Closest To The Century:

Approach:

1. **Input:**
   - Read the number of test cases `test`.
   - For each test case:
     - Read the number of players `n`.
     - Read the runs scored by each player into the array `arr`.
2. **Sort the Array:**
   - Sort the array `arr` in non-decreasing order.
3. **Output:**
   - Print the last element of the sorted array, which represents the closest run to the century.

```
#include<bits/stdc++.h>

using namespace std;

int main() {

    /* Enter your code here. Read input from STDIN. Print output to
STDOUT */

    int test;

    cin>>test;

    while(test--){

        int n;

        cin>>n;

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

            cin>>arr[i];

        }

        sort(arr,arr+n);

        cout<<arr[n-1]<<endl;

    }

    return 0;

}
```

Chocolate Paradox

To determine if Alisa and Meena can exactly consume T chocolates given that Alisa eats X chocolates at a time and Meena eats Y chocolates at a time, we need to check if there exist non-negative integers i and j such that:

$$X \cdot i + Y \cdot j = T$$

where:

- X is the number of chocolates Alisa eats at once.
- Y is the number of chocolates Meena eats at once.
- T is the total number of chocolates they need to consume.

Approach

1. **Understand the Equation:**
    - We need to find integers i and j such that the linear combination of X and Y equals T.

2. **Brute Force Solution:**

   - Iterate over possible values of i from 0 to T/X (because if i is larger, $X \cdot i$ will exceed T).
   - For each i, check if there exists a j such that the remaining chocolates $T - X \cdot i$ is divisible by Y.

3. **Conditions for Validity:**

   - If $T - X \cdot i$ is non-negative and divisible by Y, then $j = (T - X \cdot i)/Y$ is a valid integer.
   - If we find such a pair (i, j), print "YES".
   - If no such pair is found after all iterations, print "NO".

Pseudocode

```
1. Read $X$, $Y$, and $T$.
2. Initialize a flag to check if a solution is found.
3. Loop over possible values of $i$ from 0 to $T/X$:
    • Compute the remaining chocolates after Alisa eats $i$ times: $remainder = T - X \cdot i$.
    • Check if $remainder$ is non-negative and divisible by $Y$:
        • If true, set the flag and break the loop.
4. If the flag is set, print "YES".
5. Otherwise, print "NO".
```

Implementation

Here's the C code that follows this approach:

```
#include <stdio.h>

int main() {
    int X, Y, T;
    scanf("%d %d %d", &X, &Y, &T);

    int found = 0;

    // Loop through possible counts of Alisa's chocolates
    for (int i = 0; i <= T / X; i++) {
        int remainder = T - X * i;
        if (remainder >= 0 && remainder % Y == 0) {
            found = 1;
            break;
        }
    }

    if (found) {
        printf("YES\n");
    } else {
        printf("NO\n");
    }

    return 0;
}
```

Lav Loss

To determine the price at which Mr. Rahim should sell a TV to achieve a desired profit percentage, we need to first understand the relationship between the selling price, loss percentage, and profit percentage.

Steps to Solution

1. **Understanding the Problem:**
   - Mr. Rahim sold a TV at a price of X and incurred a loss of $Y\%$.
   - We need to determine the new selling price to achieve a profit of $Z\%$.
2. **Calculate the Cost Price:**
   - The loss percentage $Y\%$ tells us that the selling price X is less than the cost price.
   - Let the cost price be C.
   - The relationship between selling price and cost price with loss can be described as:

$$C = \frac{X \times 100}{100 - Y}$$

- This formula rearranges the standard loss formula $(C - X)/C \times 100 = Y$ to solve for C.

3. **Calculate the Desired Selling Price for Profit:**
   - To achieve a profit of $Z\%$, we need to set a new selling price S.
   - The relationship between cost price and selling price with profit can be described as:

$$S = C \times \left(1 + \frac{Z}{100}\right)$$

   - This formula comes from the profit formula $(S - C)/C \times 100 = Z$.

4. **Combine the Formulas:**
   - First, calculate C using the given selling price X and loss percentage Y.
   - Then, calculate S using the cost price C and the desired profit percentage Z.

5. **Output the Result:**
   - Print the new selling price S rounded to two decimal places.

Sample Calculation

Given:

- $X = 80$
- $Y = 20\%$
- $Z = 30\%$

Step-by-Step Calculation:

1. Calculate cost price C:

$$C = \frac{80 \times 100}{100 - 20} = \frac{8000}{80} = 100$$

2. Calculate the new selling price S for 30% profit:

$$S = 100 \times \left(1 + \frac{30}{100}\right) = 100 \times 1.30 = 130.00$$

So, Mr. Rahim should sell the TV for 130.00 tk to achieve a 30% profit.

Implementation

```c
#include <stdio.h>

int main() {
    int X, Y, Z;
    scanf("%d %d %d", &X, &Y, &Z);

    // Calculate the cost price
    double cost_price = (double)(100 * X) / (100 - Y);

    // Calculate the new selling price for Z% profit
    double selling_price = cost_price * (1 + (double)Z / 100);

    // Print the result rounded to two decimal places
    printf("%.2lf\n", selling_price);

    return 0;
}
```

Solution Explanation

1. **Reading Input:**
   - Read the number of test cases T.
   - For each test case, read the integer N which denotes the size of the array.
   - Read the array A of size N.
2. **Removing Duplicates:**
   - Insert the elements of A into a \``set`\`. This automatically removes duplicates since sets do not allow duplicate values.
3. **Sorting:**
   - Convert the set back to a vector and sort it.
4. **Output:**
   - Print the sorted vector for each test case.

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

int main() {
    int t;
    scanf("%d", &t);

    while (t--) {
        int n;
        scanf("%d", &n);
        int a[n];
        for (int i = 0; i < n; i++) {
            scanf("%d", &a[i]);
        }

        // Step 1: Sort the array (Bubble Sort)
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
```

```c
                    int temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Step 2: Remove duplicates
        int b[n];
        b[0] = a[0];
        int j = 1;
        for (int i = 1; i < n; i++) {
            if (a[i] != b[j - 1])
                b[j++] = a[i];
        }

        // Step 3: Print the result
        for (int i = 0; i < j; i++) {
            printf("%d ", b[i]);
        }
        printf("\n");
    }

    return 0;
}
```

Palindro-Cost

1. **Understanding the Problem:**
    - A palindrome reads the same forwards and backwards.
    - To transform a string into a palindrome, we need to ensure that the characters at mirrored positions from the start and end are the same.
    - The cost to change one character to another is given by the absolute difference in their positions in the alphabet.
2. **Approach:**
    - For each string, compare characters from the start and the end moving towards the center.
    - Calculate the cost to make mirrored characters the same.
    - Sum these costs and check if the total cost is within the given budget K.
3. **Algorithm:**
    - Loop through the given queries.
    - For each query, initialize `` `i` `` to the start and `` `j` `` to the end of the string.
    - Iterate while `` `i` `` is less than `` `j` ``:
        - If characters at positions `` `i` `` and `` `j` `` are different, calculate the cost to make them the same.
        - Add this cost to a running total.
    - After processing the string, check if the total cost is within the given budget K.
    - Print "YES" if it is, otherwise print "NO".
4. **Efficiency Considerations:**
    - The algorithm processes each string in linear time relative to its length, $O(N)$.
    - Given the constraints, this approach will efficiently handle the upper limits of input size.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int t;
    scanf("%d", &t);

    while (t--) {
        char a[100001];
        int cost;
        scanf("%s", a);
        scanf("%d", &cost);

        int n = strlen(a);
        int i = 0, j = n - 1;
        int total_cost = 0;

        while (i < j) {
            if (a[i] != a[j]) {
                total_cost += abs(a[i] - a[j]);
            }
            i++;
            j--;
        }

        if (total_cost <= cost) {
            printf("YES\n");
        } else {
            printf("NO\n");
        }
    }

    return 0;
}
```

1. **Reading Input:**
    - We first read the number of test cases `t`.
    - For each test case, we read the string `a` and the integer `cost`.
2. **Processing Each String:**
    - Calculate the length of the string `n`.
    - Initialize pointers `i` and `j` to the start and end of the string.
    - Initialize `total_cost` to keep track of the accumulated cost.
3. **Comparing Characters:**
    - Loop through the string from both ends towards the center.
    - If the characters at `i` and `j` are different, compute the cost to change one to the other and add it to `total_cost`.
    - Continue until all pairs of characters have been compared.
4. **Output Result:**
    - After calculating the total cost for the current string, compare it to `cost`.
    - Print "YES" if the total cost is within the budget, otherwise print "NO".

Oddness of a String

Approach

Given the constraints, a direct approach of calculating the frequency of each character for each query would be too slow. Instead, we can use **prefix frequency arrays** to efficiently compute the oddness for any substring in constant time.

Steps to Solve the Problem

1. **Precompute Prefix Frequencies:**
   - Construct a 2D prefix frequency array `` `v` `` where `` `v[i][j]` `` represents the number of times the character `` `j` `` ('a' + j) appears in the substring `` `s[0, i-1]` ``.
   - For each character in the string, update the prefix frequency array.
2. **Query Handling:**
   - For each query, compute the frequency of each character in the substring `` `s[l, r]` `` using the prefix frequency array.
   - Determine the oddness by counting the characters that appear an odd number of times.

Detailed Algorithm

1. **Initialization:**
   - Create a 2D array `` `v` `` with dimensions `` `(n+1) x 26` `` (since there are 26 lowercase English letters), initialized to 0.
2. **Building Prefix Frequency Array:**
   - For each character `` `s[i-1]` `` in the string `` `s` ``, update `` `v[i]` `` based on `` `v[i-1]` `` and the character at position `` `i-1` ``.
3. **Processing Queries:**
   - For each query `` `(l, r)` ``, determine the frequency of each character in the substring `` `s[l-1, r-1]` `` using the difference between `` `v[r]` `` and `` `v[l-1]` ``.
   - Count the number of characters with odd frequencies to determine the oddness.

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

```
#define fastread() (ios_base::sync_with_stdio(0),cin.tie(0));

void solve() {
    int n, q;
    cin >> n >> q;
    string s;
    cin >> s;
    vector<vector<int>> v(n + 1, vector<int>(26, 0));

    // Build the prefix frequency array
    for (int i = 1; i <= n; i++) {
        int charIndex = s[i - 1] - 'a';
        for (int j = 0; j < 26; j++) {
            v[i][j] = v[i - 1][j];
        }
        v[i][charIndex]++;
    }

    while (q--) {
        int l, r;
        cin >> l >> r;
        int oddCount = 0;

        // Calculate the oddness for the substring s[l-1, r-1]
        for (int i = 0; i < 26; i++) {
            int frequency = v[r][i] - v[l - 1][i];
            if (frequency % 2 != 0) {
                oddCount++;
            }
        }

        cout << oddCount << "\n";
    }
}

int main() {
    fastread();
    int t;
    cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}
```

Builder’s Dream

Solution Outline

To solve this problem, we need to find the maximum possible minimum height of the buildings after spending the given money. This suggests a binary search approach on the possible heights of the lowest building.

Steps

1. **Binary Search Setup:**
   - Initialize the binary search with `` `l` `` (left) set to the current minimum height of the buildings and `` `r` `` (right) set to a sufficiently large number (in this case, 10^{12}).
2. **Binary Search Execution:**
   - For each midpoint `` `mid` `` in the binary search, calculate the total cost required to increase all buildings below `` `mid` `` to reach `` `mid` ``.
   - If the total cost is within `` `k` ``, it means `` `mid` `` is a feasible height for the lowest building. Adjust the left boundary (`` `l` ``) to `` `mid + 1` `` to search for a potentially higher feasible height.
   - If the total cost exceeds `` `k` ``, adjust the right boundary (`` `r` ``) to `` `mid - 1` `` to search for a lower feasible height.
3. **Termination:**
   - The binary search terminates when `` `l` `` exceeds `` `r` ``, at which point `` `r` `` holds the maximum feasible minimum height.

```
#include<bits/stdc++.h>
#define int long long
```

```
#define fastread() ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
using namespace std;

void solve() {
    int n, k;
    cin >> n >> k;
    vector<int> h(n);
    for (int i = 0; i < n; i++) {
        cin >> h[i];
    }

    // Binary search for the maximum minimum height
    int l = *min_element(h.begin(), h.end()), r = 1e12, ans = l;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        int total = 0;
        for (int i = 0; i < n; i++) {
            if (h[i] < mid) {
                total += (mid - h[i]);
            }
        }
        if (total <= k) {
            ans = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }

    cout << ans << "\n";
}

int32_t main() {
    fastread();
    int t = 1;
    while (t--) {
        solve();
    }
    return 0;
}
```

Easy Multiplication!!!

Solution Approach

Given the constraints, it is infeasible to calculate the product directly for each query due to potential integer overflow and inefficiency. Instead, we leverage the following insights:

1. The product of a subarray will be zero if there is at least one zero in the subarray.
2. If there are no zeros in the subarray, the product's sign will depend on the number of negative numbers:
   - Even count of negative numbers results in a positive product.
   - Odd count of negative numbers results in a negative product.

To efficiently answer each query, we use precomputed data:

1. **Prefix Sum Arrays:**
   - `` `prefix_zero[i]` ``: Number of zeros from the start of the array up to the i-th index.
   - `` `prefix_negative[i]` ``: Number of negative numbers from the start of the array up to the i-th index.

Using these prefix arrays, we can determine the number of zeros and negative numbers in any subarray in constant time.

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

void solve() {
    int n;
    cin >> n;
    vector<int> arr(n + 1);
    vector<int> prefix_zero(n + 1, 0);
    vector<int> prefix_negative(n + 1, 0);

    for (int i = 1; i <= n; ++i) {
        cin >> arr[i];
        prefix_zero[i] = prefix_zero[i - 1] + (arr[i] == 0 ? 1 : 0);
        prefix_negative[i] = prefix_negative[i - 1] + (arr[i] < 0 ? 1 :
0);
    }

    int q;
    cin >> q;
    while (q--) {
        int l, r;
        cin >> l >> r;

        int zero_count = prefix_zero[r] - prefix_zero[l - 1];
        int negative_count = prefix_negative[r] - prefix_negative[l - 1];

        if (zero_count > 0) {
            cout << "NO\n";
        } else if (negative_count % 2 == 0) {
            cout << "YES\n";
        } else {
            cout << "NO\n";
        }
    }
}

int32_t main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    solve();
    return 0;
}
```

Make Everything Equal

Solution Approach

To determine if it's possible to make all elements of the array equal with at most `k` operations, we need to follow these steps:

1. **Count Frequencies:**
   - For each test case, count the frequency of each element in the array.
2. **Find the Most Frequent Element:**
   - Determine the maximum frequency (`max_freq`) of any element in the array.
3. **Calculate Required Changes:**
   - The number of changes required to make all elements equal to the most frequent element is `n - max_freq`.
4. **Check Feasibility:**
   - If `n - max_freq <= k`, print "YES".
   - Otherwise, print "NO".

This approach ensures that we use the minimal number of operations by converting all elements to the most frequently occurring element.

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int t;
    cin >> t;
    while (t--) {
        int n, k;
        cin >> n >> k;

        vector<int> arr(n);
        unordered_map<int, int> freq;
        for (int i = 0; i < n; ++i) {
            cin >> arr[i];
            freq[arr[i]]++;
        }

        int max_freq = 0;
        for (const auto& entry : freq) {
            max_freq = max(max_freq, entry.second);
        }

        int required_changes = n - max_freq;
        if (required_changes <= k) {
            cout << "YES\n";
        } else {
            cout << "NO\n";
        }
    }

    return 0;
}
```