LEETCODE 242

Approach 1 – Sorting Method

```
bool isAnagram(string s, string t) {
    sort(s.begin(),s.end());
    sort(t.begin(),t.end());
    if(s==t){return true;}
    else return false;
}
```

Logic Flow

1. **Length check** – Tumne isme explicitly length check nahi kiya, lekin indirectly ho jata hai kyunki agar length alag hai to `sort` ke baad compare false de dega.
2. **Sorting** –
   - Dono strings ko sort kar diya.
   - Agar wo anagram hain, to sort hone ke baad dono ka order same ho jayega.
3. **Comparison** – Agar sorted strings same hain → return `true`, warna `false`.

Time Complexity – O(n log n) (sorting ki wajah se).

Space Complexity – O(1) (in-place sorting).

Approach 2 – Two Maps Method

```
bool isAnagram(string s,string t){
    if(s.length()!=t.length())return false;

    unordered_map<char,int> mp1; // for s
    unordered_map<char,int> mp2; // for t
```

```
    // Count frequency in s
    for(int i=0;i<s.length();i++){
        mp1[s[i]]++;
    }
    // Count frequency in t
    for(int i=0;i<t.length();i++){
        mp2[t[i]]++;
    }

    // Compare both maps
    for(auto pr : mp1){
        char ch = pr.first;
        char freq = pr.second;

        if(mp2.find(ch)!=mp2.end()){ // element present in t
            if(mp2[ch] != mp1[ch]){ // frequency mismatch
                return false;
            }
        }
        else return false; // character not found in t
    }

    return true;
}
```

Logic Flow

1. **Length check** – Agar length alag hai, return `false`.
2. **Frequency counting** –
   - `mp1` me `s` ke har character ka frequency count store kiya.
   - `mp2` me `t` ke har character ka frequency count store kiya.
3. **Comparison** –
   - `mp1` ke har entry ke liye check kiya ki `mp2` me wahi char same frequency ke saath hai ya nahi.

- Agar koi char missing ya frequency mismatch hui → return `false`.

4. Sab match hue to return `true`.

Time Complexity – O(n)

Space Complexity – O(n) (do alag hash maps).

Approach 3 – Single Map Method

```
bool isAnagram(string s,string t){
    if(s.length()!=t.length())return false;

    unordered_map<char,int> mp;

    // Count frequency from s
    for(int i=0;i<s.length();i++){
        mp[s[i]]++;
    }

    // Reduce frequency from t
    for(int i=0;i<t.length();i++){
        char ch = t[i];
        if(mp.find(ch)!=mp.end()){
            mp[ch]--;
            if(mp[ch]==0){
                mp.erase(ch); // remove if count 0
            }
        }
        else return false; // character not present
    }

    return true;
}
```

Logic Flow

1. **Length check** – Agar alag hai to `false`.
2. **Build frequency map** – `mp` me `s` ke har character ka count.
3. **Reduce counts** –
    - Har char of `t` ke liye `mp` me count minus kiya.
    - Agar count 0 ho gaya, to map se erase kiya (map clean rakhne ke liye).
    - Agar character nahi mila, to directly `false`.
4. End tak agar koi mismatch nahi mila → return `true`.

Time Complexity – O(n)

Space Complexity – O(n) (sirf ek map).

📌 Summary Table

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting	O(n log n)	O(1)	Simple code	Slower for large n
Two Maps	O(n)	O(n)	Clear logic, easy to debug	Extra map memory
Single Map	O(n)	O(n)	Less space than two maps	Slightly more careful implementation

---