1. Using Sorting

A naive approach might involve checking all possible pairs of strings, which would be inefficient. However, this optimized solution uses the sorted version of each string as a unique key for grouping. Sorting ensures that all anagrams produce identical keys, simplifying the identification of groups. Sorting each string reduces the problem to key generation, which is straightforward to handle using a `Map`.

Algorithm

1. Initialize an empty `Map` to store groups of anagrams.
    - The keys represent the sorted version of strings, and the values are arrays of original strings.
2. Iterate over the input array of strings.
    - For each string, convert it into a character array and sort it alphabetically.
    - Join the sorted characters to create a key.
3. Check if the key already exists in the `Map`.
    - If the key does not exist, initialize it with an empty array.
    - Append the original string to the array corresponding to the key.
4. Convert the values of the `Map` into an array of arrays and return it.

```
export default function anagramGroups(strs: string[]): string[][] {
 // Create a map to store the anagrams.
 // The keys are the sorted versions of the strings,
 // and the values are arrays containing the original strings (anagrams).
 let map = new Map<string, string[]>();

 // Iterate over each string in the input array
 for (let str of strs) {
   // Convert the string to a character array and sort it alphabetically
   let chars = Array.from(str);
   chars.sort();

   // Join the sorted characters to form a key
   // This key will be the same for all anagrams
   let key = chars.join('');

   // If the key doesn't exist in the map, add it with an empty array as the value
   if (!map.has(key)) map.set(key, []);
```

```
    // Add the original string to the array of anagrams for this key
    map.get(key)!.push(str);
  }

  // Convert the map's values to an array and return it.
  // Each element in the array is a group of anagrams.
  return Array.from(map.values());
}
```

Big-O analysis

- **Time complexity: O(n.m log m)**. Sorting each string of length `m` takes O(m log m), and there are `n` strings.
- **Space complexity: O(n.m)**. The `Map` stores all strings, and the space required for keys and values depends on the total characters in the input.

2. Using Frequency Count

Instead of sorting strings to generate keys like previous solution, a frequency array of fixed size (26 for the English alphabet) is used to count occurrences of each character. This solution optimizes the process by representing each string using its character frequency count. This eliminates the need for sorting, which is the bottleneck in the sorting-based approach.

The key for each group is derived from the frequency count, formatted as a concatenated string with counts of all characters. This ensures all anagrams map to the same key while avoiding sorting overhead, improving the efficiency of the solution.

Algorithm

1. Check if the input array is empty.
    - If the array is empty, return an empty array.
2. Initialize an empty object to store groups of anagrams.
    - The keys represent character count patterns, and the values are arrays of strings matching the pattern.
3. Iterate over each string in the input array.
    - Initialize a count array of size 26 to represent character frequencies.
    - For each character in the string, increment the corresponding index in the count array.

4. Generate a key string from the count array.
    - Concatenate `#` and the count of each character to form the key.
5. Check if the key exists in the object.
    - If the key does not exist, initialize it with an empty array.
    - Append the string to the array corresponding to the key.
6. Return the values of the object as an array of groups of anagrams.

```
export default function anagramGroups(strs: string[]): string[][] {
 // If the input array is empty, return an empty array
 if (strs.length === 0) return [];

 // Initialize an object to store groups of anagrams
 // The keys are unique representations of the character counts
 // The values are arrays of strings that match the key's character count
 let ans: { [key: string]: string[] } = {};

 // Iterate over each string in the input array
 for (let s of strs) {
  // Create an array to count occurrences of each character (26 letters of the alphabet)
  let count: number[] = Array(26).fill(0);

  // Increment the corresponding index in the count array for each character in the string
  for (let c of s) count[c.charCodeAt(0) - 'a'.charCodeAt(0)]++;

  // Create a key string based on the character counts
  // The key is formatted as a series of "#count" for each character
  let key = '';
  for (let i = 0; i < 26; i++) {
   key += '#';
   key += count[i];
  }

  // If the key doesn't exist in the map, initialize it with an empty array
  if (!ans[key]) ans[key] = [];

  // Add the original string to the array associated with the key
  ans[key].push(s);
 }

 // Return the values of the map, which are the groups of anagrams
```

```
    return Object.values(ans);
}
```

Big-O analysis

- **Time complexity: O(n.m)**. Counting characters for each string of length `m` takes O(m), and there are `n` strings.
- **Space complexity: O(n.m)**. The object stores all strings, and additional space is used for character counts and keys.