Previous Next

526. Beautiful Arrangement

April 29, 2017 | 30.7K views

Suppose you have **N** integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these **N** numbers successfully if one of the following is true for the i_{th} position (1 <= i <= N) in this array:

1. The number at the i_{th} position is divisible by **i**.
2. **i** is divisible by the number at the i_{th} position.

Now given N, how many beautiful arrangements can you construct?

Example 1:

```
Input: 2
Output: 2
Explanation:

The first beautiful arrangement is [1, 2]:

Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

The second beautiful arrangement is [2, 1]:

Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.
```

Note:

1. **N** is a positive integer and will not exceed 15.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

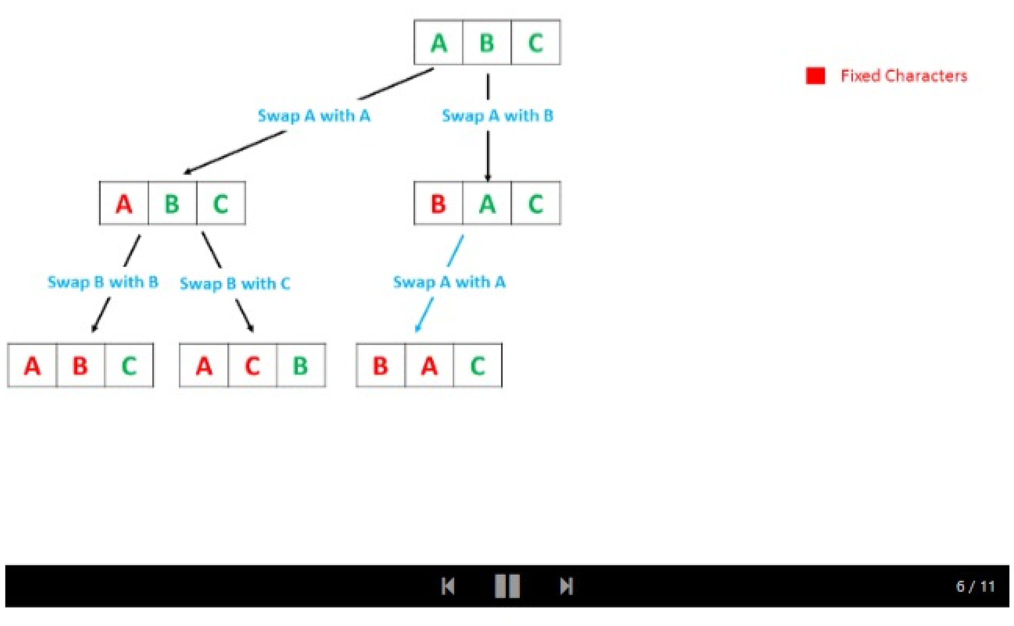
Algorithm

In the brute force method, we can find out all the arrays that can be formed using the numbers from 1 to N(by creating every possible permutation of the given elements). Then, we iterate over all the elements of every permutation generated and check for the required conditions of divisibility.

In order to generate all the possible pairings, we make use of a function `permute(nums, current_index)`. This function creates all the possible permutations of the elements of the given array.

To do so, `permute` takes the index of the current element $current_index$ as one of the arguments. Then, it swaps the current element with every other element in the array, lying towards its right, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to permute but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call.

Thus, when we reach the end of the array, a new ordering of the array's elements is generated. The following animation depicts the process of generating the permutations.



Java Copy

```java

public class Solution {
    int count = 0;
    public int countArrangement(int N) {
        int[] nums = new int[N];
        for (int i = 1; i <= N; i++)
            nums[i - 1] = i;
        permute(nums, 0);
        return count;
    }
    public void permute(int[] nums, int l) {
        if (l == nums.length - 1) {
            int i;
            for (i = 1; i <= nums.length; i++) {
                if (nums[i - 1] % i != 0 && i % nums[i - 1] != 0)
                    break;
            }
            if (i == nums.length + 1) {
                count++;
            }
        }
        for (int i = l; i < nums.length; i++) {
            swap(nums, i, l);
            permute(nums, l + 1);
            swap(nums, i, l);
        }
    }
    public void swap(int[] nums, int x, int y) {
```

Complexity Analysis

- Time complexity : $O(n!)$. A total of $n!$ permutations will be generated for an array of length n.
- Space complexity : $O(n)$. The depth of the recursion tree can go upto n. $nums$ array of size n is used.

Approach #2 Better Brute Force [Accepted]

Algorithm

In the brute force approach, we create the full array for every permutation and then check the array for the given divisibilty conditions. But this method can be optimized to a great extent. To do so, we can keep checking the elements while being added to the permutation array at every step for the divisibility condition and can stop creating it any further as soon as we find out the element just added to the permutation violates the divisiblity condition.

Java Copy

```java

public class Solution {
    int count = 0;
    public int countArrangement(int N) {
        int[] nums = new int[N];
        for (int i = 1; i <= N; i++)
            nums[i - 1] = i;
        permute(nums, 0);
        return count;
    }
    public void permute(int[] nums, int l) {
        if (l == nums.length) {
            count++;
        }
        for (int i = l; i < nums.length; i++) {
            swap(nums, i, l);
            if (nums[l] % (l + 1) == 0 || (l + 1) % nums[l] == 0)
                permute(nums, l + 1);
            swap(nums, i, l);
        }
    }
    public void swap(int[] nums, int x, int y) {
        int temp = nums[x];
        nums[x] = nums[y];
        nums[y] = temp;
    }
}
```

Complexity Analysis

- Time complexity : $O(k)$. k refers to the number of valid permutations.
- Space complexity : $O(n)$. The depth of recursion tree can go upto n. Further, $nums$ array of size n is used, where, n is the given number.

Approach #3 Backtracking [Accepted]

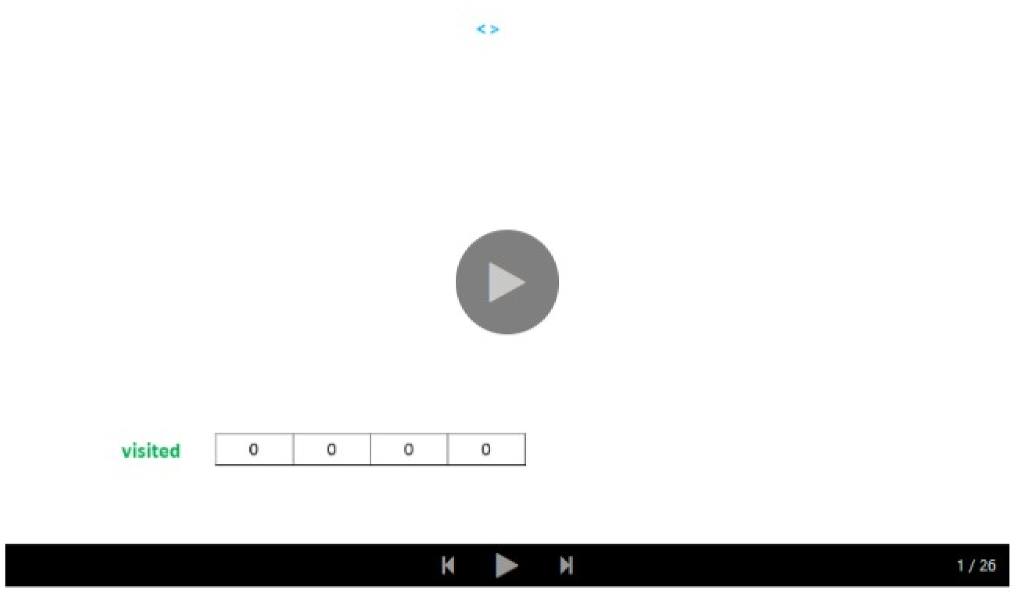
Algorithm

The idea behind this approach is simple. We try to create all the permutations of numbers from 1 to N. We can fix one number at a particular position and check for the divisibility criteria of that number at the particular position. But, we need to keep a track of the numbers which have already been considered earlier so that they aren't reconsidered while generating the permutations. If the current number doesn't satisfy the divisibility criteria, we can leave all the permutations that can be generated with that number at the particular position. This helps to prune the search space of the permutations to a great extent. We do so by trying to place each of the numbers at each position.

We make use of a visited array of size N. Here, $visited[i]$ refers to the i^{th} number being already placed/not placed in the array being formed till now(True indicates that the number has already been placed).

We make use of a `calculate` function, which puts all the numbers pending numbers from 1 to N(i.e. not placed till now in the array), indicated by a $False$ at the corresponding $visited[i]$ position, and tries to create all the permutations with those numbers starting from the pos index onwards in the current array. While putting the pos^{th} number, we check whether the i^{th} number satisfies the divisibility criteria on the go i.e. we continue forward with creating the permutations with the number i at the pos^{th} position only if the number i and pos satisfy the given criteria. Otherwise, we continue with putting the next numbers at the same position and keep on generating the permutations.

Look at the animation below for a better understanding of the methodology:



Below code is inspired by @shawngao

Java Copy

```java
public class Solution {
    int count = 0;
    public int countArrangement(int N) {
        boolean[] visited = new boolean[N + 1];
        calculate(N, 1, visited);
        return count;
    }
    public void calculate(int N, int pos, boolean[] visited) {
        if (pos > N)
            count++;
        for (int i = 1; i <= N; i++) {
            if (!visited[i] && (pos % i == 0 || i % pos == 0)) {
                visited[i] = true;
                calculate(N, pos + 1, visited);
                visited[i] = false;
            }
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(k)$. k refers to the number of valid permutations.
- Space complexity : $O(n)$. $visited$ array of size n is used. The depth of recursion tree will also go upto n. Here, n refers to the given integer n.

Rate this article:

Previous Next

Comments: 13 Sort By

Type comment here... (Markdown is supported)

Preview Post

william5 6 December 2, 2018 6:40 AM
Is approach 3 really O(num permutations?). I feel like it's possible to deep into the recursion before realizing it's a dead end.
6 Share Reply
SHOW 1 REPLY

pikachang272 54 June 7, 2018 6:37 AM
This question is very confusing and doesn't make sense to me. Not even the given example makes sense. How is it that [2,1] counts as a beautiful arrangement? 1 is at the 2nd position but 1%2 != 0. And 2 is at the 1st position but, again, 1%2 != 0. Can someone please explain if you understand, thanks
7 Share Reply
SHOW 3 REPLIES

wierzba 103 December 23, 2017 3:27 AM
Can someone please help clarify why input of N=3 should return 3?
Permutations of N=3: [123], [132], [213], [231], [321], [312]
According to problem statement, valid permutation is one such that either is true:
1. The number at the ith position is divisible by i.
2. i is divisible by the number at the ith position.
Read More
2 Share Reply
SHOW 2 REPLIES

prohorenko 3 June 7, 2018 5:39 AM
Is it possible to explain how was counted time complexity and depth of recursion tree?
1 Share Reply

AbeyK 4 October 8, 2017 6:02 AM
Approach #3 is TLE for Python. Please fix
1 Share Reply
SHOW 1 REPLY

random_number 3 September 10, 2017 6:11 AM
Space complexity seems incorrect. We're using just one array of O(n) size all 3 approaches.
1 Share Reply

user1833 0 June 10, 2018 9:37 PM
Why can't we find the number of unique arrangements for each number and multiply them via multiplication rule? For example, for N = 4,
1 <=> 1
1 <=> 2
1 <=> 3
Read More
0 Share Reply
SHOW 1 REPLY

coolbond007 0 March 17, 2018 4:02 PM
@wierzba the condition should be valid for every i in the array.
0 Share Reply

GoingMyWay 138 October 5, 2017 4:41 PM
Approach #3 got TLE in Python. And a `return` statement is needed in function `calculate`.
0 Share Reply
SHOW 1 REPLY

pratikgaikar 0 October 5, 2017 2:08 PM
use || instead of && in 1st solution
0 Share Reply