Passwords with Constraints

You are given a list of **n** lower case alphabets and 4 symbols namely $,#,&,@ . You need to print all **unique** passwords of length **k** in a sorted order. After a couple of hits and trials, you can figure out easily that all symbols can be used any number of times but the alphabets are limited to their occurrence in the alphabets. This conclusion can also be derived from the fact that the alphabet array has repetitions allowed (This wouldn't have been mentioned if repetition of use of alphabets from the list was allowed in the string).

Points to Note from the question

1. All the passwords generated should be **unique**. Hence we need to avoid generating the same pattern twice.
2. All the passwords should be present in **sorted** order.
3. The symbols in the **sorted** order are **#, $, &, @** . This can be obtained from the sample test case associated with the question.
4. Symbols **can** be repeated. Although it is not mentioned, unless otherwise mentioned or indicated by examples ,assume repetitions are allowed.
5. Alphabets can be repeated in their list. This is the challenge in ensuring **unique** passwords. This also conveys, although very vaguely, that repetition of alphabets is **restricted** to their occurrence in the list.
6. Although the question is very vague,the doubts regarding the fact mentioned in point 4 and point 5 can be clarified by multiple submissions on each alternative of these options. Just a trick to be noted in case you ever vague questions in any challenge in the future

Brute Force

The brute force approach involves recursing over all alternatives and storing them in a list.Then the list can be sorted and repetition can be removed from the sorted array while printing.Some common optimizations over brute force

- Since the resultant list of passwords have repetitions possible, using a set or map(dictionary in python) to store passwords would help maintain only unique passwords.
- Since the resultant list of passwords have to be sorted, considering the list of alphabets and symbols in sorted order would create all the passwords in the sorted order and hence sorting won't be needed at the end. As mentioned in the beginning, the symbol in the sorted order are `[ # , $ , & , @ ]`.

Although the following optimizations are helpful they don't help us solve the problem optimally.

Optimal Solution

Before getting into what's the most optimal way to solve the problem, let's dive into why the above optimization methods aren't sufficient. Obtaining the passwords in sorted order will save us the time of sorting, but isn't sufficient improvement due to the large number of repetitions that are possible .
For example, for k = 2 and alphabets = `[‘a’,‘a’,‘a’,‘a’,‘a’,‘a’,‘a’]` , the same set of patterns i.e. `[ ‘#a’,‘$a’,‘&a’,‘@a’,‘a#’,‘a$’,‘a&’,‘a@’]` repeated over 6 times. For larger values of k, this problem caused due to repetitions is a bigger issue than sorting.

The other optimization involved using sets or maps to avoid including repetition of patterns in our final solution. Although this would reduce space usage and hence would save memory and coupled with the previous optimization would also help generate non repeated sorted order of passwords. However, this doesn't optimize the time required to reach the solution. This can be understood easily by the following example.
Say K = 3, and alphabets = [‘a’, ‘a’, ‘b’] .
Suppose the current state of recursion involves ‘#’ as the password. Now an alphabet is considered. While considering the alphabets ‘a’ gets considers twice i.e. recursion for the password ‘#a’ occurs twice. The 2nd recursion for ‘#a’ gets discarded only when the final password is obtained during the base check for `passwordLengh == k.` If we can prune the recursion by avoid the 2nd recursion for ‘#a’ in this case, we can reduce the time complexity to a large extent.

The logic behind doing the above optimization is similar to removing duplicates for a sorted array (a question we have done in concept Array ME). Since, we need to avoid the 2nd recursions for the same character, we need to avoid picking the alphabet in the same recursion state again. This can be done by skipping the duplicates while traversing the alphabet list during recursion. Since the list of alphabets is already sorted (to obtain the passwords in the sorted order), we can easily skip duplicates by considering the last alphabet it was recursed upon.