Brute Force Approach:

Approach:

- The simplest way to check if two strings are anagrams is to sort both strings and then compare them.
- If both sorted strings are equal, then the strings are anagrams; otherwise, they are not.

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

    sort(s.begin(), s.end());
    sort(t.begin(), t.end());

    return s == t;
}
```

Complexity:

- **Time Complexity:** O(n log n), where n is the length of the strings. This is due to the sorting step.
- **Space Complexity:** O(1), as sorting is done in-place and only constant extra space is used.

Better Approach:

Approach:

- Use a frequency count of characters to determine if the two strings have the same characters in the same quantities.
- Create an array of size 26 (for each letter in the alphabet) and count the occurrences of each character in both strings.
- If the counts match for every character, the strings are anagrams.

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

    vector<int> count(26, 0);

    for (int i = 0; i < s.length(); i++) {
        count[s[i] - 'a']++;
        count[t[i] - 'a']--;
    }

    for (int i = 0; i < 26; i++) {
        if (count[i] != 0) return false;
    }

    return true;
}
```

Complexity:

- **Time Complexity:** O(n), where n is the length of the strings. We are iterating through the strings once.
- **Space Complexity:** O(1), as the extra space used is fixed at 26 for the alphabet array.

Best Approach:

Approach:

- The best approach is similar to the better approach, but instead of using an array of size 26, you could use a hash map to handle any character set (including Unicode characters).
- This method is more general and can handle cases where the characters are not limited to just lowercase English letters.

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

    unordered_map<char, int> count;

    for (char c : s) count[c]++;
    for (char c : t) {
        count[c]--;
        if (count[c] < 0) return false;
    }

    return true;
}
```

Complexity:

- **Time Complexity:** O(n), where n is the length of the strings. We are iterating through the strings once.
- **Space Complexity:** O(k), where k is the number of distinct characters in the string set.