Solution: Permutations

Let's solve the Permutations problem using the Subsets pattern.

We'll cover the following

Statement

Given an input string, return all possible permutations of the string.

> **Note:** The order of permutations does not matter.

Constraints:

- All characters in the input string are unique.
- $1 \leq$ `word.length` ≤ 6

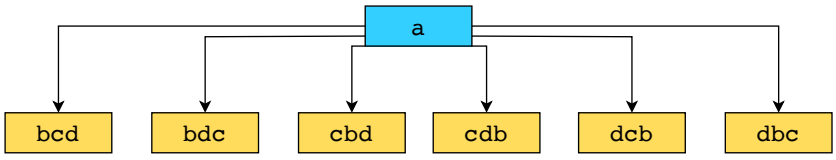
Pattern: Subsets

Problems such as this one, where we need to find the combinations or permutations of a given string, are good examples to solve using the subsets pattern as it describes an efficient **Breadth-First Search (BFS)** approach to handle all these problems.

Solution

Let's discuss a few basics first. We know that $n!$ is the number of permutations for a set of size n. 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 $(n-1)!$.

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:

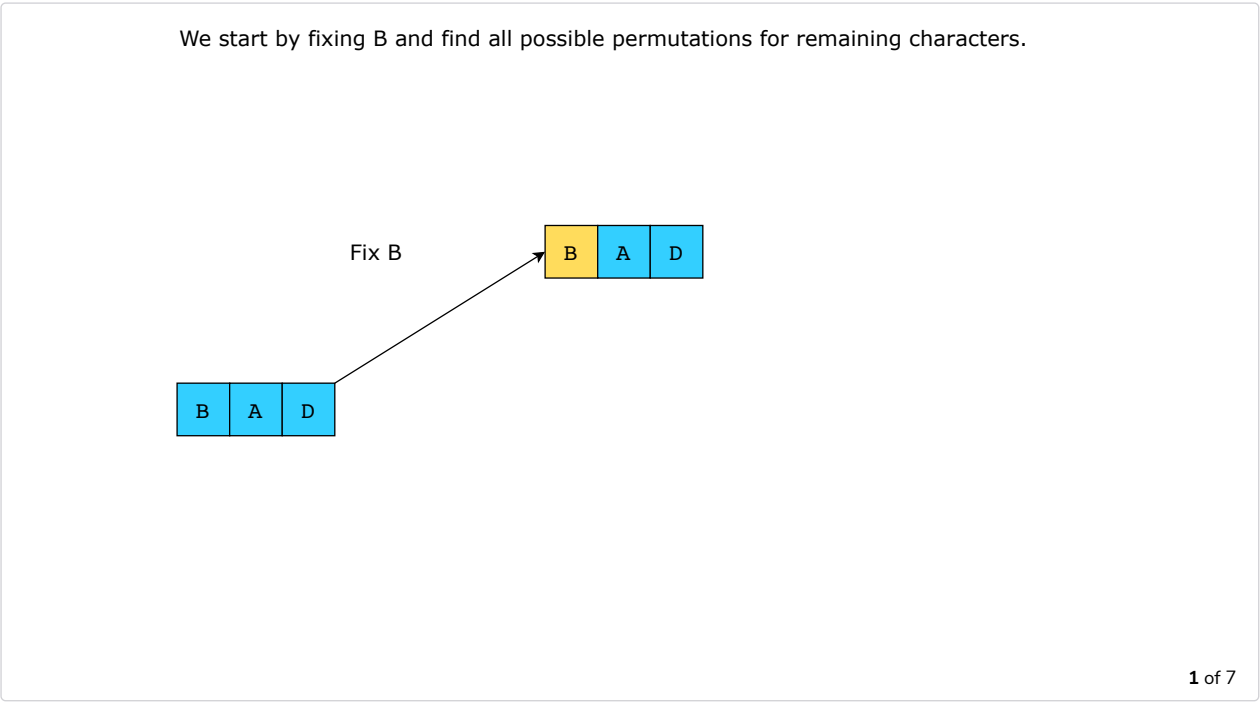


Similarly, if we pick "b" as the first element, permute "acd", and prepend each permutation with "b", we can observe a pattern here as shown in the illustration above. That pattern tells us how to find all remaining

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.

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



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

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 i^{th} and j^{th} indexes of a given string.

Java

```java
class Permutations {

    // This function will swap characters for every permutation
    public static char [] swapChar(String word)
    {
        char[] swapIndex = word.toCharArray();
        char temp = swapIndex[0];
        swapIndex[0] = swapIndex[1];
        swapIndex[1] = temp;
        return swapIndex;
    }
    // Driver code
    public static void main( String args[] ) {
        String[] inputWord = {"ab", "bad", "abcd"};
        for (int index = 0; index < inputWord.length; index++)
        {
            char [] permutedWords = swapChar(inputWord[index]);
            System.out.println(index + 1 + ".\t Input string: '" + inputWord[index] + "'");
            System.out.println("\t Swapping character at index 0 with index 1");
            System.out.println("\t Swapped indices: " + Arrays.toString(permutedWords));
             System.out.println(PrintHyphens.repeat("-", 100));
        }
    }
```

```
24  }
```

Permutations

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 1 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.

Java

```java
class Permutations {

    // This function will swap characters for every permutation
    public static char [] swapChar(String word , int i, int j)
    {
       char[] swapIndex = word.toCharArray();
       char temp = swapIndex[i];
       swapIndex[i] = swapIndex[j];
       swapIndex[j] = temp;

       return swapIndex;
    }

    public static void permuteStringRec(String word, int currentIndex, ArrayList<String> results)
    {
        // Prevents adding duplicate permutations
        if(currentIndex == 1)
        {
            results.add(word);
            return;
        }
        for (int index = currentIndex; index < word.length(); index++)
        {
            // swaps character for each permutation
            char [] swappedStr = swapChar(word, currentIndex, index);
            System.out.println("\t After swapping the indices in the string: " + String.valueOf(swappedSt
            // recursively calls itself to find each permutation
            permuteStringRec(String.valueOf(swappedStr), currentIndex + 1, results);
```

Permutations

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.

Java

```java
class Permutations {

    public static char [] swapChar(String word , int i, int j)
    {
       char[] swapIndex = word.toCharArray();
       char temp = swapIndex[i];
       swapIndex[i] = swapIndex[j];
       swapIndex[j] = temp;

       return swapIndex;
    }
    public static  void permuteStringRec(String word, int currentIndex, ArrayList<String> results)
    {
        // Prevents adding duplicate permutations
        if(currentIndex == word.length() - 1)
        {
            results.add(word);
            //System.out.println(Arrays.toString(results));
            return ;
        }
        for (int index = currentIndex; index < word.length(); index++)
         {
            // swaps character for each permutation
            char [] swappedStr = swapChar(word, currentIndex, index);

            // recursively calls itself to find each permutation
            permuteStringRec(String.valueOf(swappedStr), currentIndex + 1, results);
         }
```

Permutations

Just the code

Here's the complete solution to this problem:

Java

```java
class Permutations {

    public static char [] swapChar(String word , int i, int j)
    {
       char[] swapIndex = word.toCharArray();
       char temp = swapIndex[i];
       swapIndex[i] = swapIndex[j];
       swapIndex[j] = temp;

       return swapIndex;
    }
    public static  void permuteStringRec(String word, int currentIndex, ArrayList<String> results)
    {
        if(currentIndex == word.length() - 1)
        {
            results.add(word);
            return ;
        }
        for (int index = currentIndex; index < word.length(); index++)
         {
            char [] swappedStr = swapChar(word, currentIndex, index);
            permuteStringRec(String.valueOf(swappedStr), currentIndex + 1, results);
         }
    }
    public static ArrayList<String> permuteWord(String word)
    {
        ArrayList<String> results = new ArrayList<String>();
        permuteStringRec(word, 0, results);
```

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

The time complexity of computing permutations for an input string is $O(n!)$, where n is the length of the input string.

Space complexity

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

Back
Permutations

Next
Letter Combinations o...

Mark as Completed