**Binary Search (in simple words):**  
Binary search is a way to quickly find something in a sorted list. Instead of checking one by one, you always look at the middle element. If the answer is not there, you throw away half the search space and continue with the other half. This makes the search very fast, only taking about log(n) steps instead of n.

**The Question:**  
You are given versions from 1 to n. At some point, the versions become “bad,” and once a version is bad, all the later versions are also bad. You have a function isBadVersion(version) to check if a version is bad. The goal is to find the **first bad version** using the fewest calls to isBadVersion.

**Your Solution:**

class Solution {

public:

int firstBadVersion(int n) {

int num = n/2;

while(num <= n){

if(isBadVersion(num)){

int temp = num - 1;

if(!isBadVersion(temp)){

return num; // found the first bad

}

num--; // move backward

}

else{

num++; // move forward

}

}

return 0;

}

};

* ✅ This works because you check neighbors to find the first bad version.
* ❌ But it is **slow (O(n))**, because you only move one step forward or backward at a time. If n is very large, this takes too long.

**Efficient Solution (Binary Search):**

class Solution {

public:

int firstBadVersion(int n) {

int left = 1, right = n;

while (left < right) {

int mid = left + (right - left) / 2; // middle point

if (isBadVersion(mid)) {

right = mid; // first bad is here or before

} else {

left = mid + 1; // first bad is after mid

}

}

return left; // or right, both meet at the first bad

}

};

* ✅ This is **fast (O(log n))**, because each step throws away half of the versions.
* For example, if n = 1,000,000,000, it only needs about **30 steps** instead of checking all billion versions.

**Summary:**

* Your solution checked **step by step**, so it was slower.
* Binary search **cuts the search in half every time**, so it’s much faster.
* Both find the answer, but the binary search version is efficient and works well for very large n.

**Step 1: Problem explanation**

**Problem:**  
You are given two strings: ransomNote and magazine.

* You want to see if you can **construct the ransom note** using letters from the magazine.
* **Rules:**
  1. You can only use each letter in magazine **once**.
  2. You can rearrange letters, but you cannot invent new letters.

**Example:**

ransomNote = "aa"

magazine = "aab"

* ransomNote needs **2 'a' letters**.
* magazine has **2 'a' letters** and 1 'b' → ✅ possible → return true

Another example:

ransomNote = "aa"

magazine = "ab"

* ransomNote needs 2 'a', magazine has only 1 → ❌ impossible → return false

**Step 2: How your frequency array solution works**

**Code**

bool canConstruct(string ransomNote, string magazine) {

int freq[26] = {0};

for(char c : magazine) freq[c - 'a']++;

for(char c : ransomNote) {

if(--freq[c - 'a'] < 0) return false;

}

return true;

}

**Explanation:**

1. int freq[26] = {0};
   * Create an array to **count each letter** in the magazine.
   * Index 0 → 'a', index 1 → 'b', … index 25 → 'z'.
2. for(char c : magazine) freq[c - 'a']++;
   * Count how many times each letter appears in the magazine.

**Example:**

magazine = "aab"

freq array after counting:

freq[0] = 2 // 'a'

freq[1] = 1 // 'b'

freq[2..25] = 0

1. for(char c : ransomNote) { if(--freq[c - 'a'] < 0) return false; }
   * For each letter in ransomNote, **use it once** by decrementing the count in freq.
   * If the count goes below 0, that means magazine doesn’t have enough of that letter → return false.
2. return true;
   * If all letters are successfully used without going negative → ransom note can be constructed → return true.

**Step 3: Example Walkthrough**

ransomNote = "aa"

magazine = "aab"

**Step 3a: Count letters in magazine**

| **Letter** | **Count** |
| --- | --- |
| a | 2 |
| b | 1 |
| c-z | 0 |

**Step 3b: Check letters in ransomNote**

* First 'a': decrement freq['a'] → 2 → 1 → OK
* Second 'a': decrement freq['a'] → 1 → 0 → OK

**All letters checked, no negatives → return true**

**Step 4: Brute Force Solution**

**Idea:** For each letter in ransomNote, search for it in magazine **and remove it** (like you were trying earlier).

bool canConstructBruteForce(string ransomNote, string magazine) {

for (char c : ransomNote) {

size\_t pos = magazine.find(c); // search for letter

if (pos == string::npos) return false; // not found

magazine.erase(pos, 1); // remove used letter

}

return true;

}

**Step 4a: Walkthrough of same example**

ransomNote = "aa"

magazine = "aab"

1. First 'a': find 'a' in magazine → pos = 0 → erase → magazine = "ab"
2. Second 'a': find 'a' in magazine → pos = 0 → erase → magazine = "b"  
   ✅ All letters used → return true

**Step 4b: Time Complexity**

* find + erase → O(n) each
* For ransomNote of size m → **O(m \* n)** → slower than frequency array

**Frequency array method is O(m+n), which is much faster for large strings.**