Approach Used to Solve the Problem

This problem is solved using the **Sliding Window** technique along with **Frequency Counting**.

- **Key Idea:** Instead of generating all permutations of `s1` and checking if they exist in `s2`, we count the frequency of characters in `s1` and use a **sliding window** of the same length in `s2` to check if any substring has the same frequency count.
- **Efficiency:** This approach runs in **O(n)** time complexity instead of **O(n! × m)** in a brute-force approach.

Step-by-Step Explanation

Step 1: Helper Function `checkEqual()`

```
bool checkEqual(int a[26], int b[26]) {
    for (int i = 0; i < 26; i++) {
        if (a[i] != b[i])
            return 0;
    }
    return 1;
}
```

- This function **compares two frequency arrays** (`a` and `b`).
- If they are equal (i.e., both arrays have the same character counts), return `true (1)`, else return `false (0)`.
- We need this function to check if a substring of `s2` is a permutation of `s1`.

Step 2: Character Frequency Array for `s1`

```
int count1[26] = {0};

for (int i = 0; i < s1.length(); i++) {
    int index = s1[i] - 'a';
    count1[index]++;
}
```

- We create an **array of size 26** (`count1[26]`) to **store the frequency of each character in `s1`**.
- Each index of the array corresponds to a letter (`'a' → index 0`, `'b' → index 1`, ..., `'z' → index 25`).
- **Example**: If `s1 = "ab"`, then:

  ```
  count1['a' - 'a'] = count1[0] = 1
  count1['b' - 'a'] = count1[1] = 1
  ```

  So, `count1 = [1, 1, 0, 0, 0, ..., 0]`

Step 3: Initializing the Sliding Window

```
int i = 0;
int windowSize = s1.length();
int count2[26] = {0};

while (i < windowSize && i < s2.length()) {
    int index = s2[i] - 'a';
    count2[index]++;
    i++;
}
```

- We **initialize a second frequency array (`count2`)** for tracking the character count **inside a sliding window** of size `s1.length()` within `s2`.
- This **first window** checks the first `windowSize` characters of `s2`.

Example: If `s1 = "ab"` and `s2 = "eidbaooo"`,

- `windowSize = 2`
- First window in `s2` → `"ei"`

  ```
  count2['e' - 'a'] = count2[4] = 1
  count2['i' - 'a'] = count2[8] = 1
  ```

Step 4: Checking First Window

```
if (checkEqual(count1, count2))
    return 1;
```

- After setting up the first window, we **check if `count1` and `count2` are equal**.
- If they match, **`s1` is a permutation of this substring**, so we return `true`.

Step 5: Sliding the Window Through `s2`

```
while (i < s2.length()) {
    char newChar = s2[i];
    int index = newChar - 'a';
    count2[index]++;

    char oldChar = s2[i - windowSize];
    index = oldChar - 'a';
    count2[index]--;
```

```
    i++;

    if (checkEqual(count1, count2))
        return 1;
}
```

- **New Character Addition:** The next character of `s2` is added to the window (`newChar`).
- **Old Character Removal:** The leftmost character from the previous window is removed (`oldChar`).
- **Window Shift:** We slide the window one step to the right.
- **Check Again:** If `count1` matches `count2`, return `true`.

Example Execution For `s1 = "ab"`, `s2 = "eidbaooo"`:

1. **Window: `"ei"` → No match**
2. **Window: `"id"` → No match**
3. **Window: `"db"` → No match**
4. **Window: `"ba"` → Match! 🎯**
    - `"ba"` is a permutation of `"ab"`, so the function **returns `true`**.

Time Complexity Analysis

- **Building `count1` → O(26) ≈ O(1)**
- **Initializing first window → O(windowSize)**
- **Sliding the window → O(n)**
- **Checking permutation (using `checkEqual()`) → O(26) ≈ O(1)**

Total Complexity: [O(n)] which is optimal.

Summary

✔ **Approach Used: Sliding Window + Frequency Counting**
✔ **Why This Works?** Instead of checking all permutations explicitly, we use frequency comparison to check in constant time.
✔ **Time Complexity: O(n)** → Efficient compared to brute-force **O(n! × m)**.
✔ **Space Complexity: O(1)** → Uses fixed-size arrays of size 26.

Dry Run (Code Flow)

Great! Now, let's walk through the **flow of the code step by step** using the input:

Input:

```
s1 = "ab"
s2 = "eidbaooo"
```

Step 1: Create Frequency Array for `s1`

```
int count1[26] = {0};
for (int i = 0; i < s1.length(); i++) {
    int index = s1[i] - 'a';
    count1[index]++;
}
```

We calculate the frequency of each character in `"ab"`:

- `'a' - 'a' = 0` → `count1[0] = 1`
- `'b' - 'a' = 1` → `count1[1] = 1`

Final `count1` array:

```
count1 = [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

Step 2: Initialize Sliding Window in `s2`

We start with a **window of size 2 (`s1.length()`)**.

```
int i = 0;
int windowSize = s1.length();
int count2[26] = {0};

while (i < windowSize && i < s2.length()) {
    int index = s2[i] - 'a';
    count2[index]++;
    i++;
}
```

Characters in the first window `"ei"`:

- `'e' - 'a' = 4` → `count2[4] = 1`
- `'i' - 'a' = 8` → `count2[8] = 1`

Final `count2` after initializing the first window:

```
count2 = [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

Step 3: Check First Window

```
if (checkEqual(count1, count2)) return 1;
```

- `count1` ≠ `count2`, so we **move the window forward**.

Step 4: Slide the Window

Now, we slide the window one character at a time while checking for permutations.

Iteration 1 (`s2` window: `"id"`)

1. **New character added:** `'d'` → `count2['d' - 'a'] = count2[3]++`
2. **Old character removed:** `'e'` → `count2['e' - 'a'] = count2[4]--`

Updated `count2`:

```
count2 = [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

- `count1 ≠ count2`, continue.

Iteration 2 (`s2` window: `"db"`)

1. **New character added:** `'b'` → `count2['b' - 'a'] = count2[1]++`
2. **Old character removed:** `'i'` → `count2['i' - 'a'] = count2[8]--`

Updated `count2`:

```
count2 = [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

- `count1 ≠ count2`, continue.

Iteration 3 (`s2` window: `"ba"`)

1. **New character added:** `'a'` → `count2['a' - 'a'] = count2[0]++`
2. **Old character removed:** `'d'` → `count2['d' - 'a'] = count2[3]--`

Updated `count2`:

```
count2 = [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
```

- 🎯 **Match found!** `count1 == count2`, so we return `true`.

Final Output

```
return true;
```

Since `"ba"` is a permutation of `"ab"`, `s2` contains `s1`'s permutation.

☑ **Final Output:** `true`

Summary of the Sliding Window Process

Step	Window in `s2`	Added Char	Removed Char	`count2` Array	Match?
Init	`"ei"`	-	-	`[0, 0, 0, 0, 1, 0, ..., 1, 0, ...]`	✗
1	`"id"`	`'d'`	`'e'`	`[0, 0, 0, 1, 0, 0, ..., 1, 0, ...]`	✗
2	`"db"`	`'b'`	`'i'`	`[0, 1, 0, 1, 0, 0, ..., 0, 0, ...]`	✗
3	`"ba"`	`'a'`	`'d'`	`[1, 1, 0, 0, 0, 0, ..., 0, 0, ...]`	☑ **Match!**

Key Takeaways

- **Used Sliding Window + Frequency Counting** for an **efficient O(n) solution**.
- **Shifted window one character at a time** while updating counts.
- **Matched frequency arrays** to find a permutation.

Would You Like to Try Another Example? 🚀