**B. AGAGA XOOORRR -** [**https://codeforces.com/contest/1516/problem/B**](https://codeforces.com/contest/1516/problem/B)

**1. Problem Statement (in simple words)**

You have:

* An array a of length n.
* You can **pick two adjacent elements**, replace them with their **bitwise XOR**, which reduces the array size by 1.
* You must **stop** when at least **2 elements remain**.
* Goal: Check if you can make **all remaining elements equal**.

**Output:** "YES" if possible, otherwise "NO".

**Example**

**Example 1:**

n = 3

a = [0, 2, 2]

Step:

Pick (0, 2) → XOR = 2 → array becomes [2, 2]

All equal → YES

**Example 2:**

n = 4

a = [2, 3, 1, 10]

No matter how you XOR adjacent pairs, you can’t end with ≥2 elements all equal → NO

**2. How to Think About It**

**Key observation about XOR**

* XOR is **associative** and **commutative**:
* a ^ b ^ c = c ^ a ^ b
* If all remaining elements must be equal to some value X, then:
  + Every segment we create must XOR to the same value.

**Case 1:**

If totalXor = a[0] ^ a[1] ^ ... ^ a[n-1] is **0** →  
We can always split the array into at least 2 equal parts (since 0 means we can balance segments easily). → Answer is **YES**.

**Case 2:**

If totalXor ≠ 0 →  
We can still succeed if we can split the array into **at least 3 parts** such that:

XOR of each part = totalXor

Why 3 parts?

* If you split into 3 parts, you can merge first part to 1 element, merge second part to 1 element, and the third will also be same because XOR matches.

**3. Solution Approach**

1. Compute totalXor of the array.
2. If totalXor == 0 → print "YES".
3. Else:
   * Traverse the array, keep cumulative XOR.
   * Whenever cumulative XOR equals totalXor, reset cumulative XOR and increase segment count.
   * If count >= 2 by the end, print "YES".
   * Else, "NO".

**4. C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

bool canMakeEqual(vector<int>& arr) {

int totalXor = 0;

for (int num : arr) {

totalXor ^= num;

}

if (totalXor == 0) return true;

int currentXor = 0, count = 0;

for (int num : arr) {

currentXor ^= num;

if (currentXor == totalXor) {

count++;

currentXor = 0;

}

}

return count >= 2; // need at least 3 segments → 2 cuts

}

int main() {

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<int> arr(n);

for (int i = 0; i < n; i++) {

cin >> arr[i];

}

cout << (canMakeEqual(arr) ? "YES" : "NO") << "\n";

}

return 0;

}

**5. Complexity Analysis**

* **Time Complexity:**
  + Computing totalXor: **O(n)**
  + Traversal to find segments: **O(n)**
  + Overall per test case: **O(n)**  
    With n ≤ 2000 and t ≤ 15 → safe.
* **Space Complexity:**
  + Only storing the array: **O(n)**

**Another solution :**

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int t;

    cin >> t;

    while (t--)

    {

        int n;

        cin >> n;

        int ar[n + 3];

        for (int i = 1; i <= n; i++)

        {

            cin >> ar[i];

        }

        int pre[n + 3];

        pre[0] = 0;

        for (int i = 1; i <= n; i++)

        {

            pre[i] = pre[i - 1] ^ ar[i];

        }

        int ans = 0;

        // p 2

        for (int i = 1; i < n; i++)

        {

            int a = pre[i];

            int b = pre[n] ^ pre[i];

            if (a == b)

            {

                ans = 1;

                break;

            }

        }

        // p - 3

        for (int i = 1; i < n - 1; i++)

        {

            int a = pre[i];

            for (int j = i + 1; j < n; j++)

            {

                int b = pre[j] ^ pre[i];

                int c = pre[n] ^ pre[j];

                if (a == b && b == c)

                {

                    ans = 1;

                    break;

                }

            }

        }

        if (ans == 1)

            cout << "YES" << endl;

        else

            cout << "NO" << endl;

    }

    return 0;

}

**B. Morning Jogging -** [**https://codeforces.com/contest/1517/problem/B**](https://codeforces.com/contest/1517/problem/B)

#include<bits/stdc++.h>

using  namespace  std;

int main()

{

    int t;

    cin>>t;

    while(t--)

    {

        int n,m,x;cin>>n>>m;

        vector<int>v[n+4];

        vector<int>ans[n+3];

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<m;j++)

            {

                cin>>x;

                v[i].push\_back(x);

            }

        }

        for(int i=0;i<n;i++)

        {

            sort(v[i].begin(),v[i].end());

        }

        for(int i=0;i<m;i++)

        {

            int mn=INT\_MAX;

            int ind=-1;

            for(int j=0;j<n;j++)

            {

                if(v[j][0]<mn)

                {

                    mn=v[j][0];

                    ind=j;

                }

            }

            for(int j=0;j<n;j++)

            {

                if(j==ind)

                {

                    ans[j].push\_back(v[j][0]);

                    v[j].erase(v[j].begin());

                }

                else

                {

                    ans[j].push\_back(v[j][v[j].size()-1]);

                    v[j].pop\_back();

                }

            }

        }

        for(int i=0;i<n;i++)

        {

           for(int j=0;j<ans[i].size();j++)cout<<ans[i][j]<<" ";

           cout<<endl;

        }

    }

    return 0;

}

**B. TMT Document -** [**https://codeforces.com/contest/1509/problem/B**](https://codeforces.com/contest/1509/problem/B)

**🔹 Problem Restatement**

We are given a string consisting of only **T** and **M** characters. We want to check if we can partition the string into subsequences (not substrings) such that **each subsequence is exactly "TMT"**.

**Key Points:**

* Each character in the string must belong to exactly one "TMT".
* Subsequence means: we can skip characters but must keep relative order.
* String length n is always divisible by 3.
* We need to answer YES or NO for each test case.

**🔹 How to Think**

1. Each "TMT" has **2 T’s** and **1 M**.  
   ✅ So:
   * Count of T = 2 \* count(M)  
     Otherwise → immediately **NO**.
2. Order must be valid:
   * For every "M", there must be a "T" **before it** (because "TMT" starts with a T).
   * For every "M", there must also be a "T" **after it** (because "TMT" ends with a T).

That means:

* + Left-to-right check: When scanning, at any point, count(M) ≤ count(T) (otherwise we would get an M without a left T).
  + Right-to-left check: When scanning backward, same rule: count(M) ≤ count(T) (otherwise M has no right T).

**🔹 Solution Approach**

1. Check frequency condition: count(T) == 2 \* count(M).  
   If not, → NO.
2. Left-to-right pass: ensure every M has a T before it.
3. Right-to-left pass: ensure every M has a T after it.
4. If all conditions satisfied → YES.

**Code :**

#include <bits/stdc++.h>

using namespace std;

#define ll long long

bool solve(string s)

{

    int m = 0, t = 0;

    for (int i = 0; i < s.size(); i++)

    {

        if (s[i] == 'M')

            m++;

        else

            t++;

        if (m > t)

            return false;

    }

    return true;

}

int main()

{

    int t;

    cin >> t;

    while (t--)

    {

        int n;cin >> n;

        string s;

        cin >> s;

        int M = 0, T = 0;

        for (int i = 0; i < s.size(); i++)

        {

            if (s[i] == 'T')

                T++;

            else

                M++;

        }

        if (2 \* M != T)

            cout << "NO" << endl;

        else

        {

            if (solve(s) == true)

            {

                reverse(s.begin(), s.end());

                if (solve(s) == true)

                    cout << "YES" << endl;

                else

                    cout << "NO" << endl;

            }

            else

                cout << "NO" << endl;

        }

    }

    return 0;

}

**🔹 Complexity Analysis**

* Counting T and M → **O(n)**.
* Left-to-right check → **O(n)**.
* Right-to-left check → **O(n)**.
* Total per test case: **O(n)**.
* Since sum of n ≤ 10^5, the solution is efficient.

**B. The Cake Is a Lie -** [**https://codeforces.com/contest/1519/problem/B**](https://codeforces.com/contest/1519/problem/B)

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int t;

    cin >> t;

    while (t--)

    {

        int n, m, k;

        cin >> n >> m >> k;

        int result = (n - 1) + (m - 1) \* n;

        if (result != k)

            cout << "NO" << endl;

        else

            cout << "YES" << endl;

    }

    return 0;

}

**B. Phoenix and Puzzle -** [**https://codeforces.com/contest/1515/problem/B**](https://codeforces.com/contest/1515/problem/B)

**🔹 Problem Statement (in my words)**

You are given n puzzle pieces, where each piece is a **right isosceles triangle** (two equal sides, one right angle).  
You must decide if it is possible to arrange **all n pieces** to form a **perfect square** (without gaps or overlaps).

For each test case, print **YES** if possible, otherwise **NO**.

**🔹 How to Think**

1. **Understand the piece**:
   * Each piece is a **right isosceles triangle**.
   * Two such triangles can form a **square of side length 1** (like joining them along the hypotenuse).
   * So, **n must be even** (otherwise, one triangle will always be left unused).

✅ First condition: n % 2 == 0

1. **Can every even n work?**  
   Let’s test:
   * n = 2 → Yes (forms a 1x1 square).
   * n = 4 → Yes (forms a √2 x √2 square).
   * n = 6 → ❌ Not possible.

So not all even n work.

1. **Key Insight**
   * If you take **2 triangles**, they form a square of area 1.
   * So, n triangles can form an area = n/2.
   * For this to be arranged as a **big square**, that area (n/2) must itself be a **perfect square**.

✅ Condition: (n / 2) should be a perfect square.

Example:

* + n = 2 → n/2 = 1 → perfect square → YES
  + n = 4 → n/2 = 2 → not square ❌ … but wait example says YES. Why?

1. **Careful Observation**
   * Another way: sometimes 4 triangles form a 1x1 square (not 2 triangles).
   * Actually, two possibilities exist:
     + Case 1: n / 2 is a perfect square.
     + Case 2: n / 4 is a perfect square.

Let’s check again:

* + n = 2 → n/2 = 1 (perfect square) → YES
  + n = 4 → n/4 = 1 (perfect square) → YES
  + n = 6 → n/2 = 3 (not square), n/4 = 1.5 (not integer) → NO ✅
  + n = 8 → n/2 = 4 (perfect square) → YES

**🔹 Final Solution Approach**

1. If n is **odd** → NO
2. Else check:
   * If (n / 2) is a perfect square → YES
   * Or (n / 4) is a perfect square → YES
   * Otherwise → NO

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

bool isPerfectSquare(long long x) {

long long r = sqrt(x);

return r \* r == x;

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

long long n;

cin >> n;

if (n % 2 == 0 && isPerfectSquare(n / 2)) {

cout << "YES\n";

}

else if (n % 4 == 0 && isPerfectSquare(n / 4)) {

cout << "YES\n";

}

else {

cout << "NO\n";

}

}

return 0;

}

**🔹 Complexity**

* **Square root check**: O(1)
* For each test case → O(1)
* For t test cases → O(t)

Memory usage is minimal (O(1)).

**Another Approach :**

**Code :**

#include<bits/stdc++.h>

using  namespace  std;

#define ll long long

int main()

{

    map<ll,ll>mp;

    ll ans=4;

    mp[2]=1;

    mp[4]=1;

    ll i=1;

    while(ans<=1e9)

    {

        ll an=i+i+1;

        an\*=4;

        ans+=an;

        mp[ans]=1;

        i++;

    }

    ans=2;

    i=1;

    while(ans<=1e9)

    {

        ll an=i+i+1;

        an\*=2;

        ans+=an;

        mp[ans]=1;

        i++;

    }

    int t;

    cin>>t;

    while(t--)

    {

        int n;

        cin>>n;

        if(mp[n]==1)cout<<"YES"<<endl;

        else cout<<"NO"<<endl;

    }

}

**B. Ordinary Numbers -** [**https://codeforces.com/contest/1520/problem/B**](https://codeforces.com/contest/1520/problem/B)

**🔎 Problem Understanding**

We are asked to count **ordinary numbers** between 1 and n.

👉 **Ordinary number**: A number where **all digits are the same**.  
Examples:

* ✅ 1, 2, 3, ..., 9, 11, 22, 33, ..., 99, 111, 222, ... are ordinary.
* ❌ 10, 12, 101, 2021 are not.

We are given t test cases.  
For each n, we need to output how many ordinary numbers exist between 1 and n.

**🔎 Example Walkthrough**

Input:

6

1

2

3

4

5

100

Output:

1

2

3

4

5

18

* For n = 1: Only {1} → **1**.
* For n = 2: {1, 2} → **2**.
* For n = 5: {1, 2, 3, 4, 5} → **5**.
* For n = 100:  
  Ordinary numbers are:
  + 1-digit: 1..9 (9 numbers).
  + 2-digit: 11, 22, 33, ..., 99 (9 numbers).
  + 3-digit ordinary numbers ≤ 100: Only 111 is > 100, so we stop.  
    ✅ Total = 9 + 9 = **18**.

**💡 How to Think About It**

1. Any **ordinary number** is of the form:  
   d, dd, ddd, dddd, ... where d ∈ [1..9].
   * For example: digit 7 generates → 7, 77, 777, 7777, ...
2. To count ordinary numbers ≤ n:
   * Check how many "digit-repeated" numbers exist that are ≤ n.
3. Key insight:
   * Count of ordinary numbers depends on:
     + The number of digits in n.
     + The first digit of n.

**✅ Solution Approach**

1. Convert n into string (to get length and first digit).
2. If n has k digits:
   * All ordinary numbers with **fewer digits** (1..k-1) are automatically valid.  
     → That gives (k-1) \* 9 ordinary numbers.  
     (For each digit 1..9, we can form exactly 1 ordinary number with len < k.)
   * For length = k, check how many exist:
     + Form a number like ddd...d (length k) for each digit 1..first\_digit.
     + Count only those ≤ n.

**🖥️ C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

long long n;

cin >> n;

string s = to\_string(n);

int len = s.size(); // number of digits

int first\_digit = s[0] - '0';

// (len - 1) \* 9 covers all ordinary numbers with fewer digits

int ans = (len - 1) \* 9;

// Build the number "ddd...d" with same length as n

long long candidate = 0;

for (int i = 0; i < len; i++) candidate = candidate \* 10 + first\_digit;

// If candidate <= n, we can count it; else only first\_digit-1

if (candidate <= n) ans += first\_digit;

else ans += (first\_digit - 1);

cout << ans << "\n";

}

return 0;

}

**📊 Complexity Analysis**

* **Time Complexity**:  
  For each test case:
  + Conversion to string → O(log n)
  + Simple arithmetic → O(1)  
    ✅ Total = O(t \* log n) (but since n ≤ 1e9, log n ≤ 10 → effectively O(t)).
* **Memory Complexity**: O(1) (just a few variables).

**✔ Example Dry Run**

Input:

n = 100

* len = 3, first\_digit = 1.
* Numbers with length < 3 → (3-1)\*9 = 18.
* Candidate with length 3 → 111. But 111 > 100, so don’t count.
* Add first\_digit - 1 = 0.
* ✅ Answer = 18.

**B. Nastia and a Good Array -** [**https://codeforces.com/contest/1521/problem/B**](https://codeforces.com/contest/1521/problem/B)

**Approach( from “WA” ) :**

index = 0 1 2 3 4 Operation

Array = 2 3 4 5 6 -------------------------

Mod Arr = x 2 4 5 6 0 1 X = x Y = min(2,3)=2

Mod Arr = x y 2 5 6 1 2 X = y Y = min(2,4)=2

Mod Arr = x y x 2 6 2 3 X = x Y = min(2,5)=2

Mod Arr = x 2 x y 2 1 2 X = y Y = min(2,6)=2

**🔹 Problem Statement (Rephrased in your words)**

We are given an array of nn positive integers.  
The array is called **good** if for every pair of consecutive elements:

gcd⁡(ai−1,ai)=1for all 2≤i≤n\gcd(a\_{i-1}, a\_i) = 1 \quad \text{for all } 2 \leq i \leq n

We are allowed an **operation**:

* Choose indices i≠ji \neq j,
* Choose numbers x,yx, y (with 1≤x,y≤2⋅1091 \leq x, y \leq 2 \cdot 10^9),
* Replace ai→xa\_i \to x, aj→ya\_j \to y, such that min⁡(ai,aj)=min⁡(x,y)\min(a\_i, a\_j) = \min(x, y).

The task:  
Make the array **good** using at most nn operations (not necessarily minimum).

**🔹 How to Think (Your Approach)**

1. **Goal:** Ensure consecutive elements are coprime.
2. You pick **two large prime numbers**:
   * x=1999999973x = 1999999973
   * y=1999999943y = 1999999943  
     Both are primes, so gcd⁡(x,y)=1\gcd(x,y) = 1.
3. Process the array from left to right:
   * At each step, replace the current element and the next element with either (x,min⁡(ai,ai+1))(x, \min(ai, ai+1)) or (y,min⁡(ai,ai+1))(y, \min(ai, ai+1)).
   * Alternate between x and y so that each consecutive pair contains at least one large prime.
4. This ensures all neighbors are coprime because:
   * gcd(prime, anything smaller) = 1 (unless that smaller is a multiple of the prime, which can’t happen here since your primes are huge).
   * gcd(x, y) = 1.

Thus, after n−1n-1 operations, the array is guaranteed to be good.

**🔹 Step-by-Step Example**

Suppose array = [2, 3, 4, 5, 6].

* Start with first pair (2, 3):  
  Replace with (x, min(2,3)=2) → Array = [x, 2, 4, 5, 6].
* Next pair (2, 4):  
  Replace with (y, min(2,4)=2) → Array = [x, y, 2, 5, 6].
* Next pair (2, 5):  
  Replace with (x, min(2,5)=2) → Array = [x, y, x, 2, 6].
* Next pair (2, 6):  
  Replace with (y, min(2,6)=2) → Array = [x, 2, x, y, 2].

Now, every consecutive pair has gcd = 1.

**🔹 C++ Code**

#include <bits/stdc++.h>

using namespace std;

int main() {

int x, y;

// Pick two large primes

x = 1999999973;

y = 1999999943;

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

int ar[n+3];

for (int i = 0; i < n; i++) {

cin >> ar[i];

}

// Always use n-1 operations

cout << n-1 << endl;

for (int i = 0; i < n-1; i++) {

if (i % 2 == 0) {

// Even index: use prime x

cout << i+1 << " " << i+2 << " " << x << " " << min(ar[i], ar[i+1]) << endl;

ar[i+1] = min(ar[i], ar[i+1]); // keep min condition satisfied

} else {

// Odd index: use prime y

cout << i+1 << " " << i+2 << " " << y << " " << min(ar[i], ar[i+1]) << endl;

ar[i+1] = min(ar[i], ar[i+1]);

}

}

}

return 0;

}

**🔹 Complexity Analysis**

1. **Finding min for each pair:** O(1)O(1).
2. **Looping through n-1 pairs:** O(n)O(n).
3. For all test cases:

O(∑n)≤2⋅105O\left(\sum n\right) \leq 2 \cdot 10^5

which is very efficient.

* **Time Complexity:** O(n) per test case.
* **Space Complexity:** O(n) for storing the array.

**🔹 Summary (According to Your Approach)**

* The problem is about making the array **pairwise coprime**.
* You fix two **large coprime numbers (primes)** x, y.
* For each consecutive pair, replace with (x, min(ai, ai+1)) or (y, min(ai, ai+1)).
* Alternate between x and y.
* Always takes exactly n-1 operations, which is within the allowed limit.
* Complexity: **O(n)**.

**B1. Palindrome Game (easy version) -** [**https://codeforces.com/contest/1527/problem/B1**](https://codeforces.com/contest/1527/problem/B1)

**🔹 Problem Statement**

We are given a game problem involving a binary string (0 and 1).

* **Players:** Alice and Bob.
* **Game Rule (inferred):**
  + Each player alternately picks and changes 0s into 1s (or removes them, depending on original problem statement).
  + The game ends when there are no more 0s.
* **Who wins?**
  + Based on the number of 0s (cntZero), the winner is decided.

**🔹 How to Think**

The key lies in **counting the number of zeros** (cntZero), because the game only depends on the availability of 0s (since 1s don’t affect turns).

* If there are **no zeros (cntZero == 0)**, the game cannot start → it’s a **DRAW**.
* If the number of zeros is **even**, Bob will win → because after equal moves, Bob plays last.
* If the number of zeros is **odd**:
  + If there is **only 1 zero**, Bob will still win → because Alice cannot force a winning strategy.
  + Otherwise (odd and >1), Alice wins → because Alice can always control the last move.

**🔹 Solution Approach (according to your code)**

1. Read number of test cases t.
2. For each test case:
   * Input n (length of the string) and the binary string s.
   * Count how many 0s are in the string.
   * Apply the decision rules:
     + If cntZero == 0 → **DRAW**
     + Else if cntZero % 2 == 0 → **BOB**
     + Else if cntZero == 1 → **BOB**
     + Else → **ALICE**

**🔹 C++ Solution (Your Code, Explained)**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int t;

cin >> t;

while (t--)

{

int n;

cin >> n;

string s;

cin >> s;

int cntZero = 0;

for (int i = 0; i < n; i++)

{

if (s[i] == '0')

cntZero++;

}

if (cntZero == 0) {

cout << "DRAW" << endl;

}

else if (cntZero % 2 == 0) {

cout << "BOB" << endl;

}

else {

if (cntZero == 1)

cout << "BOB" << endl;

else

cout << "ALICE" << endl;

}

}

return 0;

}

**🔹 Complexity Analysis**

* **Counting Zeros:** You loop over the string once → O(n).
* **Decision Rules:** Constant time checks → O(1).
* **Per Test Case Complexity:** O(n)
* **For t test cases:** O(t \* n)

**Space Complexity:**

* Uses only a few integers and the string s → O(n) for storing the string.

✅ This matches your approach perfectly:

* You solved it by **counting zeros** and applying simple game-theory rules.
* No extra data structures, just direct simulation.

**B2. Palindrome Game (hard version) -** [**https://codeforces.com/contest/1527/problem/B2**](https://codeforces.com/contest/1527/problem/B2)

**🔹 Problem Statement (B2 Hard Version)**

Alice and Bob play a game on a binary string s of length n:

1. **Operation 1:** Choose an index i where s[i] = '0', change it to '1', and pay **1 dollar**.
2. **Operation 2:** Reverse the string (pay **0 dollars**), but:
   * Only allowed if the string is **not a palindrome**.
   * Cannot be performed twice in a row.

The game ends when the string has **all '1's**.

* The **winner** is the player who spends fewer dollars.
* If both spend the same, the result is a **DRAW**.

We must determine the outcome ("ALICE", "BOB", or "DRAW") when both play optimally.

**🔹 How to Think**

Your solution **divides the problem into two cases**:

**Case 1: s is initially a palindrome**

* Then **reversing is not possible at the start**, so only the **first operation** (changing 0 → 1) is available.
* The game is reduced to the **easy version (B1)**:
  + Count the number of 0s (cntZero).
  + If cntZero == 0 → game already finished → **DRAW**.
  + If cntZero is even → Bob wins.
  + If cntZero == 1 → Bob wins (since Alice must move first).
  + Otherwise (cntZero odd and > 1) → Alice wins.

**Case 2: s is NOT a palindrome**

* Now reversing can be used strategically (sometimes forcing the opponent to pay more).
* The **special tricky case**:
  + If cntZero == 2, n is odd, and the **middle character is '0'**, then → **DRAW**.  
    Why? Because Alice’s best strategy and Bob’s response balance out exactly.
* Otherwise, Alice can always find a strategy to force Bob to pay more → **Alice wins**.

**🔹 Solution Approach**

1. For each test case, read n and string s.
2. Count the number of zeros (cntZero).
3. If s is a palindrome:
   * Apply the **easy version rules**.
4. If s is not a palindrome:
   * Check the **special draw condition** (cntZero == 2 && n odd && middle == '0').
   * Otherwise, Alice wins.

**🔹 C++ Solution (Code)**

#include <bits/stdc++.h>

using namespace std;

bool isPalindrome(string s)

{

string s1 = s;

reverse(s1.begin(), s1.end());

return s == s1;

}

int main()

{

int t;

cin >> t;

while (t--)

{

int n;

cin >> n;

string s;

cin >> s;

int cntZero = 0;

for (int i = 0; i < n; i++)

{

if (s[i] == '0') cntZero++;

}

if (isPalindrome(s))

{

if (cntZero == 0) {

cout << "DRAW" << endl;

}

else if (cntZero % 2 == 0) {

cout << "BOB" << endl;

}

else {

if (cntZero == 1) cout << "BOB" << endl;

else cout << "ALICE" << endl;

}

}

else

{

if (cntZero == 2 && n % 2 == 1 && s[n/2] == '0')

cout << "DRAW" << endl;

else

cout << "ALICE" << endl;

}

}

return 0;

}

**🔹 Complexity Analysis**

* **Palindrome check:** O(n)
* **Counting zeros:** O(n)
* **Decision making:** O(1)
* **Per test case complexity:** O(n)
* **For t test cases:** O(t \* n)

**Space Complexity:**

* Storing string s: O(n)
* No extra heavy data structures.

✅ So according to **your code**, the problem boils down to:

* **If palindrome → use easy game rules (cntZero decides).**
* **If not palindrome → either DRAW in 1 tricky case, else ALICE wins.**

**B. Sifid and Strange Subsequences -** [**https://codeforces.com/contest/1529/problem/B**](https://codeforces.com/contest/1529/problem/B)

**🔎 Problem Restatement (in simple words)**

We are given an array **a** of size **n**.  
We want to find the **longest subsequence** that is **"strange"**, meaning:

For every pair (i,j)(i,j),

∣ai−aj∣≥MAX|a\_i - a\_j| \geq \text{MAX}

where **MAX** is the maximum element of that subsequence.

* Any sequence of size ≤ 1 is automatically strange.
* Goal: maximize the length of the strange subsequence.

**💡 How to Think About It**

1. Sort the array.
   * Sorting makes it easier to compare differences between consecutive numbers.
2. If you only take **non-positive numbers (≤ 0)**, the condition holds **as long as the minimum gap between them is large enough**.
   * Because negatives are smaller, the maximum element in that subsequence will be ≤ 0 or small, so they can "fit together".
3. Adding a **positive number** is tricky:
   * Suppose you already have non-positives. Their **minimum difference (gap)** matters.
   * If you want to add a **positive number x**, then it must not be larger than that minimum gap.
   * Otherwise, condition fails (since |small - small| < x).

So the strategy:

* Take as many non-positives as possible (while maintaining min difference).
* Then, check if the **smallest positive number** can be added — only if it’s ≤ min difference.

**📝 Solution Approach (what the code does)**

1. Read n and the array.
2. Sort the array.
3. Start with ans = 1 (at least one element can always form a strange subsequence).
4. Track maxx = INT\_MAX, which represents the **minimum gap so far** between consecutive elements.
5. Iterate through sorted array:
   * Update maxx = min(maxx, ara[i] - ara[i-1]).
   * If the current element ara[i] is **larger than maxx**, stop (cannot include this element).
   * Otherwise, increase ans.
6. Print ans.

**🧮 Example Walkthrough**

Input:

1

4

-1 -2 0 0

Sorted: [-2, -1, 0, 0]

* Start: ans = 1
* Compare (-2, -1): gap = 1 → maxx = 1 → -1 ≤ 1 → include → ans = 2
* Compare (-1, 0): gap = 1 → maxx = 1 → 0 ≤ 1 → include → ans = 3
* Compare (0, 0): gap = 0 → maxx = 0 → current 0 ≤ 0 → include → ans = 4

Output: 4 ✅

**⚡ Complexity**

* Sorting: **O(n log n)**
* Single pass through array: **O(n)**
* Total per test case: **O(n log n)**
* Given constraints (∑n≤105\sum n \leq 10^5), this is efficient.

**✅ C++ Code (same as yours, explained)**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<int> ara(n);

for (int i = 0; i < n; i++) cin >> ara[i];

sort(ara.begin(), ara.end());

int ans = 1;

int maxx = INT\_MAX;

for (int i = 1; i < n; i++) {

maxx = min(maxx, ara[i] - ara[i - 1]);

if (ara[i] > maxx) break;

ans++;

}

cout << ans << "\n";

}

return 0;

}

✅ So the code correctly solves the **longest strange subsequence** problem using sorting + greedy construction based on minimum gap.

**B. Spreadsheets -** [**https://codeforces.com/contest/1/problem/B**](https://codeforces.com/contest/1/problem/B)

**🔹 Problem Statement**

We are working with **two types of spreadsheet cell notations** (similar to Microsoft Excel or Google Sheets):

1. **RxCy Format**:
   * Example: R23C55
   * Means **Row = 23, Column = 55**
2. **Column-Row Format** (Excel style):
   * Example: BC23
   * Means **Column = BC, Row = 23**
   * Here, BC represents the 55th column (A=1, B=2, ..., Z=26, AA=27, AB=28, ...).

👉 The task is to **convert between these two formats**.

* If input is in **RxCy**, output should be in **ColumnRow** format.
* If input is in **ColumnRow**, output should be in **RxCy** format.

**🔹 Example**

**Input:**

4

R23C55

BC23

R1C1

ZZ99

**Output:**

BC23

R23C55

A1

R99C702

**🔹 How to Think**

1. **Identify input format**:
   * If the string starts with R and later contains C, it’s in **RxCy** format.
   * Otherwise, it’s in **ColumnRow** format.
2. **Conversion rules**:
   * From RxCy → convert column number to string (like base-26 with letters A-Z).
   * From ColumnRow → convert letters to a number (like converting base-26).

**🔹 Solution Approach**

1. **Parse the string** into parts: letters and digits.
   * Example: "R23C55" → "23" and "55"
   * Example: "BC23" → "BC" and "23"
2. **Case 1: RxCy → ColumnRow**
   * Row = first number (after R, before C)
   * Col = second number (after C)
   * Convert column number to Excel letters:
   * 1 -> A
   * 26 -> Z
   * 27 -> AA
   * 28 -> AB
   * ...

(This is similar to converting a number to base-26 but digits are A–Z instead of 0–25).

1. **Case 2: ColumnRow → RxCy**
   * Row = digits after letters.
   * Convert letters → column number:
   * A -> 1
   * Z -> 26
   * AA -> 27
   * AB -> 28
   * ...

(This is like converting base-26 with A=1 instead of 0).

1. Print the result.

**🔹 Complexity Analysis**

* **Parsing string** → O(n) where n = length of string.
* **Conversion base-26** (number ↔ letters) → O(log(col)) (at most ~7 for Excel max).
* So total = **O(n)** per query.
* For t test cases → **O(t \* n)**.

Memory = O(1) (only a few strings and integers stored).

**🔹 Clean C++ Implementation**

Here’s a neat version of your code:

#include <bits/stdc++.h>

using namespace std;

string numToStr(int num) {

string res = "";

while (num > 0) {

num--; // adjust because A=1 not 0

res = char('A' + (num % 26)) + res;

num /= 26;

}

return res;

}

int strToNum(string s) {

int res = 0;

for (char c : s) {

res = res \* 26 + (c - 'A' + 1);

}

return res;

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(0);

int t; cin >> t;

while (t--) {

string s; cin >> s;

if (s[0] == 'R' && s.find('C') != string::npos && isdigit(s[1])) {

// Case 1: RxCy format

int posC = s.find('C');

int row = stoi(s.substr(1, posC - 1));

int col = stoi(s.substr(posC + 1));

cout << numToStr(col) << row << "\n";

} else {

// Case 2: ColumnRow format

int i = 0;

string colStr = "";

while (i < s.size() && isalpha(s[i])) colStr += s[i++];

int row = stoi(s.substr(i));

int col = strToNum(colStr);

cout << "R" << row << "C" << col << "\n";

}

}

return 0;

}

**B. I Hate 1111 -** [**https://codeforces.com/contest/1526/problem/B**](https://codeforces.com/contest/1526/problem/B)

**🔹 Problem Restatement**

We are given a number x.  
We need to check if it can be represented as the sum of numbers of the form:

11, 111, 1111, 11111, ...

Basically numbers made only of 1s (with at least 2 digits).  
We can use any of them, **any number of times**.

For example:

* 33 = 11 + 11 + 11 → YES
* 144 = 111 + 11 + 11 + 11 → YES
* 69 → NO

We must answer **YES/NO** for each test case.

**🔹 How to Think**

**Step 1: What numbers are allowed?**

We can use 11, 111, 1111, ....  
Notice that **all of them are congruent to either 0 or 1 (mod 11)**:

* 11 % 11 = 0
* 111 % 11 = 1
* 1111 % 11 = 1
* 11111 % 11 = 1

So effectively:

* We have unlimited 11s,
* We have unlimited numbers ≡ 1 mod 11.

**Step 2: Key Observation**

If we want to form x, we can write:

x = 11 \* a + 111 \* b

(or with any larger "1111..1", but it behaves like 111 mod 11 anyway).

So the main idea:

* Try to see if some combination of 11s and 111s adds up to x.

**Step 3: Simplify**

Let’s fix b (the count of 111s). Then:

x - 111 \* b must be divisible by 11

So, the condition is:

∃ b ≥ 0 such that (x - 111 \* b) % 11 == 0 and x - 111\*b ≥ 0

**Step 4: Bound on b**

Since x ≤ 10^9, we cannot check all b.  
But notice:

* If b > 11, then 111\*b already covers all residues mod 11 (because 111 ≡ 1 mod 11).  
  So we only need to check up to b = 11.

✅ **Conclusion**:  
For each test x, check b = 0..11.  
If (x - 111\*b) % 11 == 0 and non-negative → **YES**. Otherwise → **NO**.

**🔹 Example Walkthrough**

Input:

3

33

144

69

1. 33: Try b=0 → (33 % 11 == 0) → YES.
2. 144: Try b=1 → 144-111=33, divisible by 11 → YES.
3. 69:
   * b=0 → 69%11=3
   * b=1 → 69-111 < 0 stop.  
     No valid b → NO.

Output:

YES

YES

NO

**🔹 C++ Solution**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

long long x;

cin >> x;

bool ok = false;

for (int b = 0; b <= 11; b++) {

if (x >= 111LL \* b && (x - 111LL \* b) % 11 == 0) {

ok = true;

break;

}

}

cout << (ok ? "YES\n" : "NO\n");

}

}

**🔹 Complexity**

* For each test, we try at most 12 values of b.
* So complexity = **O(12 \* t) ≈ O(t)**.
* With t ≤ 10000, this is very efficient.

✅ So the trick was to reduce it to checking **x = 111*b + 11*a**, then bound b up to 11.

Another Approach :

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int t;

    cin >> t;

    while (t--)

    {

        int n;

        cin >> n;

        bool flag = false;

        for (int i = 0; i \* 111 <= n; i++)

        {

            if ((n - 111 \* i) % 11 == 0)

            {

                flag = true;

                break;

            }

        }

        if (flag)

            cout << "YES" << endl;

        else

            cout << "NO" << endl;

    }

    return 0;

}

**B. Lord of the Values -** [**https://codeforces.com/contest/1523/problem/B**](https://codeforces.com/contest/1523/problem/B)

**🔹 Problem Restatement**

We are given n integers (n is always even).  
We want to transform them into their **negatives** (a1 → -a1, a2 → -a2, …).

We are allowed **two operations** on indices i < j:

1. ai = ai + aj
2. aj = aj - ai

Constraints:

* 2 ≤ n ≤ 1000 (so up to 1000 variables),
* Each ai ≤ 10^9,
* At most 5000 operations,
* After every operation, values must stay within [-10^18, 10^18].

We must print a sequence of operations that achieves the transformation.

**🔹 How to Think**

We are asked to **invert all numbers** using only "pair operations".  
Notice:

* Since n is even, we can process the array in **pairs** (a1, a2), (a3, a4), ….
* In each pair, we want to turn (x, y) into (-x, -y).

So the problem boils down to:  
👉 Find a sequence of ≤ 6–8 operations to flip **one pair**.

**Step 1: Try with one pair (a, b)**

We want (a, b) → (-a, -b).

A known trick:

(1) op2(i, j) → b = b - a

(2) op1(i, j) → a = a + b

(3) op2(i, j) → b = b - a

(4) op1(i, j) → a = a + b

(5) op2(i, j) → b = b - a

(6) op1(i, j) → a = a + b

After these 6 steps, (a, b) becomes (-a, -b).  
(This sequence is derived from simulating modular elimination tricks, like in Euclidean algorithm.)

So **one pair takes exactly 6 operations**.

**Step 2: Apply to all pairs**

Since n is even, just apply this 6-operation sequence on pairs (1,2), (3,4), ….

Total operations = 6 \* (n/2) = 3n.

With n ≤ 1000,  
3n = 3000 ≤ 5000 ✅ within the limit.

**Step 3: Bound check**

Do numbers exceed 10^18?

* Each op is of the form a+b or b-a.
* Starting numbers ≤ 10^9.
* In worst case, values can grow by constant factors across operations.
* But since n=1000 and we apply only 6 ops per pair independently, the growth stays far below 10^18.  
  So constraints are satisfied.

**🔹 Solution Approach**

1. Process array in pairs (a[i], a[i+1]).
2. For each pair, apply the fixed 6-operation sequence.
3. Collect and print all operations.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<long long> a(n);

for (int i = 0; i < n; i++) cin >> a[i];

vector<array<int,3>> ops; // {type, i, j}

// Process in pairs

for (int i = 1; i <= n; i += 2) {

int j = i + 1;

// Apply 6-step fixed sequence

ops.push\_back({2, i, j});

ops.push\_back({1, i, j});

ops.push\_back({2, i, j});

ops.push\_back({1, i, j});

ops.push\_back({2, i, j});

ops.push\_back({1, i, j});

}

cout << ops.size() << "\n";

for (auto &op : ops) {

cout << op[0] << " " << op[1] << " " << op[2] << "\n";

}

}

}

**🔹 Complexity**

* For each test case:
  + Process each pair in **O(1)** with exactly 6 ops.
  + Total = O(n) operations.
* Since n ≤ 1000, t ≤ 20, complexity = **O(t·n) ≤ 20000**.  
  Very efficient.

✅ So the trick: Use a **6-step fixed sequence per pair**.  
That ensures all numbers flip signs, stays within limits, and uses ≤ 5000 operations.

Another Proces :

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int t;cin >> t;

    while (t--)

    {

        int n;cin >> n;

        int ar[n + 4];

        for (int i = 1; i <= n; i++)

        {

            cin >> ar[i];

        }

        vector<pair<int, pair<int, int>>> v;

        for (int i = 1; i <= n; i += 2)

        {

            int a = i;

            int b = i + 1;

            if (ar[i] != ar[i + 1])

            {

                v.push\_back({1, {a, b}});

                v.push\_back({2, {a, b}});

                v.push\_back({1, {a, b}});

                v.push\_back({1, {a, b}});

                v.push\_back({2, {a, b}});

                v.push\_back({1, {a, b}});

            }

            else

            {

                v.push\_back({2, {a, b}});

                v.push\_back({2, {a, b}});

                v.push\_back({1, {a, b}});

                v.push\_back({1, {a, b}});

            }

        }

        cout << v.size() << endl;

        for (int i = 0; i < v.size(); i++)

        {

            cout << v[i].first << " " << v[i].second.first << " " << v[i].second.second << endl;

        }

    }

    return 0;

}

**B. Array Reodering -** [**https://codeforces.com/contest/1535/problem/B**](https://codeforces.com/contest/1535/problem/B)

**🔹 Problem Restatement**

We are given an array a of size n.  
A pair (i, j) (with i < j) is **good** if:

gcd(ai,  2aj)>1\text{gcd}(a\_i, \; 2a\_j) > 1

We can reorder the array however we like.  
We need to find the **maximum number of good pairs** after reordering.

**🔹 How to Think**

**Step 1: Understand the GCD condition**

We need:

gcd⁡(ai,2aj)>1\gcd(a\_i, 2a\_j) > 1

That means a\_i and 2a\_j share at least one common factor greater than 1.

Key observations:

* If a\_i is **even**, then gcd(a\_i, 2a\_j) is **always ≥ 2** (since both are divisible by 2).  
  ✅ So, **any pair where a\_i is even is always good**.
* If a\_i is **odd**, then it depends on whether gcd(a\_i, a\_j) > 1.

So, the **strategy** is:  
👉 Place all **even numbers in front**, then odds.  
Why? Because then for each even a\_i, it forms good pairs with **all elements after it**.

**Step 2: Counting Good Pairs**

1. **When a\_i is even**:
   * If a\_i is at position i, then it makes good pairs with **all n - i elements after it**.
2. **When both are odd**:
   * Need to check if gcd(a\_i, a\_j) > 1.

So the solution is:

* Reorder array → evens first, then odds.
* Count good pairs:
  + Automatic good pairs for evens.
  + Among odds, check gcd.

**Step 3: Example**

Example: [3, 6, 5, 3]  
Reorder → [6, 3, 5, 3].

* a1=6 (even): forms pairs with (3,5,3) → 3 pairs.
* Odds: (3,5), (3,3), (5,3).
  + gcd(3,5)=1 → not good.
  + gcd(3,3)=3 → good.
  + gcd(5,3)=1 → not good.  
    So total = 4 good pairs ✅.

**🔹 Solution Approach**

1. Read input.
2. Reorder array: evens first, odds later.
3. Initialize counter ans = 0.
4. Double loop over pairs (i, j):
   * If gcd(a[i], 2\*a[j]) > 1, increment counter.
5. Output result.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<int> a(n);

for (int i = 0; i < n; i++) cin >> a[i];

// Step 1: reorder → evens first, odds later

sort(a.begin(), a.end(), [](int x, int y) {

return (x % 2 == 0) && (y % 2 == 1);

});

long long ans = 0;

// Step 2: count good pairs

for (int i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

if (gcd(a[i], 2 \* a[j]) > 1) ans++;

}

}

cout << ans << "\n";

}

return 0;

}

**🔹 Complexity**

* Sorting: **O(n log n)**
* Pair checking: **O(n²)**
  + Max n = 2000, so n² = 4,000,000 per test case → efficient.
* Total across all test cases: ≤ 4M checks (since sum of n ≤ 2000) → fine within 2s.

✅ So the **trick** was:

* Put evens first (they guarantee pairs).
* For odds, only check gcd manually.

Another Approach :

#include<bits/stdc++.h>

using  namespace  std;

int main()

{

    int t;cin>>t;

    while(t--)

    {

        int n,x;cin>>n;

        vector<int>v,v1;

        for(int i=0;i<n;i++)

        {

            cin>>x;

            if(x%2==0)v.push\_back(x);

            else v1.push\_back(x);

        }

        int ans=0;

        int z=1;

        for(int i=0;i<v.size();i++)

        {

            ans+=n-z;

            z++;

        }

        int ans1=0;

        sort(v1.begin(),v1.end());

        for(int i=0;i<v1.size();i++)

        {

            for(int j=i+1;j<v1.size();j++)

            {

                if(\_\_gcd(v1[i],v1[j])>1)ans1++;

            }

        }

        cout<<ans+ans1<<endl;

    }

    return 0;

}

**B. Prinzessin der Verurteilung -** [**https://codeforces.com/contest/1536/problem/B**](https://codeforces.com/contest/1536/problem/B)

**🔹 Problem Restatement**

We are given a histogram of n bars with heights a1, a2, …, an.

* Operation: Pick an index i with ai > 0, and decrease ai by 1.
* Ugliness = (vertical outline length of histogram) + (number of operations performed).
* We want to minimize ugliness.

**🔹 Step 1: What is "vertical outline length"?**

The vertical outline includes:

1. The **left side** of the first bar = a1.
2. The **right side** of the last bar = an.
3. Between bars i and i+1, we add |ai − ai+1|.

So vertical outline =

a1+an+∑i=1n−1∣ai−ai+1∣a\_1 + a\_n + \sum\_{i=1}^{n-1} |a\_i - a\_{i+1}|

**🔹 Step 2: Effect of operations**

If we reduce some bar’s height, it affects:

* Vertical length (a1, an, or differences with neighbors).
* Plus, each reduction adds **+1** to operations count.

So whenever we reduce, the ugliness changes in two opposite directions:

* ↓ Vertical outline decreases.
* ↑ Operation cost increases.

We want the best trade-off.

**🔹 Step 3: Key Insight**

Look at each **bar i**:

* Suppose ai is **higher than both neighbors**. Then the extra height above max(ai−1, ai+1) contributes to vertical outline unnecessarily.  
  → We can reduce ai down to max(ai−1, ai+1) without hurting anything.  
  → But each reduction also costs 1 operation.

❗BUT here’s the twist: The decrease in vertical outline is exactly equal to the number of operations needed.  
So total ugliness stays the same.

Thus, **no need to actually simulate operations**. We can directly compute the minimal ugliness by "cutting down" the histogram virtually.

**🔹 Step 4: Simplified Formula**

We only keep **positive rises** in the histogram.

* At the left border: cost = a1.
* Between bars: cost += max(0, ai − ai−1).
* At the right border: cost += an.

So minimal ugliness =

a1+an+∑i=1n−1max⁡(0,ai−ai−1)a\_1 + a\_n + \sum\_{i=1}^{n-1} \max(0, a\_i - a\_{i-1})

**🔹 Step 5: Verify with Examples**

**Example 1:**

[4, 8, 9, 6]

* Left: 4
* Right: 6
* Differences: (8−4)=4, (9−8)=1, (6−9)=−3 → max(0,−3)=0  
  → Total = 4+6+4+1 = 15 ❌ Wait, sample says 17.

Fix: We must be careful. The formula must include **absolute differences** but reduce local peaks. Let's re-check:

Better version:  
Minimal ugliness =

a1+an+∑i=1n−1∣ai−ai+1∣−(excess cuts)a\_1 + a\_n + \sum\_{i=1}^{n-1} |a\_i - a\_{i+1}| - \text{(excess cuts)}

But notice from editorial: The real trick is:

Answer=a1+an+∑i=1n−1∣ai−ai+1∣−∑i=1nmax⁡(0,ai−max⁡(ai−1,ai+1))\text{Answer} = a\_1 + a\_n + \sum\_{i=1}^{n-1} |a\_i - a\_{i+1}| - \sum\_{i=1}^n \max(0, a\_i - \max(a\_{i-1}, a\_{i+1}))

But since we argued operations cancel out, an even simpler way is:

👉 Final formula (from accepted solutions):

Answer=a1+an+∑i=1n−1∣ai−ai+1∣−∑i=1nmax⁡(0,ai−max⁡(ai−1,ai+1))\text{Answer} = a\_1 + a\_n + \sum\_{i=1}^{n-1} |a\_i - a\_{i+1}| - \sum\_{i=1}^{n} \max(0, a\_i - \max(a\_{i-1}, a\_{i+1}))

where a0 = an+1 = 0.

Checking [4,8,9,6]:

* Outline = 4 + 6 + (|4−8|+|8−9|+|9−6|) = 10 + (4+1+3) = 18
* Reductions: only a3=9 > max(8,6)=8, excess=1  
  → Answer = 18−1=17 ✅ matches.

Example 2: [2,1,7,4,0,0]

* Outline = 2+0+(|2−1|+|1−7|+|7−4|+|4−0|+|0−0|)  
  = 2+0+(1+6+3+4+0)=16
* Reductions: a3=7 > max(1,4)=4 → excess=3  
  → Answer = 16−3=13 ❌ sample says 12.

Check again:  
Also a1=2 > max(0,1)=1 → excess=1  
Total excess=4 → 16−4=12 ✅ matches.

Perfect.

**🔹 Step 6: Algorithm**

1. Compute outline = a1 + an + sum(|ai−ai+1|).
2. Compute total reductions:  
   For each i, excess = max(0, ai − max(ai−1, ai+1)).  
   (Set a0=an+1=0 for borders).
3. Answer = outline − total reductions.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<long long> a(n+2, 0); // pad with 0 at both ends

for (int i = 1; i <= n; i++) cin >> a[i];

long long outline = a[1] + a[n];

for (int i = 1; i < n; i++)

outline += abs(a[i] - a[i+1]);

long long excess = 0;

for (int i = 1; i <= n; i++) {

excess += max(0LL, a[i] - max(a[i-1], a[i+1]));

}

cout << outline - excess << "\n";

}

return 0;

}

**🔹 Complexity**

* **Time:** O(n) per test case.
* **Memory:** O(n).
* With total n ≤ 4⋅10^5, it runs fine within 2s.

✅ That’s the minimal ugliness formula and solution.

**Another Approach :**

#include<bits/stdc++.h>

using  namespace  std;

string ss;

void num\_to\_str(int num)

{

    if(num)

    {

        num\_to\_str((num-1)/26);

        ss+=('a'+ (num-1)%26);

    }

}

int main()

{

    int t;

    cin>>t;

    while(t--)

    {

        int n;

        cin>>n;

        string s;

        cin>>s;

        string ans;

        int i=1;

       while(1)

       {

           ss="";

           num\_to\_str(i);

           int pos=s.find(ss);

           if(pos==-1)

           {

               ans=ss;

               break;

           }

           i++;

       }

       cout<<ans<<endl;

    }

}

**B. Pleasant Pairs -** [**https://codeforces.com/contest/1541/problem/B**](https://codeforces.com/contest/1541/problem/B)

**🔹 Problem Restatement**

We are given an array a[1..n] of **distinct integers**.  
We need to count the number of pairs (i, j) such that:

* i < j
* a[i] \* a[j] == i + j

We must do this efficiently for up to **10^4 test cases** and total n ≤ 2\*10^5.

**🔹 Understanding with Example**

Example: a = [3, 1, 5, 9, 2]

* Pair (1, 2) → a[1]\*a[2] = 3\*1 = 3, i+j = 1+2 = 3 ✅
* Pair (1, 5) → a[1]\*a[5] = 3\*2 = 6, i+j = 1+5 = 6 ✅
* Pair (2, 3) → a[2]\*a[3] = 1\*5 = 5, i+j = 2+3 = 5 ✅  
  So answer = 3.

**🔹 Key Observations**

1. **Naive approach** → check all pairs (i,j) → O(n^2) → impossible for n=10^5.
2. Rearrange condition:
3. a[i] \* a[j] = i + j

Since i+j ≤ 2n (max index sum), we only need to check products up to 2n.

1. Suppose we fix i. Then:
2. i + j = a[i] \* a[j]
3. → j = a[i] \* a[j] - i

So if we know a[j], we can directly check if j is valid.

**🔹 Efficient Strategy**

* Precompute **position array**: pos[value] = index for all values in a.  
  (since all elements are distinct and 1 ≤ a[i] ≤ 2n)
* Now iterate over i = 1..n and for each a[i]:
  + Try multiples of a[i] (say k \* a[i]) such that ≤ 2n.
  + Check if that multiple can be written as i + pos[k].
* Why efficient?  
  Because the product a[i]\*a[j] must be ≤ 2n.  
  So for each a[i], number of multiples is about O(2n / a[i]).  
  Summed over all elements, this is roughly O(n log n).

**🔹 Complexity**

* Building position array: **O(n)**
* Iterating with multiples: **O(n log n)** (harmonic series bound)
* Across all test cases: **O(Σ n log n) ≤ ~ 2*10^5 log(2*10^5)** → efficient ✅

**🔹 C++ Solution**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

int n;

cin >> n;

vector<int> a(n+1), pos(2\*n+2, -1);

for (int i = 1; i <= n; i++) {

cin >> a[i];

pos[a[i]] = i; // store position of each value

}

long long ans = 0;

for (int i = 1; i <= n; i++) {

for (long long mul = a[i]; mul <= 2\*n; mul += a[i]) {

int j = mul - i; // candidate index

if (j > i && j <= n && pos[mul / a[i]] == j) {

ans++;

}

}

}

cout << ans << "\n";

}

return 0;

}

**🔹 Walkthrough with Example**

t=1, n=5, a = [3,1,5,9,2]

* Build pos:
  + pos[3]=1, pos[1]=2, pos[5]=3, pos[9]=4, pos[2]=5
* i=1 (a[1]=3): try multiples 3,6,9,12…
  + mul=6 → j=5, pos[2]=5 ✅ pair (1,5)
* i=2 (a[2]=1): try multiples 1,2,3,4,5…
  + mul=3 → j=1 (not valid since j<i)
  + mul=5 → j=3, pos[5]=3 ✅ pair (2,3)
* i=3 (a[3]=5): try multiples 5,10,…
  + mul=5 → j=2 (not valid since j<i)
* i=…
  + finds (1,2) earlier.

Answer = 3 ✅

👉 So the key trick is:  
**Only check pairs whose product ≤ 2n, using a position lookup.**

Another Approach :

#include<bits/stdc++.h>

using namespace std;

#define ll long long

int main()

{

int t;cin>>t;

while(t--)

{

int n;cin>>n;

int ar[n+2];

for(int i=1;i<=n;i++){

cin>>ar[i];

}

ll ans=0;

for(int i=2;i<=n;i++)

{

int z=i+i-1;

if(ar[i]>z)

continue;

int x=(z%ar[i]+1);

int y=i-x;

while(y>=1)

{

if(y+i==ar[i]\*ar[y])

ans++;

y-=ar[i];

}

}

cout<<ans<<endl;

}

}

**B. Plus and Multiply -** [**https://codeforces.com/problemset/problem/1542/B**](https://codeforces.com/problemset/problem/1542/B)

**🔹 Problem Restatement**

We define an **infinite set** of integers:

* 1 is in the set.
* If x is in the set:
  + x \* a is in the set
  + x + b is in the set

Given integers n, a, b, check if n belongs to this set.

Constraints:

* up to 10^5 test cases
* 1 ≤ n, a, b ≤ 10^9

**🔹 Key Observations**

1. Starting from 1, the set grows by **multiplication by a** and **addition of b**.

Example: a=3, b=5

* + 1
  + Multiply: 3
  + Add: 6
  + From 3 → 9, from 6 → 11, etc.

1. We want to check if **we can reach n** starting from 1.

**🔹 Special Case 1: a = 1**

* Then multiplying doesn’t change anything (x \* 1 = x).
* The only operation that matters is x + b.
* So the set is: 1, 1+b, 1+2b, 1+3b, …
* Condition:
* n ≡ 1 (mod b)
* i.e., (n-1) % b == 0.

**🔹 General Case: a > 1**

Think this way:

* Numbers in the set can be formed by **some multiplications first, then additions**.
* Suppose we multiply k times:
* val = 1 \* a^k
* After that, we can add b any number of times:
* val + m\*b
* We want:
* n = a^k + m\*b

for some integers k ≥ 0, m ≥ 0.

**🔹 Approach**

* Try all possible powers of a:
* cur = 1, a, a^2, a^3, ...

while cur ≤ n.

* For each cur, check if:
* (n - cur) % b == 0
* If yes, then n is reachable.

Since a > 1, powers grow exponentially, so the loop is **O(log\_a(n)) ≈ O(log n)**.  
This is very efficient for n ≤ 10^9.

**🔹 Complexity**

* Each test case: **O(log n)**
* With t ≤ 10^5 test cases: about 10^5 \* log(10^9) ≈ 3 \* 10^6 operations → totally fine.

**🔹 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

long long n, a, b;

cin >> n >> a >> b;

bool ok = false;

if (a == 1) {

// Only additions matter

if ((n - 1) % b == 0) ok = true;

} else {

long long cur = 1;

while (cur <= n) {

if ((n - cur) % b == 0) {

ok = true;

break;

}

if (cur > n / a) break; // avoid overflow

cur \*= a;

}

}

cout << (ok ? "Yes\n" : "No\n");

}

return 0;

}

**🔹 Example Walkthrough**

Input:

5

24 3 5

10 3 6

2345 1 4

19260817 394 485

19260817 233 264

1. (24,3,5)
   * Powers: 1,3,9,27…
   * For cur=9: (24-9)=15, divisible by 5 ✅ → YES
2. (10,3,6)
   * Powers: 1,3,9
   * Check (10-1)=9 % 6 ≠0, (10-3)=7 %6≠0, (10-9)=1%6≠0 → NO
3. (2345,1,4)
   * Special case: (2345-1)%4==0 ✅ → YES

… matches sample output.

✅ Key takeaway:  
The trick is realizing the set is { a^k + m\*b }, then brute-forcing a^k up to n.

Another Approach :

#include<bits/stdc++.h>

using namespace std;

#define ll long long

int main()

{

int t;cin>>t;

while(t--)

{

ll n,a,b;

cin>>n>>a>>b;

if(n==1 || b==1 || n%b==1)

cout<<"Yes"<<endl;

else if(a==1)

{

if(n%b==1)

cout<<"Yes"<<endl;

else

cout<<"No"<<endl;

}

else{

int ans=0, i=0;

ll x=a;

while(pow(a,i)<=n)

{

ll z=n - pow(a,i);

if(z%b==0)

{

ans=1;

break;

}

i++;

}

if(ans==1)

cout<<"Yes"<<endl;

else

cout<<"No"<<endl;

}

}

}

**B. Customising the Track -** [**https://codeforces.com/contest/1543/problem/B**](https://codeforces.com/contest/1543/problem/B)

**📌 Problem Restatement**

We have:

* A track divided into n sub-tracks.
* Each sub-track i has aia\_i traffic cars.
* The **inconvenience** is defined as:

inconvenience=∑i=1n∑j=i+1n∣ai−aj∣\text{inconvenience} = \sum\_{i=1}^n \sum\_{j=i+1}^n |a\_i - a\_j|

* We are allowed to **move cars between sub-tracks** any number of times.

We want to **minimize inconvenience** after optimal moves.

**🔎 Key Observations**

1. **If all aia\_i are equal**, inconvenience = 0, because ∣ai−aj∣=0|a\_i - a\_j| = 0.
2. We can **redistribute cars freely**, so the optimal distribution is:
   * Some sub-tracks having ⌊Sn⌋\lfloor \frac{S}{n} \rfloor cars,
   * Others having ⌈Sn⌉\lceil \frac{S}{n} \rceil cars,  
     where S=∑aiS = \sum a\_i (total cars).

👉 This ensures counts differ by **at most 1**, minimizing absolute differences.

1. Let:
   * q=⌊S/n⌋q = \lfloor S / n \rfloor
   * r=S mod nr = S \bmod n (number of sub-tracks that get one extra car).

Then:

* + rr sub-tracks have q+1q+1 cars,
  + n−rn-r sub-tracks have qq cars.

**📊 Computing Minimum Inconvenience**

In this distribution:

* Pairs of equal counts contribute **0**.
* Only pairs of (q,q+1)(q, q+1) contribute **1**.

Number of such pairs =

r×(n−r)r \times (n-r)

So:

Minimum Inconvenience=r⋅(n−r)\text{Minimum Inconvenience} = r \cdot (n-r)

**✅ Example Walkthrough**

**Example 1**

Input:

3

1 2 3

* S=6,n=3,q=2,r=0 S = 6, n = 3, q = 2, r = 0.
* All sub-tracks can have 2 cars.
* Inconvenience = 0×(3−0)=00 \times (3-0) = 0.

**Example 2**

Input:

0 1 1 0

* S=2,n=4,q=0,r=2 S = 2, n = 4, q = 0, r = 2.
* Distribution: 2 sub-tracks with 1, 2 sub-tracks with 0.
* Inconvenience = 2×(4−2)=42 \times (4-2) = 4.

**Example 3**

Input:

8 3 6 11 5 2 1 7 10 4

* S=57,n=10,q=5,r=7 S = 57, n = 10, q = 5, r = 7.
* 7 sub-tracks get 6 cars, 3 sub-tracks get 5 cars.
* Inconvenience = 7×3=217 \times 3 = 21.

Matches sample output ✔️.

**💻 C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

long long n;

cin >> n;

vector<long long> a(n);

long long sum = 0;

for (int i = 0; i < n; i++) {

cin >> a[i];

sum += a[i];

}

long long r = sum % n;

cout << r \* (n - r) << "\n;

}

return 0;

}

**⏱️ Complexity Analysis**

* Reading input: O(n)O(n) per test.
* Computing sum: O(n)O(n).
* Answer formula: O(1)O(1).
* Total: O(∑n)O(\sum n), bounded by 2⋅1052 \cdot 10^5. ✅ Efficient.

**Another Approach** :

#include<bits/stdc++.h>

using  namespace  std;

#define ll long long

int main()

{

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    ll t;cin>>t;

    while(t--)

    {

        ll n;cin>>n;

        ll sum=0,x;

        for(int i=0;i<n;i++)

        {

            cin>>x;

            sum+=x;

        }

        ll z=sum%n;

        ll ans=(n-z)\*z;

        cout<<ans<<endl;

    }

}

**B. Alphabetical Strings -** [**https://codeforces.com/contest/1547/problem/B**](https://codeforces.com/contest/1547/problem/B)

Approach :

#include<bits/stdc++.h>

using namespace std;

#define ll long long

int main()

{

ll t;cin>>t;

while(t--)

{

string s;cin>>s;

bool ans=true;

int k=-1;

for(int i=0;i<=s.size();i++)

{

if(s[i]=='a')

{

k=i;

break;

}

}

if(k==-1)

{

cout<<"NO"<<endl;

}

else{

int i=k-1;

int j=k+1;

char ch='b';

while(i>=0 || j<s.size())

{

if(s[i]==ch)

{

ch++;

i--;

}

else if(s[j]==ch)

{

ch++;

j++;

}

else{

ans=false;

break;

}

}

if(!ans)

cout<<"NO"<<endl;

else

cout<<"YES"<<endl;

}

}

}

**Another Approach :**

**🔹 Problem Restatement**

You’re given a string s (length 1 ≤ n ≤ 26), and you need to check if it can be built using the following rules:

* Start with an empty string.
* Iteratively take the **next letter of the alphabet** (a, b, c, …, in order).
* Place it either **at the left** or **at the right** of the current string.

Such strings are called **alphabetical strings**.

👉 Example:

* "bac" ✅
  + Step 1: "a"
  + Step 2: put "b" to the left → "ba"
  + Step 3: put "c" to the right → "bac".
* "acb" ❌
  + You’d need "c" before "b", but "c" comes later in the alphabet sequence.

**🔹 Key Observations (How to Think)**

1. The string must **contain all letters from 'a' to some letter consecutively**.
   * Example: "bac" → has a, b, c.
   * "ihfcbadeg" → has a, b, c, d, e, f, g, h, i.
   * "xyz" ❌ → letters are not consecutive from a.
2. Construction rule:
   * Start with the **highest letter** in the string (say 'c' in "bac").
   * From there, expand left and right, checking if you can place 'b', then 'a', etc.
3. Algorithmically:
   * Find the position of 'a'.
   * Then try to expand left/right while looking for the next letters ('b', 'c', …).
   * If at any step you **can’t find the required letter at the current boundary**, it’s **NO**.

**🔹 Solution Approach**

1. For each string:
   * Check if it contains all characters from 'a' to 'a'+(n-1).
   * If not, print NO.
2. Find position of 'a' in the string.
3. Set **two pointers**:
   * l = pos(a), r = pos(a)
   * Try to extend towards left or right for 'b', 'c', … up to the max letter in string.
4. If possible → YES, otherwise → NO.

**🔹 C++ Solution**

#include <bits/stdc++.h>

using namespace std;

bool isAlphabetical(string s) {

int n = s.size();

int pos = s.find('a'); // position of 'a'

if (pos == string::npos) return false;

int l = pos, r = pos;

char nextChar = 'b';

// expand until we've placed all n letters

while (nextChar < 'a' + n) {

if (l > 0 && s[l - 1] == nextChar) {

l--;

} else if (r + 1 < n && s[r + 1] == nextChar) {

r++;

} else {

return false;

}

nextChar++;

}

return true;

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int t;

cin >> t;

while (t--) {

string s;

cin >> s;

cout << (isAlphabetical(s) ? "YES" : "NO") << "\n";

}

return 0;

}

**🔹 Complexity Analysis**

* For each test case of length n:
  + find('a') → O(n).
  + Expanding for at most n steps → O(n).
* So, **time complexity** = O(n) per string, O(t \* n) overall.
* Since n ≤ 26, even with t = 10^4, this is super fast.
* **Space complexity** = O(1) (just pointers and chars).