🧩 Problem Understanding

You're given an array A.
In **one step**, you can:

- Select the **largest element** in the array
- Replace it with the **second largest** element.

🎯 Goal:

Make **all elements equal** with the **minimum number of steps**.

🧠 Key Insight (Observation):

All elements will eventually become equal to the **smallest element** of the array.

Why?

Because you can **only reduce** values to a lower value (no increasing), and the only way elements get updated is to copy smaller values.

⚡ Optimized Approach using map (O(N log N))

🎯 Idea:

- Use a **map<int, int> as it is sorted no unordered** to store **frequency of each number**.
- Since **map is sorted** in C++, we can always access the **largest and second largest** easily.
- In one operation:
  - Replace all elements with the **largest key** by the **second largest key**.
  - Decrease count of largest, add to second largest.
  - Repeat until only one key (value) remains in the map.

C++ code:

```
#include <iostream>

#include <map>

#include <vector>

using namespace std;


int minStepsToMakeAllEqual(vector<int>& A) {

  map<int, int> freq;
```

```
    for (int val : A) {
        freq[val]++;
    }

    int steps = 0;

    while (freq.size() > 1) {
        auto it_largest = prev(freq.end());      // Largest element
        auto it_second = prev(freq.end(), 2);    // Second largest

        int largest = it_largest->first;
        int count = it_largest->second;
        int second = it_second->first;  // second largest

        freq[second] += count;    // Convert all 'largest' to 'second'
        freq.erase(largest);      // Remove largest
        steps += count;           // Each conversion is a step
    }

    return steps;
}
int main(){
    vector<int> a={5,5,4,4,2};
    cout<<minStepsToMakeAllEqual(a);
}
```

☑ **Summary:**

Function	Use
prev(it)	Go one step back from it
prev(it, n)	Go n steps back from it
In map/set	Often used to get largest / second largest

Syntax	Meaning	When to Use
.	Access member of an object	When you have the object
->	Access member through pointer or iterator	When you have a pointer/iterator

In C++, std::prev is a utility function from the <iterator> header. It is used to get an iterator pointing to the previous element from a given iterator.

Max Distance Between Two Occurrences

C++CODE:

```
int maxDistance(vector<int> &arr) {
  int ans = 0;  // To store the final answer (maximum distance found)
  unordered_map<int, int> mp;  // To store the first occurrence index of each element

  for (int i = 0; i < arr.size(); i++) {
    if (mp.find(arr[i]) == mp.end()) {
      // If the element is seen for the first time, store its index
      mp[arr[i]] = i;
    } else {
      // If the element was seen before, calculate the distance
      // and update the maximum distance if it's larger
      ans = max(ans, i - mp[arr[i]]);
    }
  }

  return ans;  // Return the maximum distance between two equal elements
}
```

📝 **Summary Notes:**

- The map stores the first index where each element appears.
- On every repeated occurrence, we compute the distance to the first, and keep the maximum.
- This is a classic hash map-based linear solution for finding max distance between equal elements.

Time Complexity: O(N)
Space Complexity: O(N)

✅ **Java Code with Comments:**

```
public int maxDistance(int[] arr) {
  int ans = 0;  // Variable to store the maximum distance found so far


  // HashMap to store the first occurrence index of each element
  HashMap<Integer, Integer> mp = new HashMap<>();


  // Iterate through the array
  for (int i = 0; i < arr.length; i++) {
    // If the element is seen for the first time, store its index
    if (!mp.containsKey(arr[i])) {
      mp.put(arr[i], i);
    } else {
      // If the element is already seen, calculate the distance between
      // current index and the first occurrence, and update the max distance
      ans = Math.max(ans, i - mp.get(arr[i]));
    }
  }


  // Return the maximum distance found
  return ans;
}
```

First Unique Character in a String

🔍 C++ Code with Comments:

```
int firstUniqChar(string s) {
  int n = s.length();  // Get the length of the string
  unordered_map<char, int> mp;  // Hash map to store character frequencies

  // Step 1: Count frequency of each character in the string
  for (int i = 0; i < n; i++) {
    mp[s[i]]++;  // Increment frequency count for character s[i]
  }

  // Step 2: Find the first character with frequency == 1
  for (int i = 0; i < n; i++) {
    if (mp[s[i]] == 1) {  // If character occurs only once
      return i;       // Return its index
    }
  }

  // If no unique character is found, return -1
  return -1;
}
```

🧠 Why this works efficiently:

- The first loop counts character frequencies → O(N)
- The second loop finds the first unique character → O(N)
- unordered_map<char, int> gives O(1) average access time for both insert and lookup.

📊 Time and Space Complexity:

Operation	Complexity
Time	O(N)
Space (unordered_map)	O(1)*

*Because there are only 26 lowercase letters (as per typical constraints), space is constant.

📝 Notes:

- Use unordered_map for fast insert and lookup.
- This approach preserves the original order of the string while checking frequency.
- Returns the first unique character's index, not the character itself.

1002. Find Common Characters

🧾 Problem Statement:

Given an array of strings words, return all characters that appear in every string, including duplicates.

- The result can be in any order.
- Characters are all lowercase English letters.

🧠 Key Observations:

- We need to return only characters that are common across all strings.
- If a character appears k times in each word, it should appear k times in the result.
- Duplicates matter! For example, "l" appears twice in "bella" and "label" → include both.

✅ Optimal Approach (Using Arrays):

Since we only deal with lowercase letters 'a' to 'z', we can use fixed-size arrays (size 26) to store frequencies efficiently.

🔧 Steps:

1. Initialize a minFreq[26] array with INT_MAX.
   This will store the minimum frequency of each character across all words.
2. For each word in words:
   - Create a local freq[26] array to count character frequencies in that word.
   - Update minFreq[i] = min(minFreq[i], freq[i]) for all 26 characters.
3. After processing all words:
   - Loop over minFreq. For each character with non-zero frequency:
     - Add that character minFreq[i] times to the result.

Using Unordered map:

```
vector<string> commonChars(vector<string>& words) {
  unordered_map<char, int> mp;     // Store min frequency of each char
  vector<string> ans;

  // Step 1: Count frequency of characters in the first word
  for (char ch : words[0]) {
    mp[ch]++;
  }

  // Step 2: Intersect with each subsequent word
  for (int i = 1; i < words.size(); i++) {
    unordered_map<char, int> currmp;
    for (char ch : words[i]) {
      currmp[ch]++;
    }

    // Update global map with minimum frequency
    for (auto& it : mp) {
      it.second = min(it.second, currmp[it.first]);
    }
    }

    // Step 3: Add each character to result according to its min frequency
    for (auto it : mp) {
        for (int i = 0; i < it.second; i++) {
            ans.push_back(string(1, it.first));  // Convert char to string
        }
    }

    return ans;
}
```

Metric	Complexity
🕒 Time (TC)	O(N × L)
💾 Space (SC)	O(1)

Why O(1) space?
We use only two maps with max 26 keys (lowercase letters), so space doesn't grow with input size.

Using array of 26 characters:

```
vector<string> commonChars(vector<string>& words) {
    vector<int> minFreq(26, INT_MAX);

    for (string word : words) {
        vector<int> freq(26, 0);
        for (char ch : word) {
            freq[ch - 'a']++;
        }
        for (int i = 0; i < 26; i++) {
            minFreq[i] = min(minFreq[i], freq[i]);
        }
    }
```

```
  vector<string> result;
  for (int i = 0; i < 26; i++) {
    while (minFreq[i] --   >  0) {
      result.push_back(string(1, i + 'a'));
    }
  }

  return result;
}
```

🚀 Time & Space Complexity:

Complexity	Value
Time	O(N * L + 26) → ≈ O(N * L)
Space	O(26)

- N = number of words
- L = average word length

IMP POINTS:

✅ 1. By Reference in C++

Always use auto& in range-based loops to avoid copying elements when you want to modify the original container.

🔹 Example:

```
for (auto& it : mp) it.second = min(it.second, curr[it.first]);
```

✅ 2. Char to String in One Line

Use string(1, ch) to convert a single character ch to a string containing one character.

🔹 Example:

```
string s = string(1, 'a');  // s = "a"
string s1 = string(5, 'x');  // "xxxxx"
```