**Code Jam 2014 - Round 1C**

# Analysis: Reordering Train Cars

This is a straightforward problem, but the main issue to get it correct is to enumerate all invalid cases. The main idea of the solution is to join some strings to create sets of disjoint groups, such that every group contains a set of strings that must be joined together. Then calculate the number of ways to create every group, and the final result will be the factorial of number of disjoint groups multiplied by number of ways to create every group.

One of the main issues (that lead to huge number of incorrect attempts) was to validate that the given set of strings is invalid. So let us start the analysis by specifying valid cases that should be handled before creating the groups.

Another good hint that will help a lot in validation process is to collapse all given strings by removing all duplicate consecutive letters, for example, if given a string `"aabbccdddeaa"`, after collapsing this string it should be `"abcdea"`. So from now, we will assume that all given strings are always collapsed, even after joining two strings `"ab"` and `"bc"` this will be collapsed automatically to be `"abc"` not `"abbc"`.

## Validation Process

First, check that for every string there is no letter repeated in two different places in the string. For example, `["aba", "abca", "adeab"]` are all invalid strings as `'a'` is repeated, while `["ab", "abc", "adbe"]` are valid strings.

We start by precomputing some information that will help to enumerate the rest of invalid cases. We need to store the position of every character ***c*** in the alphabets, i.e. all letters from `'a'` to `'z'`. Character ***c*** will be either (i) the first character, (ii) the last character, or (iii) a middle character of any given non-single-character string. Also, we need to count the number of single-character strings for every character ***c***. For example, if the given list of collapsed strings is `["abc", "cdef", "a", "a", "gh"]`, then the precomputed arrays are:

```
begin = {'a': 0, 'c': 1, 'g': 4}   -- (1)
end = {'c': 0, 'f': 1, 'h': 4}     -- (2)
middle = {'b': 0, 'd': 1, 'e': 1}  -- (3)
singleChars = {'a': 2}             -- (4)
```

*Notes*:

1. `begin` means that for any character **`c`**, it is the first character of string number **`begin[c]`**.
2. `end` means that for any character **`c`**, it is the last character of string number **`end[c]`**.
3. `middle` means that for any character **`c`**, it is in the middle of string number **`middle[c]`**. Notice that, `"cdef"` (string number 1) has two middle characters, while `"gh"` (string number 4) does not have middle characters.
4. `singleChars` means that for any character **`c`**, There are **`singleChars[c]`** single-character strings composed of character **`c`**. For example, there are 2 single-character strings of character `'a'`.

While precomputing the previous arrays, you have identify a valid set of strings by sustaining the following conditions. A character is valid only if it satisfies one of the following conditions:

1. It appears in the middle of one string, and nowhere else.

2. It appears at most once in the beginning of one string, at most once at the end of one string, and any number of times as a single-character string.

For example, all the following cases are invalid:

- `["abc", "ade"]`: character `'a'` conflicts with second condition.
- `["bca", "dea"]`: character `'a'` conflicts with second condition.
- `["abc", "dbf"]`: character `'b'` conflicts with second condition.
- `["abc", "ead"]`: character `'a'` conflicts with first condition.
- `["abc", "gcf"]`: character `'c'` conflicts with first condition.
- `["a", "a", "bac"]`: character `'a'` conflicts with first condition.

By considering the previous conditions, we will be able to identify valid cases in any given set of strings.

# Groups creation

Let us define first what it is a group, the group is a set of strings that should be joined to form a valid combined string. So in this section we focus on creating a set of disjoint groups to count the number of ways. For example, `["ab", "bc"]` is a group as `"ab"` and `"bc"` should be joined together to form `"abc"`, which is a valid string. Also, assume given `["ab", "bc", "de", "xy", "yz"]`, this set of strings can be divided into 3 disjoint groups, which are `["ab", "bc"]`, `["de"]`, and `["xy", "yz"]`.

First, we need to handle the special case of single-character strings. Every single-character string can be considered initially as a group, with the number of ways to create it being equal to the factorial of the number of such strings, for example, `["a", "a", "a"]` can be grouped in 3! (which is 6 different ways). And this group will be represented by only one character which is `"a"` and not `"aaa"` (as we described in the beginning, we will collapse these strings too).

An important point to notice while creating groups is that the first string in the group should start with a character that is not used as the last character in any other string. For example, the following 2 strings `["abc", "cde"]` can form a valid group that should be represented by string `"abcde"`, we notice that we cannot start forming the group with string `"cde"` as it starts with character `'c'` which is the last character of string `"abc"` (we can make this check in constant time by using the ***end*** array). Also, notice that this group can be created in only 1 way.

We now proceed with counting the *number of disjoint groups*. We loop over all non-single-character strings that start with a character not used as the last character of any other string and use this as the first string in the group, then join it to the string that starts with its last character (get the index of this string using ***begin*** array) and so on till you reach the last string in that group (where you cannot find any other string to join it to). For example:

```
["cde", "mno", "abc", "opq", "xyz"]
From these strings we can form 3 disjoint groups as follows:
  1) "mno" + "opq" = "mnopq"
  2) "abc" + "cde" = "abcde"
  3) "xyz"
```

After counting the number of disjoint groups there is one more step remaining which is to join any single-character string to an disjoint group, if possible. As stated previously, we assumed initially that every single-character string is considered as a disjoint group, so we may assume that total number of disjoint groups equals to number of non-single-character groups and number of single-character groups. But there are some cases that we need to join a single-character strings to a one of the non-single-character strings. We can elaborate that with the following two examples:

- `["ab", "b", "b"]`: you must join the single-character string `"b"` to the end of string `"ab"` to form the group `"ab"` in 2 ways, i.e. 2 ways for the 2 single-character strings and only 1 way to join the single-character string to `"ab"`.
- `["a", "a", "bc"]`: in this case we consider `"a"` as an disjoint group and will not be joined to any other groups. So there are 4 ways, i.e. 2 ways for the single-character string, and 2 ways to order `"a"` and `"bc"` either `"abc"` or `"bca"`.