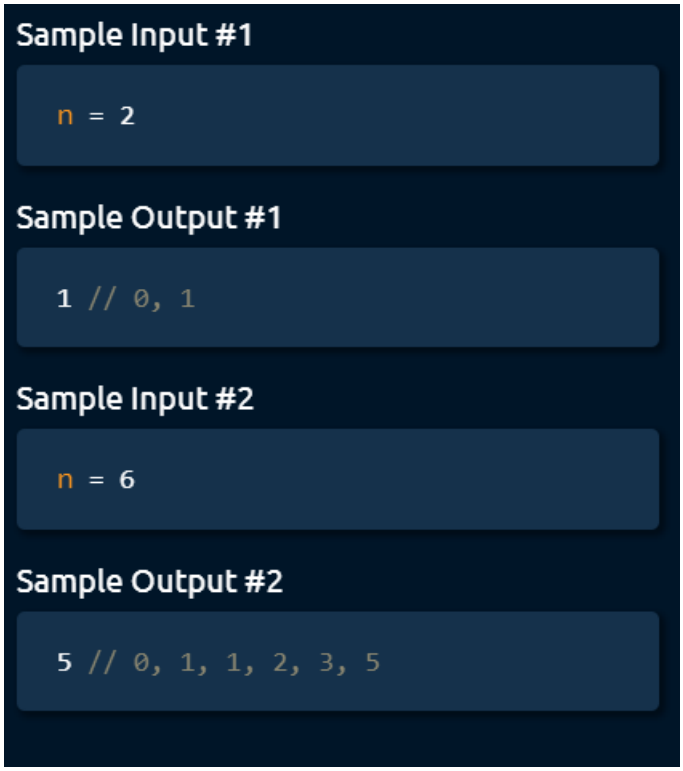
Topic - Recursion

Easy – Difficulty: Nth Fibonacci

Problem Statement:

The Fibonacci sequence is defined as follows: the first number of the sequence is 0, the second number is 1 and the nth number is the sum of the (n - 1)th and (n - 2)th numbers. Write a function that takes in an integer n and returns the nth Fibonacci number.

Important note: the Fibonacci sequence is often defined with its first two numbers as F0 = 0 and F1= 1. For the purpose of this question, the first Fibonacci number is F0; therefore, getNthFib(1) is equal to F0, getNthFib(2) is equal to F1, etc..



Solution:

```python
# O(2^n) time | O(n) space
def getNthFib(n):
    if n == 2:
        return 1
    elif n == 1:
        return 0
    else:
        return getNthFib(n - 1) + getNthFib(n - 2)
```

Fibonacci Hint :

The formula to generate the nth Fibonacci number can be written as follows: F(n) = F(n - 1) + F(n - 2). Think of the case(s) for which this formula doesn't apply (the base case(s)) and try to implement a simple recursive algorithm to find the nth Fibonacci number with this formula.

Easy – Difficulty: – Product Sum

Problem Statement:

Write a function that takes in a "special" array and returns its product sum.

A "special" array is a non-empty array that contains either integers or other "special" arrays. The product sum of a "special" array is the sum of its elements, where "special" arrays inside it are summed themselves and then multiplied by their level of depth.

The depth of a "special" array is how far nested it is. For instance, the depth of []is 1; the depth of the inner array in [[]] is 2; the depth of the innermost array in [[[]]] is 3.

Therefore, the product sum of [x, y] is x + y; the product sum of [x, [y, z]] is x + 2 * (y + z); the product sum of [x, [y, [z]]] is x + 2 * (y + 3z).

Sample Input

```
array = [5, 2, [7, -1], 3, [6, [-13, 8], 4]]
```

Sample Output

```
12 // calculated as: 5 + 2 + 2 * (7 - 1) + 3 + 2 * (6 + 3 * (-13 + 8) + 4)
```

Solution:

```python
# O(n) time | O(d) space - where n is the total number of elements in the array,
# including sub-elements, and d is the greatest depth of "special" arrays in the array
def productSum(array, multiplier=1):
    sum = 0
    for element in array:
        if type(element) is list:
            sum += productSum(element, multiplier + 1)
        else:
            sum += element
    return sum * multiplier
```

Product Sum Hint : Try to use Recursion & Initialize the product sum of the "special" array to 0. Then, iterate through all of the array's elements; if you come across a number, add it to the product sum; if you come across another "special" array, recursively call the productSum function on it and add the returned value to the product sum. How will you handle multiplying the product sums at a given level of depth?

Medium – Difficulty: – Permutations

Problem Statement:

Write a function that takes in an array of unique integers and returns an

array of all permutations of those integers in no particular order.

If the input array is empty, the function should return an empty array.

Sample Input

```
array = [1, 2, 3]
```

Sample Output

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Solution:

a.

```python
# Upper Bound: O(n^2*n!) time | O(n*n!) space
# Roughly: O(n*n!) time | O(n*n!) space
def getPermutations(array):
    permutations = []
    permutationsHelper(array, [], permutations)
    return permutations


def permutationsHelper(array, currentPermutation, permutations):
    if not len(array) and len(currentPermutation):
        permutations.append(currentPermutation)
    else:
        for i in range(len(array)):
            newArray = array[:i] + array[i + 1 :]
            newPermutation = currentPermutation + [array[i]]
            permutationsHelper(newArray, newPermutation, permutations)
```

b.

```python
# O(n*n!) time | O(n*n!) space
def getPermutations(array):
    permutations = []
    permutationsHelper(0, array, permutations)
    return permutations


def permutationsHelper(i, array, permutations):
    if i == len(array) - 1:
        permutations.append(array[:])
    else:
        for j in range(i, len(array)):
            swap(array, i, j)
            permutationsHelper(i + 1, array, permutations)
            swap(array, i, j)


def swap(array, i, j):
    array[i], array[j] = array[j], array[i]
```

Permutation Hint : A permutation is defined as a way in which a set of things can be ordered. Thus, for the list [1, 2, 3], there exist some permutations starting with 1, some starting with 2, and some starting with 3. For the permutations starting with 1, there will be a permutation where 2 is the second number and one where 3 is the second number. For permutations starting with 3, there will be a permutation where 1 is the second number and one where 2 is the second number. The idea is that, in order to construct a permutation, we can pick a random number from our list to be the first number, then we can pick a random number from the remaining list (without the first number) to be the second number, then we can pick a random number from the remaining list (without the first and second numbers) to be the third number, and

we can repeat the process until we exhaust all of our list of numbers. At that point, we will have constructed a valid permutation. How can we implement this construction algorithmically, without picking numbers at random?