- Given two strings, return true if they are anagrams, false otherwise

Thought:

- I am unsure if the two inputs are exactly the same if that counts as an anagram. I will consider that to NOT be an anagram.
- Additionally, if two input strings are of different size, then we already know they are NOT anagrams.

For the algorithm, there is a simple approach that helps adapt to any bag of characters.

1.) Sort both strings

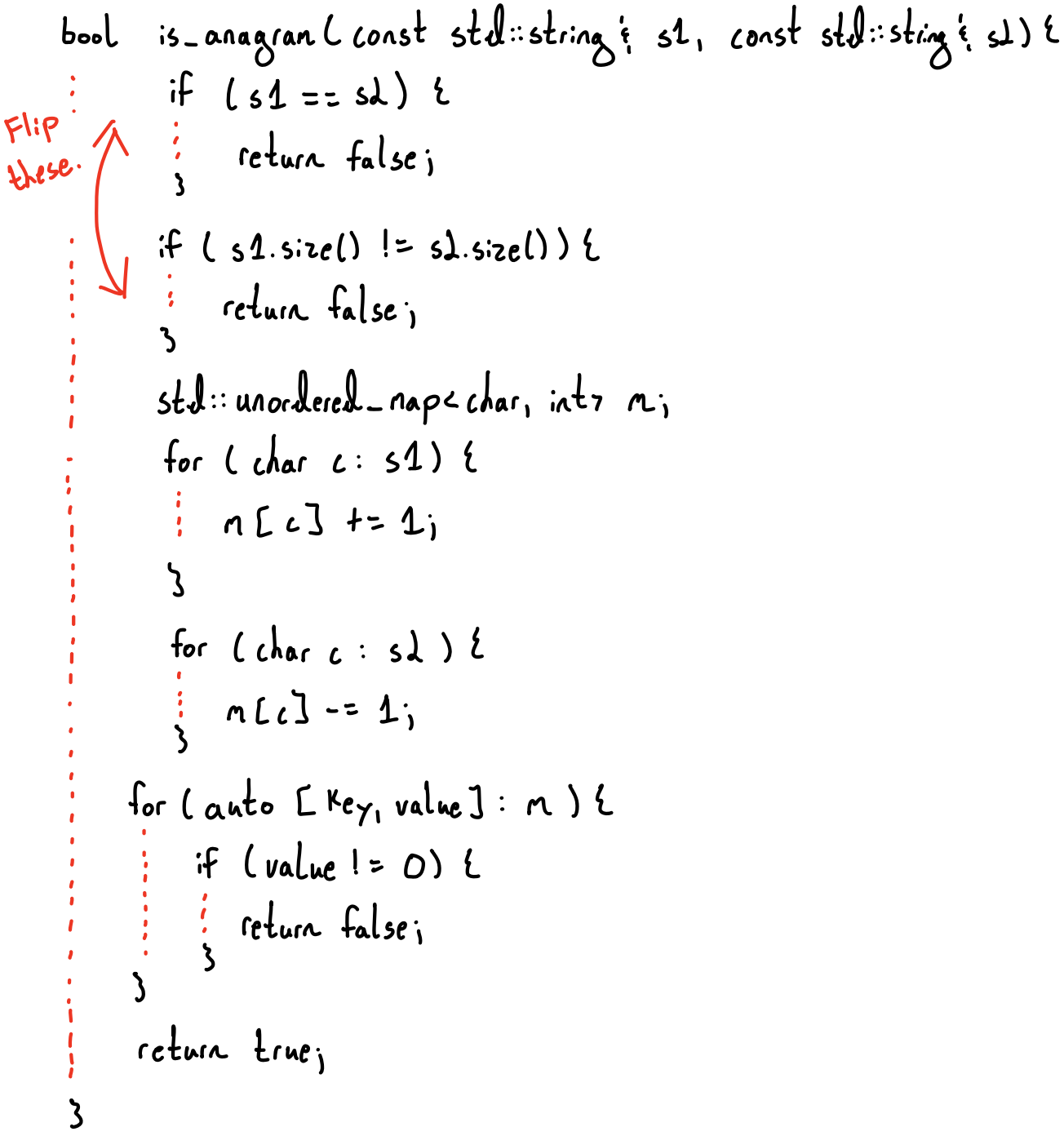
2.) Check for equality

Since O(n log n) for sorting, we could improve to O(n) by using a map. This maintains being able to use it for any bag of characters, not just a-z

1.) Create a map<char, int> counting the number of occurrences for each character in s1

2.) Enumerate over all characters in s2, decrementing the count for each character seen.

3.) Enumerate over all values placed in map. If all are zero, return true. Otherwise false.


Flip these.
bool is_anagram(const std::string& s1, const std::string& s2) {
if (s1 == s2) {
return false;
}
if (s1.size() != s2.size()) {
return false;
}
std::unordered_map<char, int> m;
for (char c: s1) {
m[c] += 1;
}
for (char c : s2) {
m[c] -= 1;
}
for (auto [key, value] : m) {
if (value != 0) {
return false;
}
}
return true;
}

* Probably more optimal for this set of characters, but can't be used for other bags.

Since characters are a-z and contiguous, we can leverage a contiguous array of size 26.

```
bool is_anagram(const std::string& s1, const std::string& s2) {
    if (s1.size() != s2.size()) {
        return false;
    }
    if (s1 == s2) {
        return false;
    }
    std::array<int, 26> a(0); // init w/ zero

    for (char c : s1) {
        a[static_cast<int>(c - 'a')] += 1;
    }
    for (char c : s2) {
        a[static_cast<int>(c - 'a')] -= 1;
    }
    return std::find_if_not(a.begin(), a.end(), 0) == a.end();
}
```

Same exact approach, using different data structure.