Permutations of an Array

LeetCode #46

General Theory

- Uses a **slate** as a global-variable and inserts the Tree-Node's result into the slate on every Manager's call.

Further, this slate uses a Dynamically adjusted **divider** within the slate to logically separate 2 individual pieces of state; 1. **slate**, 2. Elements not within the **slate**.

- Uses a loop at each node, therefore the conceptual understanding is NOT a tree, just a callstack.
  - More on this in the **Take Away** section.

Control Flow

1. The **base case** is to check if the **slate** is full. If so, then we've reached a leaf-node within the tree and we've completed the task of generating an answer. Append contents of the slate to the results.
2. Each **manager** / **node** will have the job of iterating through the sub-section of the numbers that are to the *right* of the slate's divider.
   - This loops will run n-l times (n = # of elements, l = Tree-Level).
3. Within this loop, we perform 3 operations.
4. *First*: We swap the i'th element from our current scope'd loop, with the value adjacent to the slate's current divider position. See example below

In [ ]:
```
# Control Flow.4 Example
slate = [5, 4, 3, 2, 1]
#             ^   i = 1   ---->   divider position
swap(slate, i, pos) # i = 1, pos = 1
slate = [5, 4, 3, 2, 1]
```

Control Flow (cont'd)

4. [cont'd] - Theoretically we're trying to invision that the divider sits between index 1 & 2, so we say it's at index 1. `pos` is describing the divider's current position. `0 to pos - 1` = All elements **within** the slate. And `pos to len(array)` = All elements outside the slate.
5. After we've called recursively and eventually arrived at a leaf node, we'll retrace to a previous call, and undo the swap we made. Then we'll iterate through the Node's loop,

and swap again.

6. On a high-level, we should think of separation of concerns as follows
    - Tree-Level-Work:
        - Decide the size of the slate we're filling by fixing the slate-divider location.
        - Produce less work (smaller loop length) the deeper into the tree we go.
    - Node-Level-Work:
        - Swaps elements with the divider.
        - Number of elements to be swapped = n-l. n = size of input. l = tree-level.

In [ ]:
```
def swap(a, l, r):
    a[l], a[r] = a[r], a[l]

def get_set_permutations(slate, pos=0, results=[]):
    if pos >= len(slate):
        results.append(slate.copy())
        return
    for i in range(pos, len(slate)):
        swap(slate, i, pos)
        get_set_permutations(slate, pos + 1, results)
        swap(slate, pos, i)
    return results

result = get_set_permutations([1, 2, 3])
answer = [
    [1, 2, 3],
    [1, 3, 2],
    [2, 1, 3],
    [2, 3, 1],
    [3, 2, 1],
    [3, 1, 2]]
assert result == answer, 'should be equal'
```

Tests

In [ ]:
```
import unittest
from random import randint as rint

class PermutationTests(unittest.TestCase):
    def __init__(self, *args, **kwargs):
        super(PermutationTests, self).__init__(*args, **kwargs)

    def test_result_length(self):
        test_cases = self.get_test_cases(hi=0, lo=10, case_len=3, cases=1)
        for tc in test_cases:
            result = get_set_permutations(tc, 0, [])
            self.assertEqual(len(result), 6)
            self.assertIn(tc, result)

    def test_result_1(self):
        test_case = [1, 2, 3]
        result = get_set_permutations(test_case, 0, [])
```

```
        self.assertIn([1, 2, 3], result)
        self.assertIn([3, 1, 2], result)
        self.assertIn([3, 2, 1], result)
        self.assertIn([2, 3, 1], result)
        self.assertNotIn([3, 3, 2], result)
        self.assertNotIn([3, 2, 2], result)
        self.assertNotIn([1, 2, 1], result)
        self.assertEqual(len(result), 6)

    @staticmethod
    def get_test_cases(hi, lo, case_len, cases):
        test_cases = [
            [rint(hi, lo) for i in range(0, case_len)]
            for _ in range(0, cases)]
        return test_cases

unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

TakeAways

1. This problem is concerned with moving a series of elements to a particular location. It accomplishes this by using a loop at every recursive node. Because the quantity of these elements to be moved is dynamic, we can assign a thought-pattern to this particular type of problem:
    - The Recursive Mental-Model will **NOT be a tree** but rather a **uni-directional stack** of calls.
2. Each call, will perform a **loops-worth of work**. This will generally mean, we're still doing some **Quadratic** order of work so pay close attention to Reccurence Relation to verify.
3. **Intuition** for *Recursive-Tree's* : They are a product of deciding between a *forks-in-the-road* which forces us to decide between 2 choices;
    - > "should i take it or leave it?",
    - > "should i change it or not change it?".
4. **Intuition** for *Recursive-Stacks* : They are a product of a *repetition-of-work*. Doing the same-thing over and over again, but the work-to-be-done is dynamically changing over time. I should think about a loop + recursive stack whenever similar situations arise I should think of a loop + recursion.
    - > I must repeatedly move a pile of rocks, from point A to point B to point C..
    - > I must repeatedly move N piles of rocks to M different locations.