Permutation:

**Solution**

Let’s discuss a few basics first. We know that n!n! is the number of permutations for a set of size nn. Another obvious and important concept is that if we choose an element for the first position, then the total permutations of the remaining elements are (n−1)!(n−1)!.

For example, if we’re given the string “abcd” and we pick “a” as our first element, then for the remaining elements we have the following permutations:

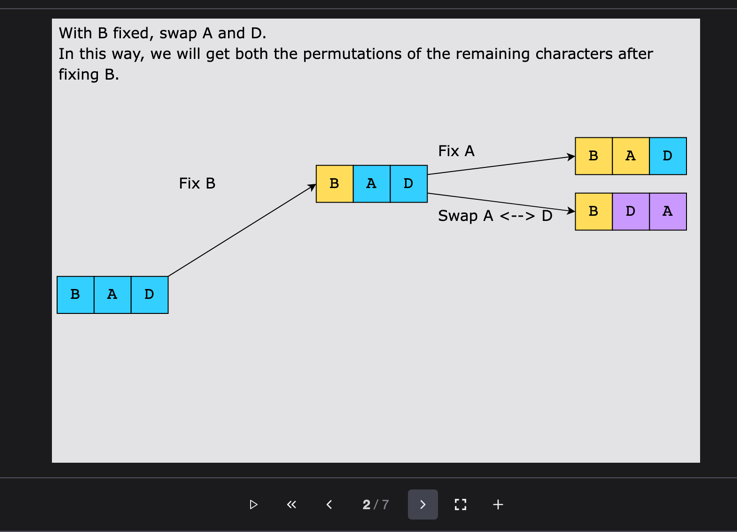
Similarly, if we pick “b” as the first element, permute “acd”, and prepend each permutation with “b”, we can observe a pattern here as shown in the illustration above. That pattern tells us how to find all remaining permutations for each character in the given string.

We can do this recursively to find all permutations of substrings, such as “bcd”, “acd”, and so on. This implies that generating all possible permutations of the given string involves exploring different combinations of characters, which can be done efficiently using the subset technique. The key idea is to take one character of the given string at a time and find all the permutations that start with this chosen character. For this, imagine filling empty positions equal to the length of the string in the following manner: place the chosen character at the first position, then against this character, try all the remaining characters in the second position. Next, for each pair of characters in the first and second positions, try all the remaining characters in the third position. Keep doing this until we reach the last position to be filled. This process will allow us to systematically arrange each character in different positions and generate all possible permutations of the given string.

Here is a visual representation of all recursions for input string “bad”:

A screenshot of a computer

AI-generated content may be incorrect.

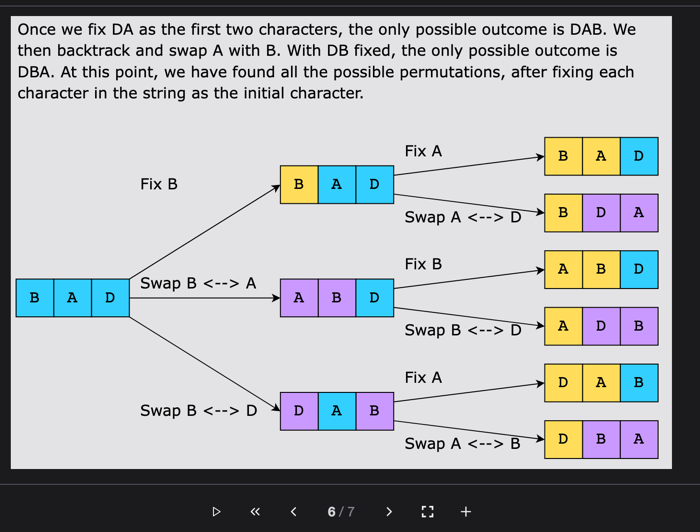
A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



### A screenshot of a computer AI-generated content may be incorrect.**Step-by-step solution construction**

Let’s start with the simplest step—swapping the indexes of the input string. We create a function to swap ithith and jthjth indexes of a given string.

The next part of the solution construction consists of actually calculating the first permutation for the input string. After swapping, we need to find the permutations as well. Let’s see how we can implement this logic.

We create a recursive function to compute the permutations of the string that has been passed as input. The function behaves in the following way:

* We fix the first character of the input string and swap it with its immediate next character.
* We swap the indexes and get a new permutation of the string, which is stored in the variable, swappedStr.
* The recursive call for the function increments the index by adding 11 to the currentIndex variable to compute the next permutation.
* All permutations of the string are stored in the result array, but for the following code snippet, only the base case—that is, only the first permutation—is displayed.

We first swapped the indexes and then displayed the first possible permutation of the input string, that is, the input string itself. Let’s update the functionality to compute and store all possible permutations.

To implement this logic, we’ve made a function that takes the string as input, calls the recursive function, permuteStringRec that computes all permutations (as done in the previous step), and finally stores and displays the result array containing the permutations.

### Solution summary

Let’s review what approach we’ve used to solve the problem above:

1. Fix the first index and keep swapping the character at this index with the other characters of the string one by one.
2. After each swap, skip the first character and recursively compute the permutation of the remaining string.
3. Add the string to the result when the last character of the string is reached.

### Time complexity

Let’s anaylze the time complexity of the solution code above:

* permuteWord()**:** There are n!n! (factorial of nn) permutations of a string of length nn.
* permuteStringRec()**:** This is a recursive function that generates these permutations. For a string of length nn, there are nn recursive calls at the first level, n−1n−1 calls for the second, and so on. This results in a total of n×(n−1)×(n−2)×⋯×1=n!n×(n−1)×(n−2)×⋯×1=n! calls.
* swapChar()**:** The swap function has a time complexity of O(n)O(n) since it involves creating a list from the string and then joining it back into a string after the swap. This operation is done for each recursive call.

So, the overall time complexity of generating all permutations of a string of length nn is O(n×n!)O(n×n!)

### Space complexity

The space complexity of this solution is dependent on the depth of the recursive call stack. The maximum depth of recursion is nn, so the space complexity is O(n)O(n).