## 60. Permutation Sequence

Recursion <u>Math</u> Hard

## **Problem Description**

that the set includes all permutations possible for numbers from 1 to n. The total number of unique permutations is n! (factorial of n). The permutations are ordered in a sequence based on the natural ordering of integers. For example, for n = 3, the sequence of permutations from the first (1st) to the last (6th) is 123, 132, 213, 231, 312, 321. Given a number n representing the length of the permutation, and a number k, the task is to return the kth permutation in the ordered sequence. Intuition

The problem presents a scenario where we are interested in finding a specific permutation of the set [1, 2, 3, ..., n], given

observation here is to understand how permutations are structured: For any given n, the first (n-1)! permutations begin with 1, the next (n-1)! permutations begin with 2, and so forth. This holds true for every subsequent digit in the permutation.

To arrive at the solution, we leverage the fact that the permutations are ordered and can be generated in a sequence. The key

- Hence, we can determine the first digit of the kth permutation by computing how many blocks of (n-1)! permutations fit into k.
- Subtract the number of full blocks from k to get the new k for the next iteration, as we proceed to find the next digit of the permutation.
- We use an array vis to keep track of which numbers have already been included in the permutation, since each number can

By repeating the process, selecting one digit at a time for the permutation, we can construct the kth permutation without

- The solution uses this approach iteratively, where for each digit in the permutation, it calculates the appropriate digit given k's position in the remaining factorial blocks. This results in a direct path to the kth permutation sequence without unnecessary
- **Solution Approach** The implementation of the solution is as follows:

• A list vis of n+1 elements is created to keep track of the numbers that have been visited (i.e., already included in the permutation), initialized to

## False for all elements.

permutation:

computations.

appear only once.

Now, the algorithm enters a for-loop that iterates n times, once for each position in the permutation string.

having to generate all the permutations up to k.

For each position  $\mathbf{i}$  in the permutation, calculate the factorial of  $(\mathbf{n} - \mathbf{i} - \mathbf{1})$ , which is the number of permutations possible

For each digit j, if it hasn't been used yet (not vis[j] is True):

string representing the kth permutation is returned.

• A list ans is initialized to hold the characters of the final permutation sequence.

for the remaining digits after fixing the first i digits. This is done using a nested for-loop to multiply the factorial value up to (n -i-1).

Another for-loop is used to iterate over the potential digits j (from 1 to n) that can go into the current position of the

■ If k > fact, decrement k by the value of fact, effectively moving k to the correct block within the permutations sequence. This does

If k <= fact, we've found the correct digit for this position. Append this digit to ans, mark it as visited in vis[j], and break from the loop</p>

 Check if the current k is greater than the calculated factorial. This would mean that k lies beyond the current block of permutations that start with the digit j.

to continue to the next position in the permutation. The break statement exits the inner for-loop early once the correct digit is found for the current position, preventing unnecessary checks on higher numbers.

not change the current j, allowing the loop to continue to the next iteration and j to be tested for the next factorial block.

After the outer for-loop ends, all the digits are placed correctly in ans. The list ans is then joined to form a string, and this

- By using this approach, the algorithm systematically determines each digit of the kth permutation sequence by excluding permutations blocks where the kth sequence would not fall into. This is done calculating the factorial decrement for each position
- generated, making the implementation efficient and scalable for larger n.

in the sequence, allowing for an efficient and direct construction of the desired sequence. No unnecessary permutations are

the set [1, 2, 3]. **Initialization**: We create an answer list ans = [] to hold the characters of the final permutation and a visited list vis = [False, False, False, False] to track the numbers that have been used. The extra index at the start of vis is to align the number positions with their index for easier access (since we don't use 0). **First Position:** 

∘ Iterate over digits and find where the 4th permutation would fall. Since 2! blocks start with 1, and 4 > 2, the first number can't be 1. We then

Let's illustrate the solution approach with n = 3 and k = 4 as an example, which means we want to find the 4th permutation of

### Since the second block (numbers starting with 2) fits our k (which is now 2), the first number is 2. We update ans = [2] and vis = [False, True, False, False].

**Second Position:** 

subtract 2 from k to get the new k = 2.

Calculate new factorial for remaining digits: 1! = 1.

def getPermutation(self, n: int, k: int) -> str:

**Example Walkthrough** 

 $\circ$  Calculate factorial of (3 - 1) = 2! = 2. This tells us each block starting with a particular number has 2 permutations.

- Now we look for the second digit. k is 2, which means within the block starting with 2, we need the second permutation. ○ We skip 1 because k > 1!, but when we reach 3 (vis[3] is False), k is not greater than 1!, so we select 3. We update ans = [2, 3] and vis = [False, True, True, False]. **Third Position:**
- Only one number is left (1), and since vis[1] is False, it is the only choice. ans = [2, 3, 1]. **Final Result:** Join all the numbers in ans to form the permutation string. So the 4th permutation of the set [1, 2, 3] is "231".

Following the steps, we have determined that the 4th permutation in the ordered sequence is indeed "231". This method

class Solution:

**Python** 

Solution Implementation

# Initialize an empty list to store the permutation permutation = []

# Compute the factorial of (n-i-1) which helps in determining the blocks

visited[j] = True # Mark the number as visited

# Found the right place for the number 'j' in the permutation

permutation.append(str(j)) # Add the number to the permutation

break # Break since we have used one number in the permutation

efficiently arrived at the answer without exhaustively enumerating all permutations.

# Initialize a list to keep track of used numbers visited = [False] \* (n + 1)# Iterate through the numbers from 1 to n for i in range(n):

# Iterate through the numbers to find the unused numbers

# This function returns the k-th permutation of the numbers 1 to n

### for j in range(1, n + 1): if not visited[j]: # If k is greater than the factorial, it means we need to move to the next block if k > factorial:

factorial = 1

for j in range(1, n - i):

k -= factorial

// Return the final permutation string

// This function returns the kth permutation of the sequence of integers [1, n]

// Go through the numbers 1 to n to find the suitable one for current position

// Found the number for the current position, add it to permutation

break; // Break since we found the number for the current position

// If there are more than 'factorial' permutations left, skip 'factorial' permutations

if (visited[j]) continue; // Skip if the number is already used

string permutation; // This will store our resulting permutation

bitset<10> visited; // A bitset to keep track of visited numbers

// Calculate the factorial for the remaining numbers

for (int j = 1; j < n - i; ++j) factorial \*= j;

permutation += to\_string(j);

return permutation; // Return the resulting permutation

return permutation; // Return the resulting permutation

# Initialize an empty list to store the permutation

# Initialize a list to keep track of used numbers

# This function returns the k-th permutation of the numbers 1 to n

# Iterate through the numbers to find the unused numbers

# Join the list of strings to form the final permutation string

# Compute the factorial of (n-i-1) which helps in determining the blocks

visited[j] = True # Mark the number as visited

# If k is greater than the factorial, it means we need to move to the next block

# Found the right place for the number 'j' in the permutation

permutation.append(str(j)) # Add the number to the permutation

break # Break since we have used one number in the permutation

def getPermutation(self, n: int, k: int) -> str:

# Iterate through the numbers from 1 to n

visited.set(j); // Mark this number as used

// Iterate through each position in the permutation

return permutation.toString();

string getPermutation(int n, int k) {

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

for (int j = 1;  $j \le n$ ; ++j) {

if (k > factorial) {

k -= factorial;

int factorial = 1;

} else {

factorial \*= j

else:

```
# Join the list of strings to form the final permutation string
       return ''.join(permutation)
Java
class Solution {
    public String getPermutation(int n, int k) {
       // StringBuilder to create the resulting permutation string
        StringBuilder permutation = new StringBuilder();
       // Visited array keeps track of which numbers have been used
       boolean[] visited = new boolean[n + 1];
       // Loop through each position in the permutation
        for (int i = 0; i < n; ++i) {
           // Calculate the factorial of the numbers left
           int factorial = 1;
            for (int j = 1; j < n - i; ++j) {
                factorial *= j;
           // Find the number to put in the current position
            for (int j = 1; j <= n; ++j) {
               if (!visited[j]) {
                   // If the remaining permutations are more than k,
                   // decrease k and find the next number
                   if (k > factorial) {
                        k -= factorial;
                    } else {
                        // Add the number to the result and mark it as visited
                        permutation.append(j);
                        visited[j] = true;
                        break;
```

public:

#include <string>

#include <bitset>

class Solution {

using namespace std;

```
TypeScript
let visited: boolean[] = new Array(10).fill(false); // An array to keep track of visited numbers
// This function returns the factorial of a given number
function factorial(n: number): number {
    let result = 1;
   for (let i = 1; i <= n; ++i) {
       result *= i;
   return result;
// This function returns the kth permutation of the sequence of integers [1, n]
function getPermutation(n: number, k: number): string {
    let permutation: string = ''; // This will store our resulting permutation
   // Iterate through each position in the permutation
   for (let i = 0; i < n; ++i) {
       // Calculate the factorial for the remaining numbers
       let factorialValue = factorial(n - i - 1);
       // Go through the numbers 1 to n to find the suitable one for current position
        for (let j = 1; j <= n; ++j) {
            if (visited[j]) {
               continue; // Skip if the number is already used
           // If there are more than 'factorialValue' permutations left, skip 'factorialValue' permutations
           if (k > factorialValue) {
               k -= factorialValue;
            } else {
               // Found the number for the current position, add it to permutation
                permutation += j.toString();
               visited[j] = true; // Mark this number as used
               break; // Break since we found the number for the current position
```

### Time and Space Complexity **Time Complexity**

return ''.join(permutation)

# The time complexity can be analyzed as follows:

n-1 natural numbers.

class Solution:

permutation = []

for i in range(n):

factorial = 1

visited = [False] \* (n + 1)

for j in range(1, n - i):

for j in range(1, n + 1):

if not visited[j]:

if k > factorial:

k -= factorial

digit of the permutation that is not yet visited.

factorial \*= j

A nested loop structure is employed where the outer loop runs for n iterations (where n is the number of digits), and the inner loop calculates the factorial (fact) for the remaining positions and then runs up to n again in the worst case to find the next

The given code's primary operation is finding the k-th permutation out of the possible permutations for a set of n numbers.

- Within the outer loop, calculating the factorial of n-i-1 takes 0(n-i-1) time, where i is the index of the current iteration of the outer loop, starting with 0. This calculation is performed n times, leading to a sum of time complexities for factorial calculations given by  $0((n-1) + (n-2) + \dots + 1)$ , which simplifies to 0(n\*(n-1)/2), using the formula for the sum of the first
- Also within the outer loop, the worst-case scenario for the inner loop to find the next digit is when it runs n times for each iteration of the outer loop. This gives us O(n) time complexity for each of the n iterations of the outer loop, adding up to O(n^2) for the complete for-loop.
- since  $n^2$  dominates (n\*(n-1)/2). **Space Complexity**

Therefore, the total time complexity for this nested loop construction is  $0(n*(n-1)/2) + 0(n^2)$ , which simplifies to  $0(n^2)$ 

## The space complexity of the given code can be considered based on the following:

- An array vis of size n+1 is used to track which numbers have been included in the permutation, leading to 0(n) space. An array ans is maintained to store the digits of the permutation incrementally, adding up to a maximum of n digits, which is O(n) space.
- As a result, the overall space complexity of the code is O(n), coming from the space used to store the vis array and the ans list.