Stack Over-Whoa!

The previous exercise ended with a *stack overflow*, which is a reminder that recursion has costs that iteration doesn’t. We saw in the first exercise that **every recursive call spends time on the call stack.**

Put enough function calls on the call stack, and eventually there’s no room left.

Even when there is room for any reasonable input, recursive functions tend to be at least a little less efficient than comparable iterative solutions because of the call stack.

The beauty of recursion is how it can reduce complex problems into an elegant solution of only a few lines of code. Recursion forces us to distill a task into its smallest piece, the base case, and the smallest step to get there, the recursive step.

Let’s compare two solutions to a single problem: producing a power set. A power set is a list of all subsets of the values in a list.

**This is a really tough algorithm. Don’t be discouraged!**

power\_set(['a', 'b', 'c'])  
# [  
#   ['a', 'b', 'c'],   
#   ['a', 'b'],   
#   ['a', 'c'],   
#   ['a'],   
#   ['b', 'c'],   
#   ['b'],   
#   ['c'],   
#   []  
# ]

Phew! That’s a lot of lists! Our input length was 3, and the list returned had a length of 8.

Producing subsets **requires** a runtime of at least O(2^N), we’ll never do better than that because a set of N elements creates a power set of 2^N elements.

Binary, [a number system of base 2](https://en.wikipedia.org/wiki/Binary_number#Binary_counting), can represent 2^N numbers for N binary digits. For example:

# 1 binary digit, 2 numbers  
# 0 in binary  
0  
# 1 in binary  
1  
   
# 2 binary digits, 4 numbers  
# 00 => 0  
# 01 => 1  
# 10 => 2  
# 11 => 3

The iterative approach uses this insight for a very clever solution by including an element in the subset if its “binary digit” is 1.

set = ['a', 'b', 'c']  
binary\_number = "101"  
# produces the subset ['a', 'c']  
# 'b' is left out because its binary digit is 0

That process is repeated **for all O(2^N) numbers!**

Here is the complete solution. You’re not expected to understand every line, just take in the level of complexity.

def power\_set(set):  
  power\_set\_size = 2\*\*len(set)  
  result = []  
   
  for bit in range(0, power\_set\_size):  
    sub\_set = []  
    for binary\_digit in range(0, len(set)):  
      if((bit & (1 << binary\_digit)) > 0):  
        sub\_set.append(set[binary\_digit])  
    result.append(sub\_set)  
  return result

Very clever but not very intuitive! Let’s try recursion.

Consider the base case, where the problem has become so simple we can solve it without doing any work.

What’s the simplest power set possible? An empty list!

power\_set([])  
# [[]]

Now the recursive step. We need to progress towards our base case, an empty list, so **we’ll be removing an element from the input.**

Examine the simplest powerset that **isn’t** the base case:

power\_set(['a'])  
# [[], ['a']]

A power set in the recursive step requires:

* all subsets which contain the element
  + in this case "a"
* all subsets which don’t contain the element
  + in this case [].

With the recursive approach, we’re able to articulate the problem **in terms of itself.** No need to bring in a whole number system to find the solution!

Here’s the recursive solution in its entirety:

def power\_set(my\_list):  
  if len(my\_list) == 0:  
    return [[]]  
  power\_set\_without\_first = power\_set(my\_list[1:])  
  with\_first = [ [my\_list[0]] + rest for rest in power\_set\_without\_first ]  
  return with\_first + power\_set\_without\_first

Neither of these solutions is simple, **this is a complicated algorithm**, but the recursive solution is almost half the code and more directly conveys what this algorithm does.

Give yourself a pat on the back for making it through a tough exercise!

**Instructions**

Run the code to see subsets of universities.

Try adding your own school.

See how large you can make the input list before these computations become impossibly slow…

O(2^N) runtime is no joke!

def power\_set(my\_list):

    # base case: an empty list

    if len(my\_list) == 0:

        return [[]]

    # recursive step: subsets without first element

    power\_set\_without\_first = power\_set(my\_list[1:])

    # subsets with first element

    with\_first = [ [my\_list[0]] + rest for rest in power\_set\_without\_first ]

    # return combination of the two

    return with\_first + power\_set\_without\_first

universities = ['MIT', 'UCLA', 'Stanford', 'NYU']

power\_set\_of\_universities = power\_set(universities)

for set in power\_set\_of\_universities:

  print(set)