Interview Cake

Write a recursive function for generating all permutations of an input string. Return them as a set.

Don't worry about time or space complexity—if we wanted efficiency we'd write an iterative version.

To start, assume every character in the input string is unique.

Your function can have loops—it just needs to also be recursive.

Gotchas

Make sure you have a base case!

The **base case** tells a recursive function when to stop. Otherwise it would keep calling itself forever!

For example, we could add all numbers 1 to `n` recursively like this:

Python 3.6

```
def sum_1_to_n(n):
    return n + sum_1_to_n(n-1)
```

If we input 3 as our `n`, this function will take 3, then add 2, then add 1, then add 0, then add -1, then add -2, etc forever (or until the program crashes).

We want to stop adding when `n` gets to 1. We'd say that our "base case" is `n <= 1`, and our code might look like:

Python 3.6

```
def sum_1_to_n(n):

    # base case:
    if n <= 1:
        return 1

    return n + sum_1_to_n(n-1)
```

Whenever writing a recursive function, be careful not to forget the base case!

Otherwise your function may never terminate!

Breakdown

Let's break the problem into subproblems. How could we re-phrase the problem of getting all permutations for "cats" in terms of a smaller but similar subproblem?

Let's make our subproblem be getting all permutations for all characters except the last one. If we had all permutations for "cat," how could we use that to generate all permutations for "cats"?

We could put the "s" in each possible position for each possible permutation of "cat"!

These are our permutations of "cat":

```
cat
cta
atc
act
tac
tca
```

For each of them, we add "s" in each possible position. So for "cat":

```
cat
    scat
    csat
    cast
    cats
```

And for "cta":

```
cta
    scta
    csta
    ctsa
    ctas
```

And so on.

Now that we can break the problem into subproblems, we just need a base case and we have a recursive algorithm!

Solution

If we're making all permutations for "cat," we take all permutations for "ca" and then put "t" in each possible position in each of those permutations. We use this approach recursively:

Python 3.6

```
def get_permutations(string):
    # Base case
    if len(string) <= 1:
        return set([string])

    all_chars_except_last = string[:-1]
    last_char = string[-1]

    # Recursive call: get all possible permutations for all chars except last
    permutations_of_all_chars_except_last = get_permutations(all_chars_except_last)

    # Put the last char in all possible positions for each of
    # the above permutations
    permutations = set()
    for permutation_of_all_chars_except_last in permutations_of_all_chars_except_last:
        for position in range(len(all_chars_except_last) + 1):
            permutation = (
                permutation_of_all_chars_except_last[:position]
                + last_char
                + permutation_of_all_chars_except_last[position:]
            )
            permutations.add(permutation)

    return permutations
```

Bonus

How does the problem change if the string can have duplicate characters?

What if we wanted to bring down the time and/or space costs?

What We Learned

This is one where *playing with a sample input* is huge. Sometimes it helps to think of algorithm design as a two-part process: *first* figure out how you would solve the problem "by hand," as though the input was a stack of paper on a desk in front of you. *Then* translate that process into code.