<https://leetcode.com/problems/permutation-sequence/description/>

The set [1, 2, 3, ..., n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3:

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given n and k, return the kth permutation sequence.

**Example 1:**

Input: n = 3, k = 3

Output: "213"

**Example 2:**

Input: n = 4, k = 9

Output: "2314"

**Example 3:**

Input: n = 3, k = 1

Output: "123"

**Constraints:**

* 1 <= n <= 9
* 1 <= k <= n!

**Attempt 1: 2023-7-31**

**Solution 1: Backtracking  Style 1 (10min, exactly refer to L46.Permutations, TLE 123/200)**

class Solution {

public String getPermutation(int n, int k) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(n, result, new ArrayList<Integer>());

return result.get(k - 1).stream().map(String::valueOf)

.collect(Collectors.joining(""));

}

private void helper(int n, List<List<Integer>> result, List<Integer> tmp) {

if(tmp.size() == n) {

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

return;

}

for(int i = 1; i <= n; i++) {

if(tmp.contains(i)) {

continue;

}

tmp.add(i);

helper(n, result, tmp);

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

}

}

}

**Solution 2: Backtracking  Style 2 (10min, exactly refer to L46.Permutations, use boolean[] visited instead of list.contains() method to avoid duplicates)**

class Solution {

public String getPermutation(int n, int k) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(n, result, new ArrayList<Integer>(), new boolean[n + 1]);

return result.get(k - 1).stream().map(String::valueOf)

.collect(Collectors.joining(""));

}

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

if(tmp.size() == n) {

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

return;

}

for(int i = 1; i <= n; i++) {

if(visited[i]) {

continue;

}

tmp.add(i);

visited[i] = true;

helper(n, result, tmp, visited);

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

visited[i] = false;

}

}

}

**Solution 3: Iterative style with sub-group permutation (60 min)**

class Solution {

// e.g n = 4, k = 9 -> "2314"

// factorial = {1,1,2,6,24}

// k-- = 8 (all permutations in a list start with index = 0)

// Round 1: index = 8/(4-1)! = 8/6 = 1 -> pick '2' from {1,2,3,4}, remain {1,3,4}

// Round 2: index = (8-1\*6)/(4-2)! = 2/2 = 1 -> pick '3' from {1,3,4}, remain {1,4}

// Round 3: index = (2-1\*2)/(4-3)! = 0/1 = 0 -> pick '1' from {1,4}, remain {4}

// Round 4: index = (0-0\*1)/1[since factorial[0]=1] = 0/1 = 0 -> pick only remain '4'

// e.g n = 4, k = 14 -> "3142"

// factorial = {1,1,2,6,24}

// k-- = 13 (all permutations in a list start with index = 0)

// Round 1: index = 13/(4-1)! = 13/6 = 2 -> pick '3' from {1,2,3,4}, remain {1,2,4}

// Round 2: index = (13-2\*6)/(4-2)! = 1/2 = 0 -> pick '1' from {1,2,4}, remain {2,4}

// Round 3: index = (1-0\*2)/(4-3)! = 1/1 = 1 -> pick '4' from {2,4}, remain {2}

// Round 4: index = (1-1\*1)/1[since factorial[0]=1] = 0/1 = 0 -> pick only remain '2'

public String getPermutation(int n, int k) {

List<Integer> list = new ArrayList<Integer>();

int[] factorial = new int[n + 1];

factorial[0] = 1;

for(int i = 1; i <= n; i++) {

factorial[i] = factorial[i - 1] \* i;

list.add(i);

}

StringBuilder sb = new StringBuilder();

k--;

for(int i = 1; i <= n; i++) {

int index = k / factorial[n - i];

sb.append(list.get(index));

list.remove(index);

k -= index \* factorial[n - i];

}

return sb.toString();

}

}

Time Complexity: O(N^2), the complexity is actually O(N^2), since remove in an ArrayList will take O(N) complexity.

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/permutation-sequence/solutions/22507/explain-like-i-m-five-java-solution-in-o-n/>

I'm sure somewhere can be simplified so it'd be nice if anyone can let me know. The pattern was that:

say n = 4, you have {1, 2, 3, 4}

If you were to list out all the permutations you have

1 + (permutations of 2, 3, 4)2 + (permutations of 1, 3, 4)3 + (permutations of 1, 2, 4)4 + (permutations of 1, 2, 3)

We know how to calculate the number of permutations of n numbers... n! So each of those with permutations of 3 numbers means there are 6 possible permutations. Meaning there would be a total of 24 permutations in this particular one. So if you were to look for the (k = 14) 14th permutation, it would be in the

3 + (permutations of 1, 2, 4) subset.

To programmatically get that, you take k = 13 (subtract 1 because of things always starting at 0) and divide that by the 6 we got from the factorial, which would give you the index of the number you want. In the array {1, 2, 3, 4}, k/(n-1)! = 13/(4-1)! = 13/3! = 13/6 = 2. The array {1, 2, 3, 4} has a value of 3 at index 2. So the first number is a 3.

Then the problem repeats with less numbers.

The permutations of {1, 2, 4} would be:

1 + (permutations of 2, 4)2 + (permutations of 1, 4)4 + (permutations of 1, 2)

But our k is no longer the 14th, because in the previous step, we've already eliminated the 12 4-number permutations starting with 1 and 2. So you subtract 12 from k.. which gives you 1. Programmatically that would be...

k = k - (index from previous) \* (n-1)! = k - 2\*(n-1)! = 13 - 2\*(3)! = 1

In this second step, permutations of 2 numbers has only 2 possibilities, meaning each of the three permutations listed above a has two possibilities, giving a total of 6. We're looking for the first one, so that would be in the 1 + (permutations of 2, 4) subset.

Meaning: index to get number from is k / (n - 2)! = 1 / (4-2)! = 1 / 2! = 0.. from {1, 2, 4}, index 0 is 1

so the numbers we have so far is 3, 1... and then repeating without explanations.

{2, 4}k = k - (index from pervious) \* (n-2)! = k - 0 \* (n - 2)! = 1 - 0 = 1;third number's index = k / (n - 3)! = 1 / (4-3)! = 1/ 1! = 1... from {2, 4}, index 1 has 4Third number is 4

{2}k = k - (index from pervious) \* (n - 3)! = k - 1 \* (4 - 3)! = 1 - 1 = 0;third number's index = k / (n - 4)! = 0 / (4-4)! = 0/ 1 = 0... from {2}, index 0 has 2Fourth number is 2

Giving us 3142. If you manually list out the permutations using DFS method, it would be 3142. Done! It really was all about pattern finding.

public class Solution {

public String getPermutation(int n, int k) {

int pos = 0;

List<Integer> numbers = new ArrayList<>();

int[] factorial = new int[n+1];

StringBuilder sb = new StringBuilder();

// create an array of factorial lookup

int sum = 1;

factorial[0] = 1;

for(int i=1; i<=n; i++){

sum \*= i;

factorial[i] = sum;

}

// factorial[] = {1, 1, 2, 6, 24, ... n!}

// create a list of numbers to get indices

for(int i=1; i<=n; i++){

numbers.add(i);

}

// numbers = {1, 2, 3, 4}

k--;

for(int i = 1; i <= n; i++){

int index = k/factorial[n-i];

sb.append(String.valueOf(numbers.get(index)));

numbers.remove(index);

k-=index\*factorial[n-i];

}

return String.valueOf(sb);

}

**Solution 4: Recursion style with sub-group permutation (60 min)**

**重点是通过该写法理解递归方法返回结果的一部分如何与下一层递归链接(e.g return 如何写) 的问题**

class Solution {

// e.g n = 4, k = 9 -> "2314"

// factorial = {1,2,6,24}

// k-- = 8 (all permutations in a list start with index = 0)

// Round 1: index = 8/(4-1)! = 8/6 = 1 -> pick '2' from {1,2,3,4}, remain {1,3,4}

// Round 2: index = (8-1\*6)/(4-2)! = 2/2 = 1 -> pick '3' from {1,3,4}, remain {1,4}

// Round 3: index = (2-1\*2)/(4-3)! = 0/1 = 0 -> pick '1' from {1,4}, remain {4}

// Round 4: when n = 1 hit base condition -> pick only remain '4'

// e.g n = 4, k = 14 -> "3142"

// factorial = {1,2,6,24}

// k-- = 13 (all permutations in a list start with index = 0)

// Round 1: index = 13/(4-1)! = 13/6 = 2 -> pick '3' from {1,2,3,4}, remain {1,2,4}

// Round 2: index = (13-2\*6)/(4-2)! = 1/2 = 0 -> pick '1' from {1,2,4}, remain {2,4}

// Round 3: index = (1-0\*2)/(4-3)! = 1/1 = 1 -> pick '4' from {2,4}, remain {2}

// Round 4: when n = 1 hit base condition -> pick only remain '2'

public String getPermutation(int n, int k) {

List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i <= n; i++) {

list.add(i);

}

// All permutations in a list start with index = 0

k--;

return helper(list, n, k);

}

// How to build 'return a value' recursion helper method pattern ?

private String helper(List<Integer> list, int n, int k) {

// Base condition

if(n == 1) {

// There will be only 1 digit left in list

return String.valueOf(list.get(0));

}

// Initialize a current recursion level's variable

// used as a component to build the final answer

// with a next level recusion

int num = 0;

// Build current recursion level logic to compute the

// component as 'num' defined above and prepare input

// for next level recursion

int factorial = getFactorial(n - 1);

int index = k / factorial;

num = list.get(index);

list.remove(index);

int new\_n = n - 1;

int new\_k = k - index \* factorial;

// Link current recursion level's variable as a component

// with the next level recursion

return num + helper(list, new\_n, new\_k);

}

private int getFactorial(int n) {

if(n == 1) {

return 1;

}

return n \* getFactorial(n - 1);

}

}

**对于递归方法返回结果的一部分如何与下一层递归链接(e.g return 如何写) 的问题，基本思路如下，首先得有base condition，然后分离出当前层需要计算得到的结果，该当前层结果用于最后的总和结果，比如如果最后总和结果是一个字符串，那么当前层结果可以是该字符串中的一位字符，然后预测最后为了得到最终总和结果所需的当前层结果和更深层递归的组装方法，比如还是以最终总和结果是一个字符串为例，如果当前层的目标是得到其中一位字符，那么递归的return写法可以是【return 当前层计算的字符 + 更深层递归】，最后完成当前层计算该字符的逻辑，并改变下一层递归的输入，完成递归方法**

**比如本题中在获得最终总和字符串之前，每一层就是计算一位字符，然后结合base condition也就是我们常说的"底"，在最终完成递归的时候是以最后一层(参数n控制倒数第几层)给予的参数list中只剩下唯一一个数并直接返回为状态的，那么这最后返回的一个数应该是最终总和结果字符串的最后一个字符，从"顶"到"底"的递归过程中，最终总和结果字符串的每一位字符应该是从最高位到最低位的添加顺序，那么就有了【return 当前层计算的字符 + 更深层递归】**

**以下是"添加当前层计算该字符的逻辑，并改变下一层递归的输入，完成递归方法"前的状态**

class Solution {

// e.g n = 4, k = 9 -> "2314"

// factorial = {1,2,6,24}

// k-- = 8 (all permutations in a list start with index = 0)

// Round 1: index = 8/(4-1)! = 8/6 = 1 -> pick '2' from {1,2,3,4}, remain {1,3,4}

// Round 2: index = (8-1\*6)/(4-2)! = 2/2 = 1 -> pick '3' from {1,3,4}, remain {1,4}

// Round 3: index = (2-1\*2)/(4-3)! = 0/1 = 0 -> pick '1' from {1,4}, remain {4}

// Round 4: when n = 1 hit base condition -> pick only remain '4'

// e.g n = 4, k = 14 -> "3142"

// factorial = {1,2,6,24}

// k-- = 13 (all permutations in a list start with index = 0)

// Round 1: index = 13/(4-1)! = 13/6 = 2 -> pick '3' from {1,2,3,4}, remain {1,2,4}

// Round 2: index = (13-2\*6)/(4-2)! = 1/2 = 0 -> pick '1' from {1,2,4}, remain {2,4}

// Round 3: index = (1-0\*2)/(4-3)! = 1/1 = 1 -> pick '4' from {2,4}, remain {2}

// Round 4: when n = 1 hit base condition -> pick only remain '2'

public String getPermutation(int n, int k) {

List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i <= n; i++) {

list.add(i);

}

// All permutations in a list start with index = 0

k--;

return helper(list, n, k);

}

// How to build 'return a value' recursion helper method pattern ?

private String helper(List<Integer> list, int n, int k) {

// Base condition

if(n == 1) {

// There will be only 1 digit left in list

return String.valueOf(list.get(0));

}

// Initialize a current recursion level's variable

// used as a component to build the final answer

// with a next level recusion

int num = 0;

// Build current recursion level logic to compute the

// component as 'num' defined above and prepare input

// for next level recursion (updated\_list, updated\_n, updated\_k)

{pending ?}

// Link current recursion level's variable as a component

// with the next level recursion

return num + helper(updated\_list {pending ?}, updated\_n {pending ?}, updated\_k {pending ?});

}

}

**最后省略掉以上这些中间思考过程优化如下：**

class Solution {

public String getPermutation(int n, int k) {

List<Integer> list = new ArrayList<Integer>();

int[] factorial = new int[n + 1];

factorial[0] = 1;

for(int i = 1; i <= n; i++) {

list.add(i);

factorial[i] = factorial[i - 1] \* i;

}

k--;

return helper(list, n, k, factorial);

}

private String helper(List<Integer> list, int n, int k, int[] factorial) {

if(n == 1) {

return String.valueOf(list.get(0));

}

int index = k / factorial[n - 1];

int num = list.get(index);

list.remove(index);

return num + helper(list, n - 1, k - index \* factorial[n - 1], factorial);

}

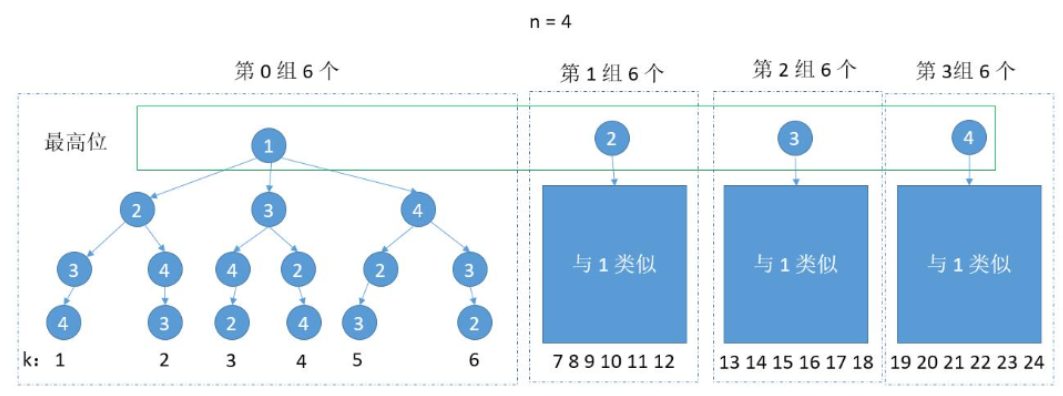
}

**Refer to**

<https://leetcode.wang/leetCode-60-Permutation-Sequence.html>

# **解法一**

以 n = 4 为例，可以结合下图看一下。因为是从小到大排列，那么最高位一定是从 1 到 4。然后可以看成一组一组的，我们只需要求出组数，就知道最高位是多少了。而每组的个数就是 n - 1 的阶乘，也就是 3 的阶乘 6。



算组数的时候， 1 到 5 除以 6 是 0，6 除以 6 是 1，而 6 是属于第 0 组的，所有要把 k 减去 1。这样做除法结果就都是 0 了。

int perGroupNum = factorial(n - 1);

int groupNum = (k - 1) / perGroupNum;

当然，还有一个问题下次 k 是多少了。求组数用的除法，余数就是下次的 k 了。因为 k 是从 1 计数的，所以如果 k 刚好等于了 perGroupNum 的倍数，此时得到的余数是 0 ，而其实由于我们求 groupNum 的时候减 1 了，所以此时 k 应该更新为 perGroupNum。

k = k % perGroupNum;

k = k == 0 ? perGroupNum : k;

举个例子，如果 k = 6，那么 groupNum = ( k - 1 ) / 6 = 0， k % perGroupNum = 6 % 6 = 0，而下次的 k ，可以结合上图，很明显是 perGroupNum ，依旧是 6。

结合下图，确定了最高位属于第 0 组，下边就和上边的情况一样了。唯一不同的地方是最高位是 2 3 4，没有了 1。所有得到 groupNum 怎么得到最高位需要考虑下。

我们可以用一个 list 从小到大保存 1 到 n，每次选到一个就去掉，这样就可以得到 groupNum 对应的数字了。

List<Integer> nums = new ArrayList<Integer>();

for (int i = 0; i < n; i++) {

nums.add(i + 1);

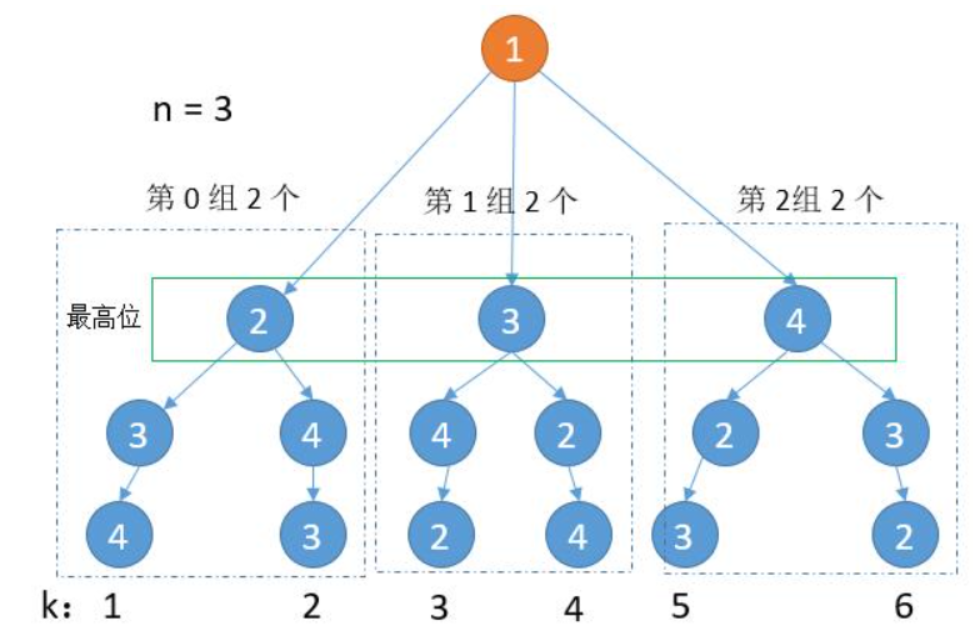
}

int perGroupNum = factorial(n - 1);

int groupNum = (k - 1) / perGroupNum;

int num = nums.get(groupNum); //根据 groupNum 得到当前位

nums.remove(groupNum);//去掉当前数字



综上，我们把它们整合在一起。

public String getPermutation(int n, int k) {

List<Integer> nums = new ArrayList<Integer>();

for (int i = 0; i < n; i++) {

nums.add(i + 1);

}

return getAns(nums, n, k);

}

private String getAns(List<Integer> nums, int n, int k) {

if (n == 1) {

//把剩下的最后一个数字返回就可以了

return nums.get(0) + "";

}

int perGroupNum = factorial(n - 1); //每组的个数

int groupNum = (k - 1) / perGroupNum;

int num = nums.get(groupNum);

nums.remove(groupNum);

k = k % perGroupNum; //更新下次的 k

k = k == 0 ? perGroupNum : k;

return num + getAns(nums, n - 1, k);

}

public int factorial(int number) {

if (number <= 1)

return 1;

else

return number \* factorial(number - 1);

}

这是最开始自己的想法，有 3 点可以改进一下。

第 1 点，更新 k 的时候，有一句

k = k % perGroupNum; //更新下次的 k

k = k == 0 ? perGroupNum : k;

很不优雅了，问题的根源就在于问题给定的 k 是从 1 编码的。我们只要把 k - 1 % perGroupNum，这样得到的结果就是 k 从 0 编码的了。然后求 groupNum = (k - 1) / perGroupNum; 这里 k 也不用减 1 了。

第 2 点，这个算法很容易改成改成迭代的写法，只需要把递归的函数参数， 在每次迭代更新就够了。

第 3 点，我们求 perGroupNum 的时候，每次都调用了求迭代的函数，其实没有必要的，我们只需要一次循环求出 n 的阶乘。然后在每次迭代中除以 nums 的剩余个数就够了。

综上，看一下优化过的代码吧。

public String getPermutation(int n, int k) {

List<Integer> nums = new ArrayList<Integer>();

int factorial = 1;

for (int i = 0; i < n; i++) {

nums.add(i + 1);

if (i != 0) {

factorial \*= i;

}

}

factorial \*= n; //先求出 n 的阶乘

StringBuilder ans = new StringBuilder();

k = k - 1; // k 变为 k - 1

for (int i = n; i > 0; i--) {

factorial /= (nums.size()); //更新为 n - 1 的阶乘

int groupNum = k / factorial;

int num = nums.get(groupNum);

nums.remove(groupNum);

k = k % factorial;

ans.append(num);

}

return ans.toString();

}

时间复杂度：O（n），当然如果 remove 函数的时间是复杂度是 O（n），那么整体上就是 O（n²）。

空间复杂度：O（1）。

# **总**

这道题其实如果写出来，也不算难，优化的思路可以了解一下。