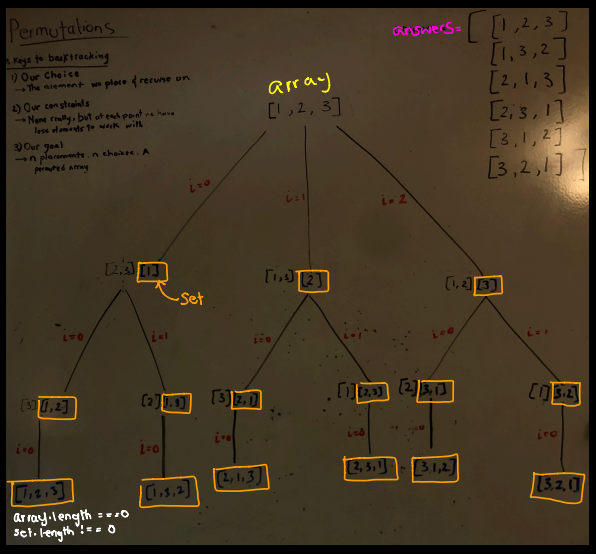
Input: An array of unique integers
array = [1, 2, 3]

Output: An array of all permutations of those integers in no particular order. If input array is empty, return empty array
[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

```
// O(n*n!) time | O(n*n!) space
function getPermutations(array) {
  const answers = [];
  permute(array, [], answers);
  return answers;
}

function permute(array, set, answers) {
  if (array.length === 0 && set.length !== 0) answers.push([...set]);

  for (let i = 0; i < array.length; i++) {
    const newArray = array.filter((num, index) => index !== i);
    set.push(array[i]);
    permute(newArray, set, answers);
    set.pop();
  }
}
```



Idea: Backtracking
1) Base case: if the array is empty and our set is non empty, then we append our current set into our answers array.
2) Otherwise we iterate through the array, and at each index we:
   Create a new array with all values except for the value at index i of the original array
   We push the value at index i of the original array
   We then recurse on the new array
3) As we work our way up, we pop off the value we added

Time: O(n * n!) (where n is the # of elements in the array) since we have n! permutations and at each one we do linear amount of work, n, to copy the array into our new array

Space: O(n * n!) since we have an array of length n at each permutation