https://leetcode.com/problems/permutations/

Given an array nums of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

**Example 1:**

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

**Example 2:**

Input: nums = [0,1]

Output: [[0,1],[1,0]]

**Example 3:**

Input: nums = [1]

Output: [[1]]

**Constraints:**

- 1 <= nums.length <= 6
- -10 <= nums[i] <= 10
- All the integers of nums are **unique**.

---

**Attempt 1: 2022-10-18**

**Solution 1: Backtracking style 1 (10min, no boolean[] visited array required just use ArrayList.contains() to identify if current number been used or not before is because one condition:** All the integers of nums are **unique**.**)**

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        helper(nums, result, new ArrayList<Integer>(), 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {
        if(tmp.size() == nums.length) {
```

```
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        for(int i = index; i < nums.length; i++) {
            // No boolean[] visited array required just use ArrayList.contains() to
            // identify if current number been used or not before is because one
condition:
            // All the integers of nums are unique.
            if(tmp.contains(nums[i])) {
                continue;
            }
            tmp.add(nums[i]);
            // Differ than L77.Combinations statement which pass local variable 'i'
            // plus 1('i + 1') into next recursion
            helper(nums, result, tmp, index);
            tmp.remove(tmp.size() - 1);
        }
    }
}

-------------------------------------------------------------------------------------
------------------
Remove redundant 'index' from parameter

class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // No need 'index = 0' as recursion parameter anymore, since
        // when recursion happens in "permutation", it always have to
        // start with first element to scan the whole input again, so
        // the for loop in recursion always start with i = 0, no need
        // receive value passed in from parameter 'index = 0'
        // That's the major difference than "combination" which requires
        // next recursion level always move 1 index ahead of current index
        //helper(nums, result, new ArrayList<Integer>(), 0);
        helper(nums, result, new ArrayList<Integer>());
        return result;
    }

```

```java
    //private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {
    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp) {
        if(tmp.size() == nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        //for(int i = index; i < nums.length; i++) {
        for(int i = 0; i < nums.length; i++) {
            // No boolean[] visited array required just use ArrayList.contains() to
            // identify if current number been used or not before is because one condition:
            // All the integers of nums are unique.
            if(tmp.contains(nums[i])) {
                continue;
            }
            tmp.add(nums[i]);
            // Differ than L77.Combinations statement which pass local variable 'i'
            // plus 1('i + 1') into next recursion
            //helper(nums, result, tmp, index);
            helper(nums, result, tmp);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

**Refer to**

https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/121098

The worst-case time complexity is O(n! * n).

For any recursive function, the time complexity is O(branches^depth) * amount of work at each node in the recursive call tree. However, in this case, we have n*(n-1)*(n*2)*(n-3)*...*1 branches at each level = n!, so the total recursive calls is O(n!)

We do n-amount of work in each node of the recursive call tree, (a) the for-loop and (b) at each leaf when we add n elements to an ArrayList. So this is a total of O(n) additional work per node.

Therefore, the upper-bound time complexity is O(n! * n).

**Refer to**

https://leetcode.com/problems/permutations/discuss/1527929/Java-or-TC%3A-O(N*N!)-or-SC%3A-O(N)-or-Recursive-Backtracking-and-Iterative-Solutions

Time Complexity: O(N * N!). Number of permutations = P(N,N) = N!.

Each permutation takes O(N) to construct

T(n) = n*T(n-1) + O(n)

T(n-1) = (n-1)*T(n-2) + O(n-1)

...

T(2) = (2)*T(1) + O(2)

T(1) = O(N) -> To convert the nums array to ArrayList.

Above equations can be added together to get:

T(n) = n + n*(n-1) + n*(n-1)*(n-2) + ... + (n....2) + (n....1) * n

= P(n,1) + P(n,2) + P(n,3) + ... + P(n,n-1) + n*P(n,n)

= (P(n,1) + ... + P(n,n)) + (n-1)*P(n,n)

= Floor(e*n! - 1) + (n-1)*n!

= O(N * N!)

Space Complexity: O(N). Recursion stack.

N = Length of input array.

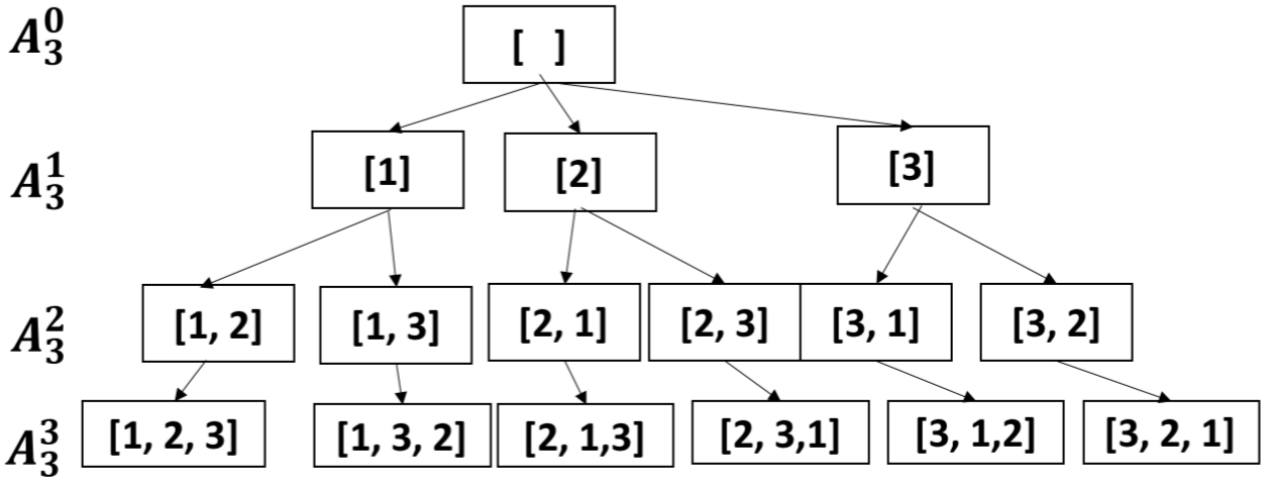
**For Backtracking style 1 Tree Structure Analysis**

Tree Structure Analysis

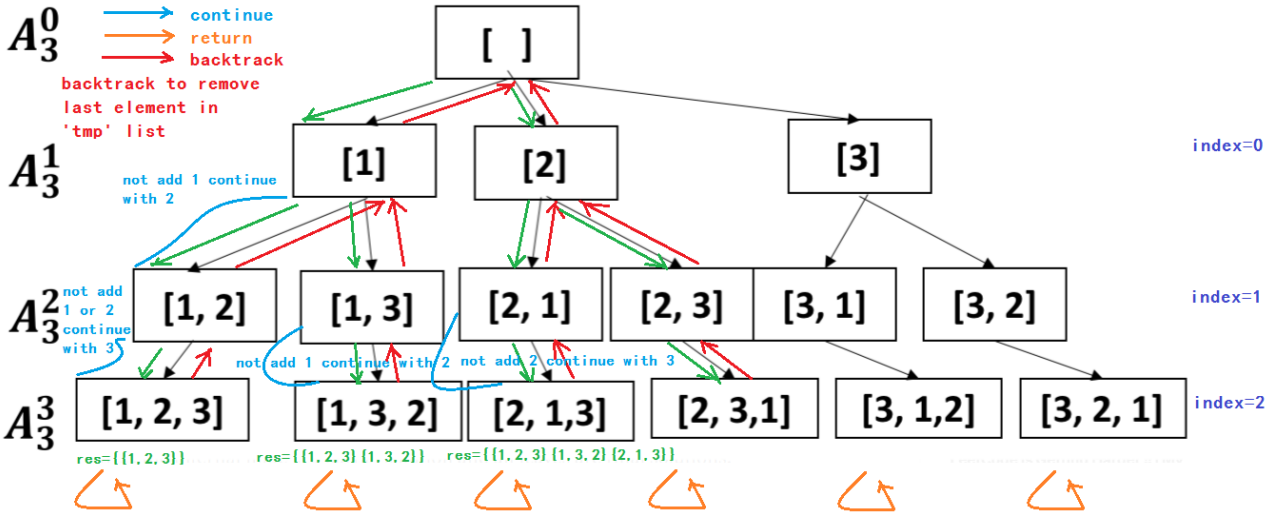
```
e.g.
Input: n = 3, k = 3
Output: {{1,2,3}{1,3,2}{2,1,3}{2,3,1}{3,1,2}{3,2,1}}

                            { }
                    /        |        \
                  {1}        {2}       {3}
                 /   \      /   \      /   \
             {1,2} {1,3} {2,1} {2,3} {3,1} {3,2}
               |     |     |     |     |     |
          {1,2,3}{1,3,2}{2,1,3}{2,3,1}{3,1,2}{3,2,1}
```

**Refer to**

https://medium.com/algorithms-and-leetcode/backtracking-e001561b9f28




**The traversal and backtrack process is below**




**Compare to L77.Combinations Solution 1: Backtracking style 1, the critical difference is L46. Permutations Solution 1: Backtracking style 1 pass 'index(=0)' or no need to pass 'index' and always for loop start from 0 into next recursion level, whereas L77. Combinations pass local variable 'i' plus 1 as 'i + 1' into next recursion level**

```
// L77. Combination pass local variable 'i' plus 1 as 'i + 1' into next recursion level instead of passing 'index'
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        int[] candidates = new int[n + 1];
        for(int i = 1; i <= n; i++) {
            candidates[i] = i;
        }
        // Since range [1,n], start index not 0 but 1
```

```
        helper(candidates, result, new ArrayList<Integer>(), k, 1);
        return result;
    }

    private void helper(int[] candidates, List<List<Integer>> result, List<Integer> tmp, int k, int index) {
        if(tmp.size() == k) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        for(int i = index; i < candidates.length; i++) {
            tmp.add(candidates[i]);
            helper(candidates, result, tmp, k, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

---

**Solution 2: Backtracking style 2 (10min, instead of contains() method check, use boolean[] visited array)**

**Note: Like mentioned in Solution 1, not use boolean[] visited array, just use tmp.contains() to filter out already added element into tmp list is not robust since its totally based on given condition as "**All the integers of nums are **unique**.**", the style 1 and style 2 will have big deviation when comes to a duplicate existing input such as {1,1,2}, for style 1, since tmp.contains() logic exist, after 1st '1' on tmp, it won't proceed for 2nd '1', the final result of all permutations is empty list {}, but for style 2, since boolean[] visited used, different indexed element will be individually judged, not based on previous element condition, so final result will be [[1, 1, 2], [1, 2, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 1, 1]], but its also something wrong, since duplicate not handling well, which resolved by L47.**

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        helper(nums, result, new ArrayList<Integer>(), 0, new boolean[nums.length]);
        return result;
    }
```

```
7
8       private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index, boolean[] visited) {
9           if(tmp.size() == nums.length) {
10              result.add(new ArrayList<Integer>(tmp));
11              return;
12          }
13          for(int i = index; i < nums.length; i++) {
14              if(visited[i]) {
15                  continue;
16              }
17              tmp.add(nums[i]);
18              visited[i] = true;
19              helper(nums, result, tmp, index, visited);
20              visited[i] = false;
21              tmp.remove(tmp.size() - 1);
22          }
23      }
24  }
25
26  ------------------------------------------------------------------------------------------
27  Remove redundant 'index' from parameter
28
29  class Solution {
30      public List<List<Integer>> permute(int[] nums) {
31          List<List<Integer>> result = new ArrayList<List<Integer>>();
32          //helper(nums, result, new ArrayList<Integer>(), 0, new boolean[nums.length]);
33          helper(nums, result, new ArrayList<Integer>(), new boolean[nums.length]);
34          return result;
35      }
36
37      //private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index, boolean[] visited) {
38      private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, boolean[] visited) {
39          if(tmp.size() == nums.length) {
40              result.add(new ArrayList<Integer>(tmp));
41              return;
42          }
43          //for(int i = index; i < nums.length; i++) {
```

```
        for(int i = 0; i < nums.length; i++) {
            if(visited[i]) {
                continue;
            }
            tmp.add(nums[i]);
            visited[i] = true;
            //helper(nums, result, tmp, index, visited);
            helper(nums, result, tmp, visited);
            visited[i] = false;
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

**Refer to**

https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/121098

The worst-case time complexity is O(n! * n).

For any recursive function, the time complexity is O(branches^depth) * amount of work at each node in the recursive call tree. However, in this case, we have n*(n-1)*(n*2)*(n-3)*...*1 branches at each level = n!, so the total recursive calls is O(n!)

We do n-amount of work in each node of the recursive call tree, (a) the for-loop and (b) at each leaf when we add n elements to an ArrayList. So this is a total of O(n) additional work per node.

Therefore, the upper-bound time complexity is O(n! * n).

**Refer to**

https://leetcode.com/problems/permutations/discuss/1527929/Java-or-TC%3A-O(N*N!)-or-SC%3A-O(N)-or-Recursive-Backtracking-and-Iterative-Solutions

Time Complexity: O(N * N!). Number of permutations = P(N,N) = N!.

Each permutation takes O(N) to construct

T(n) = n*T(n-1) + O(n)

T(n-1) = (n-1)*T(n-2) + O(n-1)

...

T(2) = (2)*T(1) + O(2)

T(1) = O(N) -> To convert the nums array to ArrayList.

Above equations can be added together to get:

T(n) = n + n*(n-1) + n*(n-1)*(n-2) + ... + (n....2) + (n....1) * n

= P(n,1) + P(n,2) + P(n,3) + ... + P(n,n-1) + n*P(n,n)

= (P(n,1) + ... + P(n,n)) + (n-1)*P(n,n)

= Floor(e*n! - 1) + (n-1)*n!

= O(N * N!)

Space Complexity: O(N). Recursion stack.

N = Length of input array.

**Refer to**

https://leetcode.com/problems/permutations/discuss/179932/Beats-100-Java-with-Explanations

**Thought**

We think about a searching tree when we apply Backtracking.

```
e.g.[1, 2, 3]
    1 -2 -3
      -3 -2

    2 -1 -3
      -3 -1

    3 -1 -2
      -2 -1
```

If we exhausted the current branch, currResult.size() == nums.length, we will backtrack. To make sure each element is used once, we establish boolean[] used.

**Code**

```java
public List<List<Integer>> permute(int[] nums) {

        if (nums == null || nums.length == 0)
            return new ArrayList<>();

        List<List<Integer>> finalResult = new ArrayList<>();
        permuteRecur(nums, finalResult, new ArrayList<>(), new boolean[nums.length]);
        return finalResult;
    }
```

```
10
11      private void permuteRecur(int[] nums, List<List<Integer>> finalResult,
    List<Integer> currResult, boolean[] used) {
12
13          if (currResult.size() == nums.length) {
14              finalResult.add(new ArrayList<>(currResult));
15              return;
16          }
17
18          for (int i = 0; i < nums.length; i++) {
19              if (used[i])
20                  continue;
21              currResult.add(nums[i]);
22              used[i] = true;
23              permuteRecur(nums, finalResult, currResult, used);
24              used[i] = false;
25              currResult.remove(currResult.size() - 1);
26          }
27      }
```

---

**No "Not pick" and "Pick" branch available for this problem yet**

**Because in L46. we have to use all numbers in the given array, not like L77. we just pick k out of n numbers, so there is no chance for a number in L46 to 'Not pick', hence no "Not pick" and "Pick" strategy here**

---

**Mathematical proof that time complexity is O(e * n!) NOT O(n * n!)**

https://leetcode.com/problems/permutations/discuss/2074177/Mathematical-proof-that-time-complexity-is-OI have seen a lot of answers here that simply state the time complexity is O(n*n!) but the justification isn't too well explained. Here I show a better approximation for the time complexity is actually O(e*n!).

First we must visualize the recursion tree (see other answers for recursive solution), the tree below shows the recursion for n=4. On the first layer of the tree we have n possible options to choose from, so we make n function calls and have n nodes in our tree. Now we have n partial permutations built up so far and have n-1 numbers to choose from, so the next layer in our tree will have n*(n-1) nodes. The layer after this will have n*(n-1)*(n-2) nodes and so on and so forth. Until we have n! leaf nodes

at the bottom of our tree. At this point it is obvious to see O(n*n!) is an over estimate for the time complexity of this algorithm, as it implies each layer (there are n in total) has n! nodes.

We know the time complexity of a recursive algorithm is the number of nodes in its recursion tree multiplied by the cost of computation at each node. At each node in our tree we either call the dfs function recursively (non-leaf nodes) or add to the results array, both of these operations are O(1), hence the time complexity is equal to the number of nodes in the recursion tree.

Now for the magic, if we sum up the nodes in each layer of the recursion tree we get to the expression:

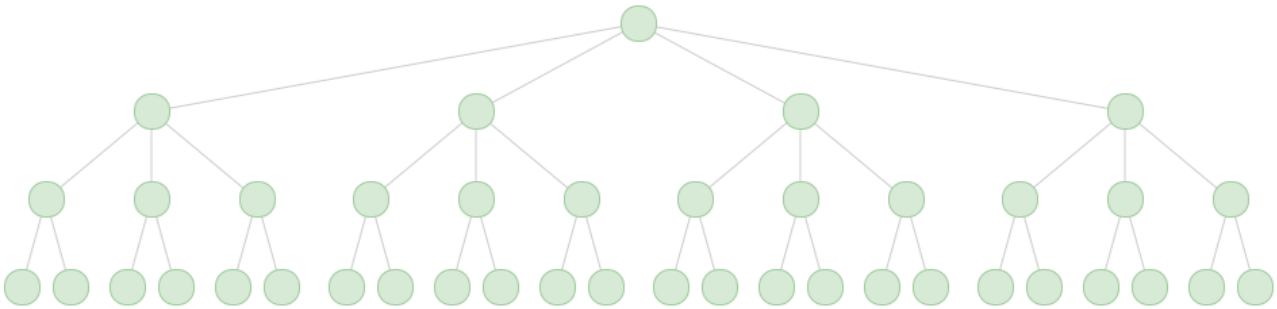
O(n) = 1 + n + n*(n-1) + n*(n-1)*(n-2) + ... + n!

If we reverse the order of terms in this series and factor out n! we get:

O(n) = n!(1/1! + 1/2! + 1/3! + ... + 1/n!)

Notice the second term is the series representation of e, so we have:

O(n) = e * n!



Here are some calculations for n = 1-10, of actual nodes in recursion tree (calculating the first summation expression in a while loop) vs.

e*n! vs. n*n!:

| n | actual | e*n! | n*n! |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 2 | 4 | 5 | 4 |
| 3 | 15 | 16 | 18 |
| 4 | 64 | 65 | 96 |
| 5 | 325 | 326 | 600 |
| 6 | 1956 | 1957 | 4320 |
| 7 | 13699 | 13700 | 35280 |
| 8 | 109600 | 109601 | 322560 |
| 9 | 986409 | 986410 | 3265920 |
| 10 | 9864100 | 9864101 | 36288000 |

**L77. Combinations 和 L46. Permutations & L47. Permutations II应用回溯法最大的区别就是L77真的需要在每一层递归的传递的时候都需要向前（从左向右视为前）移动坐标一位，因为是求Combinations，不能折返回已经用过的坐标的数（包括当前坐标），所有的坐标和坐标所代表的数值只能最多使用一次或者不用，但L46，L47不一样，因为是Permutations，所以可以折返回已经用过的所有坐标再次选取，所以每一层递归的时候传递进去的坐标都必须从0开始，重新从头开始选取数值，除了当前选取集合中已经包含的坐标之外的所有坐标都可以再次选取（例如，从{1,2,3,4}中选2个，在当前递归过程中在到达“底”之前已经形成的集合已经包含了{2,3}，除了数值4可选之外，还可以回头选择坐标为0的数值1，但是坐标位于1,2的数值2,3都不能再选了，因为已经在当前集合中出现过了）**

**不同于L77, 本题没有优化为Memoization和DP的意义, 所以此处只保留了一种Recursion的方法**
**Additional solution: Recursion (60min, insert new digit into all potential intervals on existing combination after each recursion)**

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        return helper(nums, nums.length - 1);
    }

    // index表示当前新增的数字在原数组nums中的位置
    private List<List<Integer>> helper(int[] nums, int index) {
        // 只有一个数字的时候
        if(index == 0) {
            List<List<Integer>> tmp = new ArrayList<>();
            List<Integer> one_num = new ArrayList<>();
            one_num.add(nums[0]);
            tmp.add(one_num);
            return tmp;
        }
        List<List<Integer>> cur_result = helper(nums, index - 1);
        // 遍历每一种情况
        int cur_size = cur_result.size();
        for(int i = 0; i < cur_size; i++) {
            // 在数字的缝隙插入新的数字, j是在已有数字的间隙中插入新数字时的可选位置[0到index]
            // e.g 比如在{1,2}这个组合中插入新数字3会有三个可选坐标[0到2]
            // 插入后的效果, 插入位置0,{3,1,2}, 插入位置1,{1,3,2}, 插入位置2,{1,2,3}
            for(int j = 0; j <= index; j++) {
                List<Integer> comb = new ArrayList<>(cur_result.get(i));
```

```
                comb.add(j, nums[index]);
                // 添加到结果中
                cur_result.add(comb);
            }
        }
        // 由于result此时既保存了之前的结果，和添加完的结果，所以把之前的结果要删除
        for(int i = 0; i < cur_size; i++) {
            cur_result.remove(0);
        }
        return cur_result;
    }
}
```

**Refer to**

https://leetcode.wang/leetCode-46-Permutations.html

# 解法一 插入

这是自己开始想到的一个方法，考虑的思路是，先考虑小问题怎么解决，然后再利用小问题去解决大问题。没错，就是递归的思路。比如说，
如果只有 1 个数字 [ 1 ]，那么很简单，直接返回 [ [ 1 ] ] 就 OK 了。
如果加了 1 个数字 2， [ 1 2 ] 该怎么办呢？我们只需要在上边的情况里，在 1 的空隙，也就是左边右边插入 2 就够了。变成 [ [ **2** 1 ], [ 1 **2** ] ]。
如果再加 1 个数字 3，[ 1 2 3 ] 该怎么办呢？同样的，我们只需要在上边的所有情况里的空隙里插入数字 3 就行啦。例如 [ 2 1 ] 在左边，中间，右边插入 3 ，变成 **3** 2 1，2 **3** 1，2 1 **3**。同理，1 2 在左边，中间，右边插入 3，变成 **3** 1 2，1 **3** 2，1 2 **3**，所以最后的结果就是 [ [ 3 2 1], [ 2 3 1], [ 2 1 3 ], [ 3 1 2 ], [ 1 3 2 ], [ 1 2 3 ] ]。
如果再加数字也是同样的道理，只需要在之前的情况里，数字的空隙插入新的数字就够了。
思路有了，直接看代码吧。

```
public List<List<Integer>> permute(int[] nums) {
    return permute_end(nums,nums.length-1);
}
// end 表示当前新增的数字的位置
private List<List<Integer>> permute_end(int[] nums, int end) {
    // 只有一个数字的时候
    if(end == 0){
```

```
        List<List<Integer>> all = new ArrayList<>();
        List<Integer> temp = new ArrayList<>();
        temp.add(nums[0]);
        all.add(temp);
        return all;
    }
    //得到上次所有的结果
    List<List<Integer>> all_end = permute_end(nums,end-1);
    int current_size = all_end.size();
    //遍历每一种情况
    for (int j = 0; j < current_size; j++) {
        //在数字的缝隙插入新的数字
        for (int k = 0; k <= end; k++) {
            List<Integer> temp = new ArrayList<>(all_end.get(j));
            temp.add(k, nums[end]);
            //添加到结果中
            all_end.add(temp);
        };
    }
    //由于 all_end 此时既保存了之前的结果，和添加完的结果，所以把之前的结果要删除
    for (int j = 0; j < current_size; j++) {
        all_end.remove(0);
    }
    return all_end;
}
```

**Refer to**

L77.Combinations