b`, and `c`, check if either `a + b` equals `c` or `a - b` equals `c`. Determine if there exists such an operation (`+` or `-`) making the equation true.

Solution: Simply verify if `a + b == c` or `a - b == c`. If either is true, the answer is yes; otherwise, no.

Source Code :

```
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    int a, b, c;
    cin >> a >> b >> c;

    if (a + b == c) cout << "+\n";
    else if (a - b == c) cout << "-\n";
}

int main(void){
    int t;
    cin >> t;
    while (t--) solve(t);
}
```

Input :

```
3
3 4 7
5 8 3
1 1 2
```

Output :

```
+
-
+
```

Discussion : That only requires checking which operation (`+` or `-`) yields the result. The problem guarantees exactly one valid operation, so no additional error handling is needed.

Problem - 04 :

Problem Statement : Given a string `s` and integer `k`, check if removing exactly `k` characters allows the remaining letters to be rearranged into a palindrome. A palindrome reads the same forwards and backwards. Your task is to determine whether such a rearrangement is possible after the removals.

Solution : Count the frequency of each character in the string. For a string to be rearranged into a palindrome, at most one character can have an odd frequency count. Calculate how many characters have odd frequencies. Removing characters can help reduce these odd counts. Check if after removing exactly `k` characters, the odd frequency condition is satisfied.

Source Code :

```cpp
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    int n, k;
    cin >> n >> k;
    string s;
    cin >> s;

    vector<int> freq(26, 0);
    for (char c : s) freq[c - 'a']++;

    int oddCnt = 0;
    for (int i = 0; i < 26; i++) {
        if (freq[i] % 2 == 1) oddCount++;
    }

    if (oddCnt == 0 && k >= 0) cout << "YES\n";
    else if (oddCnt - 1 <= k && k <= n) cout << "YES\n";
    else cout << "NO\n";
}

int main(void){
    int t;
    cin >> t;
```

```
    while (t--) solve(t);
}
```

Input :

```
3
1 0
a
2 0
ab
2 1
```

Output :

```
YES
NO
YES
```

Discussion : The key insight is that a palindrome can have at most one character with an odd frequency. After removing `k` characters, the number of odd-frequency characters must be at most `k + 1`. The `+1` accounts for the possible middle character in palindromes of odd length. Each removal can help reduce the count of odd-frequency characters by fixing pairs. If this condition holds, rearranging the remaining characters into a palindrome is possible.

Problem - 05 :

Problem Statement : A grasshopper starts at position `x` and makes jumps for `n` days. On each day `i`, it jumps `i` steps to the right if its current position is even, or `i` steps to the left if its current position is odd. Determine its final position after completing all `n` jumps.

Solution: Analyze the jumps using the pattern of `n % 4`. The final position depends on whether `x` is even or odd. Use these patterns to directly compute the result without simulating every jump.

Source Code :

```cpp
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    long long x, n;
    cin >> x >> n;
    long long r = n % 4;

    if (x % 2 == 0){
        if (r == 1) x -= n;
        else if (r == 2) x += 1;
        else if (r == 3) x += n + 1;
    }
    else{
        if (r == 1) x += n;
        else if (r == 2) x -= 1;
```

```
        else if (r == 3) x -= n + 1;
    }

    cout << x << '\n';
}

int main(void){
    int t;
    cin >> t;
    while (t--) solve(t);
}
```

Input :

```
4
0 1
0 2
10 10
10 99
```

Output :

```
-1
1
11
110
```

Discussion: Initially, the problem seemed to require simulating every jump, but a clear pattern emerges every 4 steps. The grasshopper's final position depends on the parity of the starting position and the value of `n % 4`. By analyzing small cases by hand, the pattern became evident, allowing for a direct calculation without simulation.

Problem 06:

Problem Statement: You have `n` integers and need to distribute them into three groups. Each group should contain exactly `n/3` elements based on their values modulo 3. Determine the minimum number of operations needed to achieve this, where each operation moves one integer from one group to another.

Solution: Count how many numbers have remainders 0, 1, and 2 when divided by 3.
Each group must contain exactly `n/3` elements. If a group has more than `n/3` elements, move the excess to the next group in a cycle `(0 → 1 → 2 → 0)`. Repeat this redistribution until all groups are balanced. The total moves made during this process is the minimum number of operations needed.

Source Code :

```
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    int n;
```

```
    cin >> n;
    vector<int> c(3, 0);
    for (int i = 0; i < n; i++){
        int x;
        cin >> x;
        c[x % 3]++;
    }
    int tar = n / 3;
    int ans = 0;
    for (int i = 0; i < 3; i++){
        if (c[i] > tar) {
            int x = c[i] - tar;
            c[(i + 1) % 3] += x;
            ans += x;
        }
    }
    cout << ans << "\n";
}

int main(void){
    int t;
    cin >> t;
    while (t--) solve(t);
}
```

Input:

```
2
6
0 2 5 5 4 8
6
2 0 2 1 0 0
```

Output :

```
3
1
```

Discussion: The crucial insight is that excess elements circulate in a cycle: from remainder 0 to 1, 1 to 2, and 2 back to 0. Each move costs one operation, and following this cyclic order minimizes the total number of operations needed. By adjusting the groups in sequence, we efficiently balance all counts with the fewest transfers.

Problem - 07 :

Problem Statement: Given integer x, determine if it can be represented as $a^3 + b^3$ where a and b are positive integers.

Solution: For each possible value of a from 1 to $\sqrt[3]{x}$, check if $x - a^3$ is a perfect cube.

Source Code :

```cpp
#include <bits/stdc++.h>
using namespace std;

void solve(int tc){
    long long x;
    cin >> x;
    bool found = false;

    for (long long a = 1; a * a * a <= x; a++) {
        long long b3 = x - a * a * a;
        if (b3 < 0) break;
        long long b = round(pow(b3, 1.0 / 3.0));
        if (b > 0 && b * b * b == b3) {
            found = true;
            break;
        }
    }

    if (found) cout << "YES\n";
    else cout << "NO\n";
}

int main(){
    int t;
    cin >> t;
    while (t--) solve(t);
}
```

Input :

```
3
1
2
4
```

Output :

```
NO
YES
NO
```

Discussion : The main challenge was checking if a number is a perfect cube efficiently. Used cube root approximation with rounding, then verified by cubing the result. Had to be careful with floating point precision for large numbers.