# Solution Review: Find All Permutations of a String

In this lesson, we will review the solution to the challenge from the previous lesson.

**We'll cover the following**


- Solution
- Explanation
- Time complexity

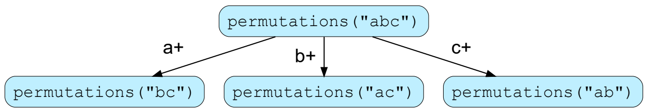

## Solution

```
def permutations(str):
  if str == "": # base case
    return [""]
  permutes = []
  for char in str:
    subpermutes = permutations(str.replace(char, "", 1))    # recursive step
    for each in subpermutes:
      permutes.append(char+each)
  return permutes

def main():
  print (permutations("abc"))

main()
```

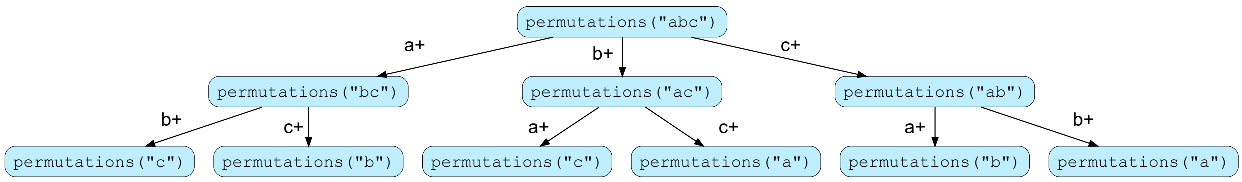
## Explanation

Just as we learned in previous lessons, the key to acing recursion is focusing on one step at a time. Try to think of this problem in this way: you already have every possible arrangement of every possible subsequence of the string `str`. All you need to do now is prepend all the characters to their corresponding list of substrings. Let's see a visualization of this before we jump into the code.

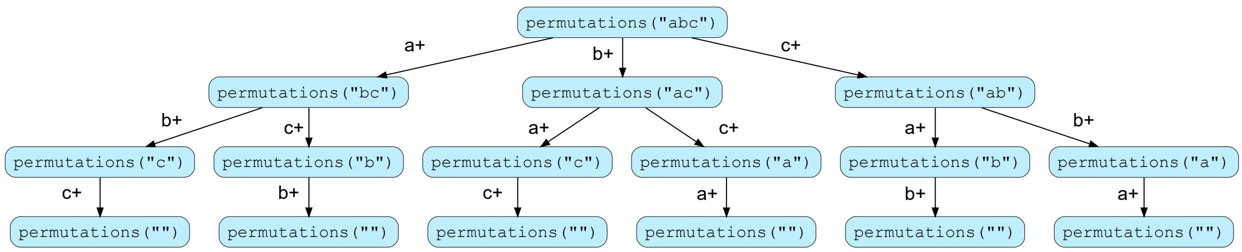

1 of 8



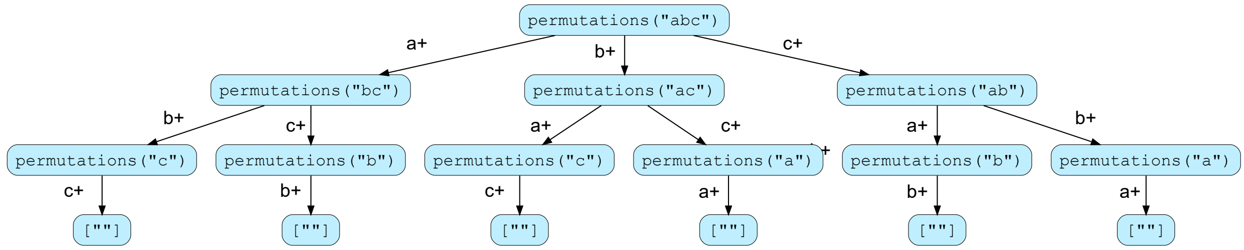

2 of 8



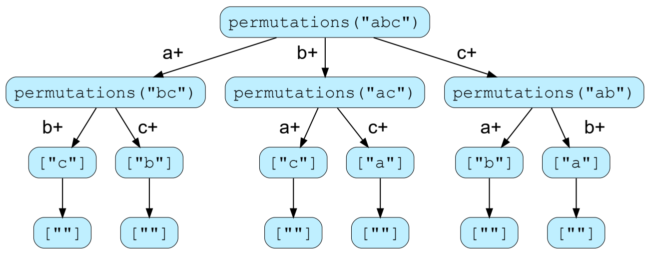

3 of 8



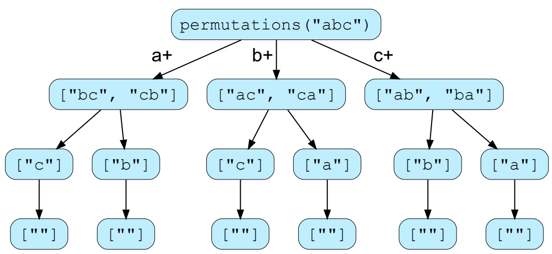

4 of 8



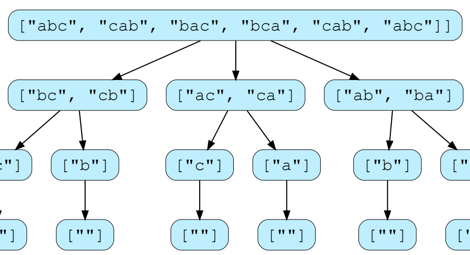

5 of 8




6 of 8




7 of 8




8 of 8

So, what is happening here? At each step, we make as many recursive calls to `permutations` as there are characters in our `str`. For each character, `char`, we find permutations of `str` excluding that character, and simply prepend the character to the results. This is the crux of this algorithm, highlighted in code at *lines 5-8*. We should also be careful about our termination condition. Our algorithm has only one base case of when the string is empty, and it returns a list containing an empty string (*lines 2-3*). You could have had the terminating condition as a string of length 1, but then our algorithm would not have worked for empty strings. This highlights the importance of choosing base cases for a recursive algorithm.

## Time complexity #

Look at the visualization again and notice how at the first step we had three branches, then two branches per branch, and then finally one. Which is `3x2x1 = 3!`. If we were to generalize this to *n*, our time complexity would be **O(n!)**. This makes sense because for an *n* character long string, there are *n!* total permutations.

---

Hopefully, this was a good exercise for you. We have another one waiting in the next lesson.