L21 Permutation Sequence or Kth Permutation Sequence

Saturday, June 17, 2023 10:11 AM

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"

→ 6

→ In this problem we exactly n! permutation
→ n = 3
So total permutation is 6

Given n and k, return the kth permutation sequence.

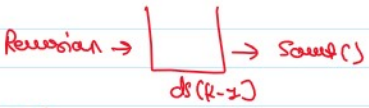
Example 1:
Input: n = 3, k = 3
Output: "213"
Example 2:
Input: n = 4, k = 9
Output: "2314"
Example 3:
Input: n = 3, k = 1
Output: "123"

Constraints:

- 1 <= n <= 9
- 1 <= k <= n!

Thought Process

→ If this problem comes into the interview then we used the Recursion here finding the All the unique permutation



Time complexity: O(N! * N) +O(N! Log N!)
Reason: The recursion takes O(N!) time because we generate every possible permutation and another O(N) time is required to make a deep copy and store every sequence in the data structure. Also, *O(N! Log N!) time required to sort the data structure*

Space complexity: O(N)
Reason: Result stored in a vector, *we are auxiliary space taken by recursion*

→ worst time interviewer not happy

```python
# todo using Recursion

class Solution:
    def getPermutation(self, n: int, k: int) -> str:
        ans = ""
        res = []
        # create string
        for i in range(1, n + 1):
            ans += str(i)
        self.permutationHelper(ans, 0, res)
        # sort the generated permutations
        res.sort()

        print(res)

        # make k 0-based indexed to point to kth sequence
        return res[k - 1]

    2 usages
    def permutationHelper(self, ans: str, index: int, res: List[str]) -> None:
        if index == len(ans):
            res.append(ans)
            return

        # Recursion and Backtracking
        for i in range(index, len(ans)):
            ans = self.swap(ans, i, index)
            self.permutationHelper(ans, index + 1, res)
            ans = self.swap(ans, i, index)

    2 usages
    def swap(self, s: str, i: int, j: int) -> str:
        s = list(s)
        s[i], s[j] = s[j], s[i]
        return "".join(s)
```

```java
import java.util.ArrayList;
import java.util.List;

public class L20_Palindrome_Partitioning {
    public static void main(String[] args) {

        System.out.println("L20_Palindrome_Partitioning");
    }
}

no usages
class Solution14 {

    no usages
    public static List <List< String >> partition(String s) {
        List < List < String >> res = new ArrayList< >();
        List < String > path = new ArrayList < > ();
        partitionHelper( index: 0, s, path, res);
        return res;
    }

    2 usages
    static void partitionHelper(int index, String s, List < String > path, List < List < String >> res) {
        if (index == s.length()) {
            res.add(new ArrayList < > (path));
            return;
        }

        for (int i = index; i < s.length(); ++i) {
            if (isPalindrome(s, index, i)) {

                path.add(s.substring(index, i + 1));
                partitionHelper( index: i + 1, s, path, res);

                // backtrack
                path.remove( index: path.size() - 1);
            }
        }
    }

    1 usage
    static boolean isPalindrome(String s, int start, int end) {

        while (start <= end) {
            if (s.charAt(start) != s.charAt(end)){
                return false;
            }
            start++;
            end--;
        }
        return true;
    }
}
```

Optimized Approach

n=4
k=17

→ if N is 4 then total permutation is (24)

Apply Mathematical Approach

num = [1, 2, 3, 4]

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

can say

3!
[(2, 3, 4)] → 6 permutation for every Number
[(1, 3, 4)] → 6
[(1, 2, 4)] → 6
[(1, 2, 3)] → 6
(24)

Starting Number of permutation → Rest of the Number

1 + (2, 3, 4)] → 6 possible permutation
2 + (1, 3, 4)] → 6
3 + (1, 2, 4)] → 6
4 + (1, 2, 3)] → 6
24 total Permutation

→ Now we used the data structure for storing the Range

n=4 ↓ [1, 2, 3, 4]
0 1 2 3

k=17

→ Now I am going to Mark All the unique 0th Based Index Permutation

Definitely 0th Index = 1, 2, 3, 4 (0)th permutation

→ Now I am going to mark all the unique 0th based index permutation

Definitely 0th index = 1,2,3,4 (0)th permutation

last 0th index = 4,3,2,1 (23)th permutation

→ So we can write the Range

→ Range

1 + (2,3,4)] 6 [0-5]
2 + (1,3,4)] 6 [6-11]
3 + (1,2,4)] 6 [12-17]
4 + (1,2,3)] 6 [18-23]

So if the Question says finding the kth permutation then I find in the that Range particular

k=17 then finding the 16th permutation

→ Range

1 + (2,3,4)] 6 [0-5]
2 + (1,3,4)] 6 [6-11]
(3) + (1,2,4)] 6 [12-17] ✓ my Range Here
4 + (1,2,3)] 6 [18-23]

→ So Definitely my Number start with (3)

3 _ _ _

How = ?
→ if the my Remaining All the permutation is (6)
then (16)/6 = (2)
[1,2,(3),4] → that's why
0 1 2 3

→ Now How to Identify the How Many Permutation in the Given Range

16 % 6
16%6 = (4) permutation So
k=4

→ Now my task is the finding the 4th sequence from the 6th sequence

3 + (1,2,4) = 6 [12-17]

↳ Generate the permutation from Remaining element

[1,2,4], k=4

↳ Again follow the same process like previously

(03)

1 + (2,4) = 2 [0,1]
2 + (1,4) = 2 [2,3]
(4) + (1,2) = 2 [4,5] ✓ my Range comes here

6 permutation possible

→ So my starting element is (4)

How = ?
→ if the my Remaining All the permutation is (2)
then k/2 = 2
[1,2,(4)] → that's why
0 1 2

3 4 _ _

→ Now How to Identify the How Many Permutation in the

_ _ _ _

[1,2,4] → that is wrong
0 1 2

→ Now How to identify the Next Nearest Permutation in the Given Range

K % 2
4 % 2 = 0
K = 0

[1,2], K=0

→ Again follows the same procedure

1 + [2] = 1 [0,0] ✓
2 + [1] = 1 [1,1]

2 permutation possible

→ So My Stationary element is 1

How = ?
→ if the my Remaining All the permutation is 1
then K [1] = 0
0/1
[1,2] → that is wrong
0 1

3 4 1 _

→ Now How to identify the Next Nearest Permutation in the Given Range

0 % 1
0 % 1 permutation So
K = 0

[2], K=0 → Again follows same procedure

2 + Nothing
→ that is the case if any thing is not Remaining that simply put the element

3 4 1 2 → Answer

→ This is the way without using Recursion we find out the Recursion

Time Complexity: O(N) * O(N) = O(N^2)

Reason: We are placing N numbers in N positions. This will take O(N) time. For every number, *we are reducing the search space by removing the element already placed in the previous step. This takes another O(N) time.*

Space Complexity: O(N)

Reason: We are storing the numbers in a data structure(here vector)

```python
# todo Optimized Solution

class Solution:
    def getPermutation(self, n: int, k: int) -> str:
        fact = 1
        numbers = []

        # finding the all the Factorial
        for i in range(1, n):
            fact *= i
            numbers.append(i)

        numbers.append(n)
        print(numbers)
        print(fact)

        ans = ""
        k -= 1

        while True:

            # store the first index value
            ans += str(numbers[k // fact])

            # pop that element from num
            numbers.pop(k // fact)

            if not numbers:
                break

            # find the new number of the k
            k %= fact

            # find the new Permutation range
            fact //= len(numbers)

        return ans
```

```java
// todo Optimized Solution
class Solution16 {
    1 usage
    static String getPermutation(int n, int k) {
        int fact = 1;
        ArrayList < Integer > numbers = new ArrayList < > ();

        for (int i = 1; i < n; i++) {
            fact = fact * i;
            numbers.add(i);
        }

        numbers.add(n);
        String ans = "";
        k = k - 1;

        while (true) {
            ans = ans + "" + numbers.get(k / fact);
            numbers.remove( index: k / fact);
            if (numbers.size() == 0) {
                break;
            }

            k = k % fact;
            fact = fact / numbers.size();
        }
        return ans;
    }


    public static void main(String args[]) {
        int n = 3, k = 3;
        String ans = getPermutation(n, k);
        System.out.println("The Kth permutation sequence is " + ans);


    }
}
```