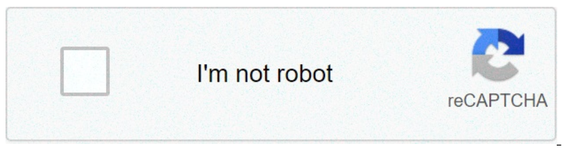

Continue

Find permutations of a string java

In this blog we are going to find out all the permutations of a given string. Like always, we will first start from the basics – Understand what is a permutation of a string, break the procedure down into smaller steps and understand the procedure of finding out one permutation. Finally, we will write code in Java for the same. Along the way, I will also be explaining each line code and show you pictorial representations of the code execution so that you can visualize it better. What is a permutation ? Different ways of arranging a set of items is permutation. A string is a sequence of characters. Permutation of a string is arranging the characters of the string in different ways. Let's take an example to understand this. Permutation of a String The string "ace" can be arranged as "ace", "aec", "cae", "cea", "eac","eca" – different arrangements of the characters a,c,e which make the string "ace". Note that the string "ace" is of length 3 and we get 6 different permutations of the same – 3 factorial. If the input string was "aced", we will get 24 permutations – 4 ! Observation about the permutations Did you notice that the first 2 permutations "ace", "aec" have a as the first letter and the 2 other letters are then concatenated to the letter a. Extracting the first character 'a' from "ace" leaves us with the remaining characters "ce". It can be rearranged as "ce" , "ec". Finally appending each to "a" results in "ace" and "aec". If we single out the character 'c' in ace, we are left with "ae". With "ae", rearranging them gives us "ae" and "ea". Appending this to 'c' results in "cae" and "cea". If we single out the character 'e' in ace, we are left with "ac". With "ac", rearranging them results in "ac" and "ca". Appending this to 'e' results in "eac" and "eca". Translating these observations into a technical perspective We are taking a single character from the given string, starting with 'a', moving on to 'c' and finally visiting 'e'. This can be done using the charAt function. We also need a for loop as we need to single out each character from the string. for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); ... } For each character that we extract, we need the rest of the string. So when we extract 'a' from the "ace", we need "ce" so that we can have different arrangements of "ce" to append it to 'a'. When we extract 'c' from "ace", we need "ae". When we extract 'e' from "ace", we need "ac". In short, when we are at a particular iteration , i , in the for loop, we need a string from the characters before and after that character. How do we extract the remaining characters ? We make use of the substring function to do that. The variable 'i' in the for loop points to current single character that we extract. When we extract 'c' from "ace", we need to get "ae". We can get all characters before i by making a call to substring(0,i) and everything after i by calling substring(i+1). We append them to get the remaining string. When i =1 and 'c' is extracted for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); String next = remainingString.substring(0,i) + remainingString.substring(i+1); ... } Applying recursion What is clear so far is that we start with the first character and apply permutation with remaining characters. Then we choose the second character and apply permutation with remaining characters. We continue this way until we visit each character in the string. To do something like this, recursion can be a good choice. So let us take the code above and add it to a function, permutations. public void permutations(String remainingString , String permutation) { for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); String next = remainingString.substring(0,i) + remainingString.substring(i+1); //Code here for recursive call to permutations } } This function takes 2 parameters – remainingString and permutation. Why do we need them ? Well, the parameter remainingString keeps track of length of string to produce one complete permutation of current string.The permutation parameter will keep track of the current permutation.The first time this code starts executing, the remainingString will be the input string, "ace", and the permutation will be a blank string, "", since we are yet to start finding permutations. Now we start with 'a',fix that and then extract "ce". So 'a' will be stored in ch and "ce" will be stored in variable referred to as next. What do we have so far ? remainingString = "ace", permutation = "", ch = 'a', next = "ce" What should the next step be? The variable, permutation, so far is "", it should be "a". It is not a valid end permutation but an intermediate one. The current value is a "". Appending "" to "a" gives us "a". So let's define a variable permute and assign it to permutation +ch. String permute = permutation+ch; So the variable permute = "" + "a" = "a" public void permutations(String remainingString , String permutation) { for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); String permute = permutation+ch; String next = remainingString.substring(0,i) + remainingString.substring(i+1); //Code here for recursive call to permutations } } Now, remainingString = "ace", permutation = "", ch = 'a', next = "ce", permute ="a" The next logical step is working on "ce" to extract 'c'. Once that is done, the intermediate permutation is "ac". "a" from the previous iteration and 'c' extract from current one. When we extract 'c' from "ce", what remains is "e". This is the same sequence as previous steps. The solution seems to repeat for the next sub-problem. This can be solved by recursion. We need to call the permutations function. It takes 2 parameters – remainingString and permutation. The variable, next, has value "ce" and permutation currently is "a". Let's make a call to permutations function and pass these parameters. public void permutations(String remainingString , String permutation) { for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); String permute = permutation+ch; String next = remainingString.substring(0,i) + remainingString.substring(i+1); //Code here for recursive call to permutations permutations(next,permute); } } After the first recursive call, remainingString = "ce", permutation = "a". When code above starts execution, i = 0 , ch = 'c' , permute = "a" + 'c' = "ac" , next = "e". Note that when this call happens, i = 0 . This block will get executed twice as the for loop checks for length of remainingString. More on this later. Then there is a recursive call again to the function by passing "e", "ac". In the next iteration, remainingString = "e", permutation = "ac". When the code starts executing, i = 0 , ch = 'e' , permute = "ace" , next = "". Then there is a call to recursive function with "" and "ace". Now , remainingString = "" , permutation ="ace". It looks like the remainingString is a blank string along with the fact that permutation is "ace". We are in a recursive function, every recursive function should have some condition to return if it has processed it's sub-problem. This part is now solved, isn't it ? So we need a terminating condition – the length of the variable, remainingString, can be that condition. We simply check if it's length is zero. public void permutations(String remainingString , String permutation) { if(remainingString.length() == 0) { System.out.println(permutation); return ; } for(int i = 0; i < remainingString.length();i++) { char ch = remainingString.charAt(i); String permute = permutation+ch; String next = remainingString.substring(0,i) + remainingString.substring(i+1); permutations(next,permute); } } After the execution of this code we get "ace" and this function returns. The code execution continues from the the location that it was called – this is really the previous step. Take a look at the following flows to get a better understanding. Start from the block which says start and then the steps have been numbered.Push and pop indicates recursive function calls and returning back from a function. Snapshot of the function calls when i = 0 and input is "ace" Produces 2 permutations "ace" and "aec" Step 1 will get executed twice as length of "ce" is 2. When i is 0, we get "ace" and when i =1 , we get "aec". When the function returns from step 2, we go back to step 1 where i will become 1. This will cause step 4 to be executed. Note that, all these steps are happening when input is "ace" and i = 0. When i =1, a similar set of steps will be executed producing 2 more permutations. Snapshot of the function calls when i = 1 and input is "ace" Produces 2 permutations "cae" and "cea" Snapshot of the functional calls when i = 2 and input is "ace" Produces 2 permutations "eac" and "eca" The images below will give you a more detailed view of how the code and function calls execute. We call the function, permutations, pass the parameters "ace" and "". Snapshot of the code execution when input is "ace" and i=0 These steps produce 2 permutations – ace and aec as seen in steps 3 and 5 Snapshot of the code execution when input is "ace" and i=1 These steps produce 2 permutations – cae and cea as seen in steps 8 and 10 I have left out the code tracing when i=2, it can be a good exercise for you. Conclusion The number of lines of code that we had to write to produce the permutations is small but there is a lot that is happening behind the scenes.Such kind of problems are being asked in technical interviews. Irrespective of this, I think recursion is an important concept to understand and it is also a good fit for a good number of problems. Recursion is not very straight forward to understand but I hope the pictorial representations and breaking up the code step by step have given you a good understanding of the same. I urge you to take a piece of paper and trace the execution for one particular iteration – this will not only solidify your understanding of the solution to the permutation problem but help you sharpen your skill set by understanding recursion. In mathematics, the notion of permutation is used with several slightly different meanings, all related to the act of permuting (rearranging) objects or values. Informally, a permutation of a set of objects is an arrangement of those objects into a particular order. For example, there are six permutations of the set {1,2,3}, namely (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1). Here is a quick simple Algorithm which computes all Permutations of a String Object in Java. First take out the first char from String and permute the remaining chars If String = "123" First char = 1 and remaining chars permutations are 23 and 32. Now we can insert first char in the available positions in the permutations. 23 -> 123, 213, 231 22 -> 132, 312, 321 CrunchifyMarmutationExample.java package com.crunchify.tutorials;import java.util.HashSet;public class CrunchifyMarmutationExample { public static void main(String[] args) { System.out.println("String " + s + ":Permutations: " + crunchifyPermutation(s)); System.out.println("String " + s1 + ":Permutations: " + crunchifyPermutation(s1)); System.out.println("String " + s2 + ":Permutations: " + crunchifyPermutation(s2)); public static Set crunchifyPermutation(String str) { Set crunchifyResult = new HashSet(); } else if (str.length() == 0) { char firstChar = str.charAt(0); String rem = str.substring(1); Set words = crunchifyPermutation(rem); for (String newString : words) { for (int i = 0; i