1718. Construct the Lexicographically Largest Valid Sequence

Given an integer `n`, find a sequence that satisfies all of the following:

- The integer `1` occurs once in the sequence.
- Each integer between `2` and `n` occurs twice in the sequence.
- For every integer `i` between `2` and `n`, the **distance** between the two occurrences of `i` is exactly `i`.

The **distance** between two numbers on the sequence, `a[i]` and `a[j]`, is the absolute difference of their indices, `|j - i|`.

Return *the **lexicographically largest** sequence. It is guaranteed that under the given constraints, there is always a solution.*

A sequence `a` is lexicographically larger than a sequence `b` (of the same length) if in the first position where `a` and `b` differ, sequence `a` has a number greater than the corresponding number in `b`. For example, `[0,1,9,0]` is lexicographically larger than `[0,1,5,6]` because the first position they differ is at the third number, and `9` is greater than `5`.

Example 1:

```
Input: n = 3
Output: [3,1,2,3,2]
Explanation: [2,3,2,1,3] is also a valid sequence, but [3,1,2,3,2] is the lexicographically largest valid sequence.
```

Example 2:

```
Input: n = 5
Output: [5,3,1,4,3,5,2,4,2]
```

Constraints:

- `1 <= n <= 20`

1718. Construct the Lexicographically Largest Valid Sequence

Overview

Given an integer n, we need to find the lexicographically largest sequence that satisfies all of these conditions:

- **The integer 1 occurs once in the sequence.**
- **All other integers from 2 to n occur exactly twice, and the distance between these occurrences is equal to the value of this integer.**

The distance between two integers is defined as the difference in the indices of both the integers. For example: in the array $nums = [1,2,3,1,2]$, the distance between both occurrences of 1 is given by 3.

> *A sequence a is lexicographically larger than a sequence b (of the same length) if in the first position where a and b differ, sequence a has a number greater than the corresponding number in b. For example, $[0,1,9,0]$ is lexicographically larger than $[0,1,5,6]$ because the first position they differ is at the third number, and 9 is greater than 5.*

Approach: Backtracking

Intuition

Observe the lexicographically largest sequences for smaller values of n:

- For $n = 1 : [1]$
- For $n = 2 : [2,1,2]$
- For $n = 3 : [3,1,2,3,2]$
- For $n = 4 : [4,2,3,2,4,3,1]$

Identifying an intuitive pattern for these sequences is challenging. Given that n lies in the range $1 \leq n \leq 20$, we can generate all possible valid sequences and find the lexicographically largest among them using backtracking. We'll use a recursive boolean function to determine whether the current sequence is valid. If it's not, we can terminate the recursive process early.

1718. Construct the Lexicographically Largest Valid Sequence

```
/*
   ***Recursive backtracking***
   Time complexity: O(n!)
   Space complexity: O(3n)=O(n)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      vi constructDistancedSequence(int n){

         vi ans(2*n-1,0);
         vi is_placed(n+1,0);
```

```
auto solve=[&](int index,auto& self)->bool{
    // If all positions are filled, we get our answer
    if(index==ans.size()) return true;

    // If current position is filled, pass the next position
    if(ans[index]) return self(index+1,self);

    // Try to place the largest number
    for(int e=n;e>=1;--e){
        // Determine the corresponded index of the element at the current index
        int corresponded_index=(e==1)?index:index+e;

        // If it is already placed or
        // we can not place the element in the corresponded position (if element>1)
        // then, go to lesser number
        if(is_placed[e] || (e>1 && (corresponded_index>=ans.size() || ans[corresponded_index]))) continue;

        // Place it
        is_placed[e]=1; // Mark the element as placed
        ans[index]=ans[corresponded_index]=e; // Put the element in both correct positions

        // Then pass to next positions
        if(self(index+1,self)) return true;

        // If we can not place the current element
        // Undo the placements for pervious elements
        ans[index]=ans[corresponded_index]=0;
        is_placed[e]=0;
    }

    return false;
};
```

```
        solve(0,solve);

        return ans;
    }
};
```