<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)**

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++) {

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);

}

}

}

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}

/ \ / \ / \

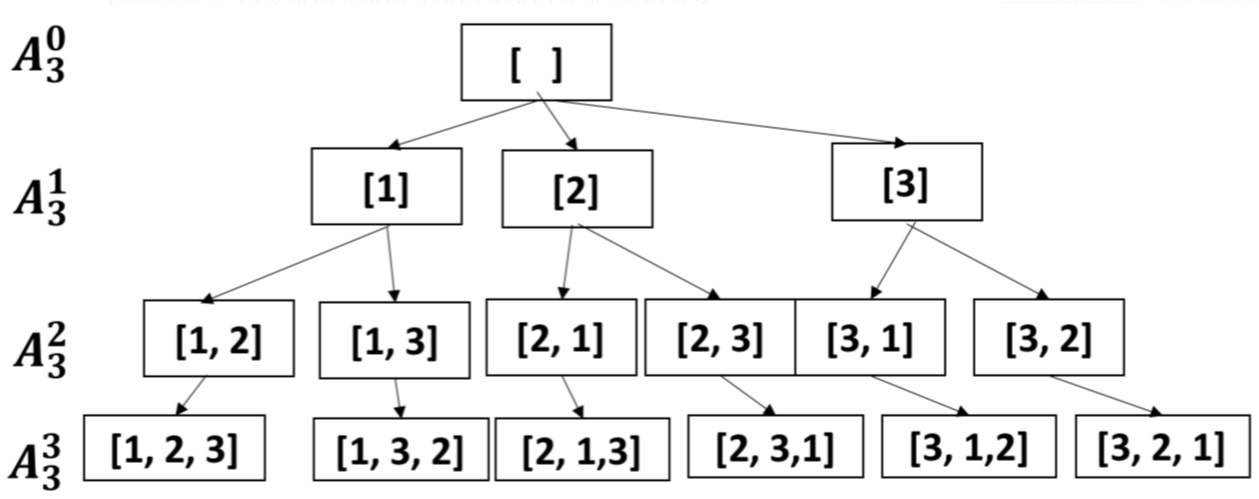
{1,2} {1,3} {2,1} {2,3} {3,1} {3,2}

| | | | | |

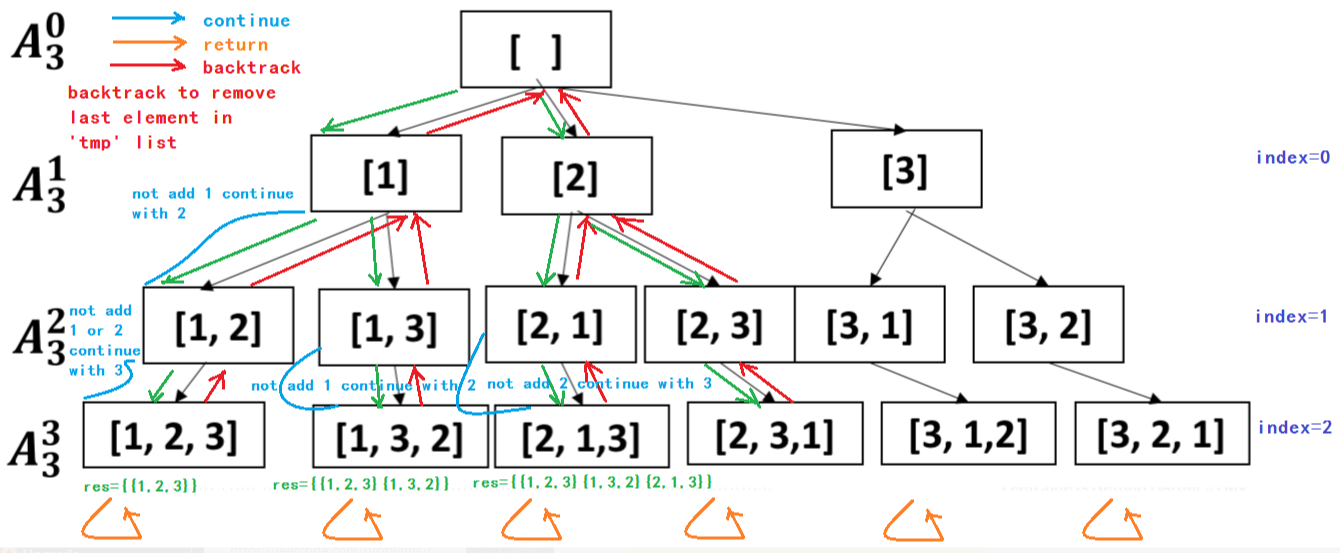
{1,2,3}{1,3,2}{2,1,3}{2,3,1}{3,1,2}{3,2,1}

**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' 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 array)**

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;

}

private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index, boolean[] visited) {

if(tmp.size() == nums.length) {

result.add(new ArrayList<Integer>(tmp));

return;

}

for(int i = index; i < nums.length; i++) {

if(visited[i]) {

continue;

}

tmp.add(nums[i]);

visited[i] = true;

helper(nums, result, tmp, index, 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**

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;

}

private void permuteRecur(int[] nums, List<List<Integer>> finalResult, List<Integer> currResult, boolean[] used) {

if (currResult.size() == nums.length) {

finalResult.add(new ArrayList<>(currResult));

return;

}

for (int i = 0; i < nums.length; i++) {

if (used[i])

continue;

currResult.add(nums[i]);

used[i] = true;

permuteRecur(nums, finalResult, currResult, used);

used[i] = false;

currResult.remove(currResult.size() - 1);

}

}

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

**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-O>

I 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:

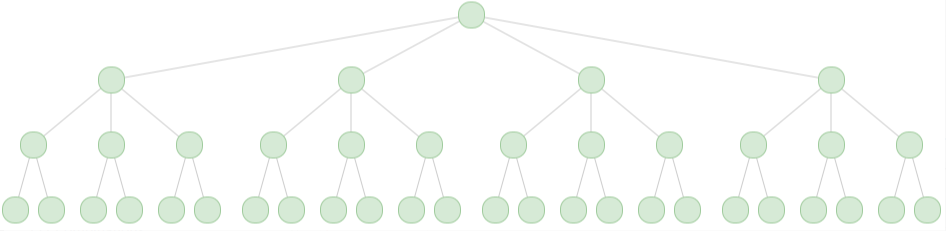
O(n) = 1 + n + n\*(n-1) + n\*(n-1)\*(n-2) + ... + n!

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