# 47. Permutations II

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations* ***in any order***.

**Example 1:**

```
Input: nums = [1,1,2]
Output:
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

**Example 2:**

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

**Constraints:**

- `1 <= nums.length <= 8`
- `-10 <= nums[i] <= 10`

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        freq = Counter(nums)
        res = []
        self.permuteUtil(nums,res,0,freq,[])
        return res


    def permuteUtil(self,nums,res,te,freq,asf):
        if te==len(nums):
            temp = asf[:]
            res.append(temp)
            return

        for key in freq.keys():
            if freq[key]>0:
                freq[key] = freq[key]-1
```

```
                self.permuteUtil(nums,res,te+1,freq,asf+[key])
                freq[key] = freq[key]+1
```

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        res = []
        visited = [0]*len(nums)
        self.permuteUtil(nums,res,[],0,visited)
        return res




    def permuteUtil(self,nums,res,asf,te,visited):
        if te==len(nums):
            temp = asf[:]
            if temp not in res:
                res.append(temp)
            return
        for i in range(len(nums)):
            if visited[i]==0:
                visited[i] = 1
                self.permuteUtil(nums,res,asf+[nums[i]],te+1,visited)
                visited[i]=0
```

## Solution

---

### Overview

As the name of the problem suggests, this problem is an extension of the Permutation problem. The problem is different from the previous permutation problem on the condition that the input array can contain ***duplicates***.

The key to solve the problem is still the ***backtracking*** algorithm. However, we need some adaptation to ensure that the *enumerated* solutions generated from our backtracking exploration do not have any duplicates.

> As a reminder, **backtracking** is a general algorithm for finding all (or some) solutions to some problems with constraints. It incrementally builds candidates to the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly lead to a solution.

In this article, we will present a yet another backtracking solution to solve the problem.

---

### Approach 1: Backtracking with Groups of Numbers

#### Intuition

First of all, let us review the general idea of permutation with an example.

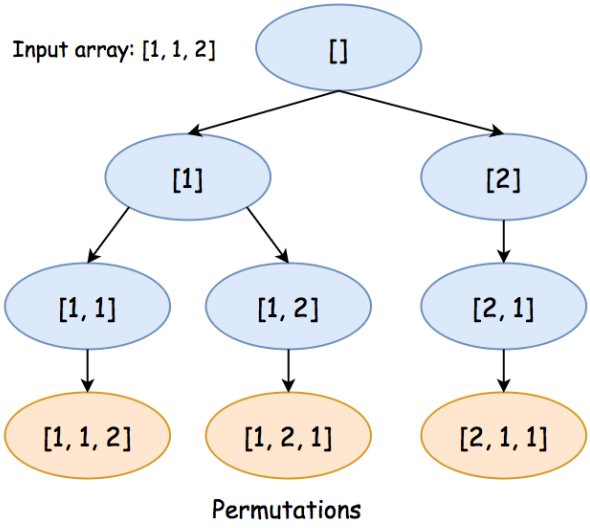
Given the input array `[1, 1, 2]`, to generate a permutation of the array, we could follow the *Depth-First Search* (DFS) approach, or more precisely the backtracking technique as one will see later.

> The idea is that we pick the numbers one by one. For a permutation of length NN, we would then need NN stages to generate a valid permutation. At each stage, we need to pick one number into the permutation, out of the remaining available numbers. Later at the same stage, we will try out all available choices. By trying out, we progressively build up candidates to the solution, and revert each choice with another alternative until there is no more choice.

Let us walk through the example with paper and pencil, as follows:

- Given the input of `[1, 1, 2]`, at the first stage, we have 2 choices to pick a number as the first number in the final permutation, *i.e.* `1` and `2`. Suppose that we pick the number `1`, now the remaining numbers would become `[1, 2]`. **Note:** The reason that we have only 2 choices instead of 3, is that there is a duplicate in the given input. Picking any of the duplicate numbers as the first number of the permutation would lead us to the same permutation at the end. Should the numbers in the array be all unique, we would then have the same number of choices as the length of the array.
- At the second stage, we now then have again 2 choices, *i.e.* `[1, 2]`. Let us pick again the number `1`, which leaves us the only remaining number `2`.
- Now at the third stage, we have only one candidate number left, *i.e.* `[2]`. We then pick the last remaining number, which leads to a final permutation sequence of `[1, 1, 2]`.
- Moreover, we need to ***revisit*** each of the above stages, and make a different choice in order to try out all possibilities. The reversion of the choices is what we call ***backtracking***.

We illustrate all potential exploration in the following graph where each node represents a choice at a specific stage:




> A key insight to avoid generating any ***redundant*** permutation is that at each step rather than viewing each number as a candidate, we consider each ***unique*** number as the true candidate. For instance, at the very beginning, given in the input of `[1, 1, 2]`, we have only two true candidates instead of three.

**Algorithm**

Given the above insight, in order to find out all the unique numbers at each stage, we can build a ***hash table*** (denoted as `counter`), with each unique number as the key and its occurrence as the corresponding value.

To implement the algorithm, first we define a function called `backtrack(comb, counter)` which generates all permutations, starting from the current combination (`comb`) and the remaining numbers (`counter`).

Once the function is implemented, it suffices to invoke the function with the initial empty combination and the hash table we built out of the input array, to solve the problem.

Here are some sample implementations.

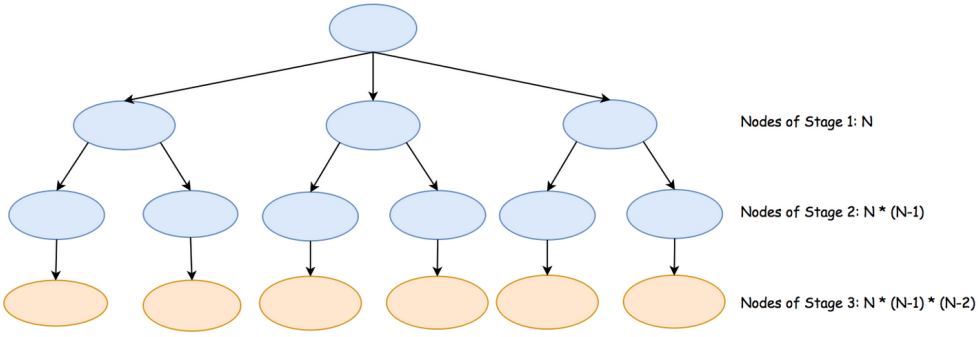
**Note:** for a backtracking algorithm, usually there are some explorations that would lead to a *dead end*, and we have to abandon those explorations in the middle.

However, due to the specificity of this problem and our exploration strategy, each exploration will result in a valid permutation, *i.e.* none of the efforts is in vain. This insight would prove to be useful in the following complexity analysis.

**Complexity Analysis**

Let NN be the length of the input array. Hence, the number of permutations would be at maximum N!N!, *i.e.* N \cdot (N-1) \cdot (N-2) ... 1N·(N−1)·(N−2)...1, when each number in the array is unique.

- Time Complexity: \mathcal{O}\big(\sum_{k = 1}^{N}{P(N, k)}\big)O(∑k=1NP(N,k)) where P(N, k) = \frac{N!}{(N - k)!} = N (N - 1) ... (N - k + 1)P(N,k)=(N−k)!N!=N(N−1)...(N−k+1) is so-called [k-permutations_of_N or partial permutation](#).
    - As one can see in the exploration graph we have shown earlier, the execution of the backtracking algorithm will unfold itself as a tree, where each node is an invocation of the recursive function `backtrack(comb, counter)`. The total number of steps to complete the exploration is *exactly* the number of nodes in the tree. Therefore, the time complexity of the algorithm is linked directly with the size of the tree.
    - It now boils down to estimating the number of nodes in the tree. As we know now, each level of the tree corresponds to a specific *stage* of the exploration. At each stage, the number of candidates to explore is **bounded**. For instance, at the first stage, *at most* we would have NN candidates to explore, *i.e.* the number of nodes at this level would be NN. Moving on to the next stage, for each of the nodes in the first stage, we would have N-1N−1 child nodes. Therefore, the number of nodes at this stage would be N \cdot (N-1)N·(N−1). So on and so forwards.




    - By summing up all the nodes across the stages, we would then obtain the total number of nodes as \sum_{k = 1}^{N}{P(N, k)}∑k=1NP(N,k) where P(N, k) = \frac{N!}{(N - k)!} = N (N - 1) ... (N - k + 1)P(N,k)=(N−k)!N!=N(N−1)...(N−k+1). As a result, the exact time complexity of the algorithm is \mathcal{O}\big(\sum_{k = 1}^{N}{P(N, k)}\big)O(∑k=1NP(N,k)).
    - The above complexity might appear a bit too abstract to comprehend. Here we could provide another ***loose upper bound*** on the complexity.
    - It takes NN steps to generate a single permutation. Since there are in total N!N! possible permutations, at most it would take us N \cdot N!N·N! steps to generate all permutations, simply assuming that there is no overlapping effort (which is not true).
- Space Complexity: \mathcal{O}(N)O(N)
    - First of all, we build a hash table out of the input numbers. In the worst case where each number is unique, we would need \mathcal{O}(N)O(N) space for the table.
    - Since we applied recursion in the algorithm which consumes some extra space in the function call stack, we would need another \mathcal{O}(N)O(N) space for the recursion.
    - During the exploration, we keep a candidate of permutation along the way, which takes yet another \mathcal{O}(N)O(N).
    - To sum up, the total space complexity would be \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(N)+O(N)+O(N)=O(N).

- **Note**, we did not take into account the space needed to hold the results. Otherwise, the space complexity would become \mathcal{O}(N \cdot N!)O(N·N!).