Documentation

1. **Problem Statement**

   Given an integer array nums, return all the different possible non-decreasing subsequences of the given array with at least two elements.

   - The order of elements must remain the same.
   - The solution must avoid duplicate subsequences.

   Constraints:

   - 1 <= nums.length <= 15
   - -100 <= nums[i] <= 100

2. **Intuition**

   Since we need all possible non-decreasing subsequences, we can use backtracking to explore all potential combinations while maintaining the order and non-decreasing condition.Using a set ensures duplicate subsequences are automatically filtered out.

3. **Key Observations**

   - A subsequence does not require consecutive elements but must preserve order.
   - Elements can be equal (non-decreasing allows equality).
   - Duplicate subsequences are possible because of repeated numbers; hence a set is needed.
   - Subsequence length must be at least 2 to be valid.

4. **Approach**

   - Start from index 0, try including or skipping each element.
   - Maintain a path list tracking the current subsequence.

- If the path has ≥ 2 elements, add it to the result set.
- Only add a number if it's ≥ last number in path.
- Use recursion (DFS) to explore all possibilities.
- Finally, convert the set to a list of lists before returning.

5. **Edge Cases**

- Array with all same numbers: Should return subsequences of all possible lengths ≥ 2.
- Array with all decreasing numbers: Only possible subsequences are between repeated numbers.
- Array with size < 2: Should return an empty list (no valid subsequences).

6. **Complexity Analysis**

Time Complexity

- Each number has two choices: include or exclude, leading to $O(2^n)$ possibilities.
- At each recursion, checking and building paths takes O(n) time.
- Thus, the total time complexity is approximately $O(n * 2^n)$.

Space Complexity

- Space for the recursion stack is O(n) in depth.
- The result set can store up to $O(2^n)$ subsequences.
- Thus, total space complexity is $O(n * 2^n)$.

7. **Alternative Approaches**

- Using extra pruning:
  Instead of using a set, use a local set at each recursion level to skip repeated numbers at the same level.
- Iterative approach:
  Build all subsequences iteratively, but handling duplicates becomes trickier and more complex than recursive DFS.

8. **Test Cases**

Input	Expected Output	Explanation
[4,6,7,7]	[[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]	All possible non-decreasing subsequences.
[4,4,3,2,1]	[[4,4]]	Only 4,4 satisfies non-decreasing condition.
[1,2,3,4]	Many subsequences like [1,2], [1,2,3], [2,3,4], etc.	All increasing subsequences.
[3,2,1]	[]	No non-decreasing subsequences of length ≥2.
[7]	[]	Not enough elements for subsequences.

9. **Final Thoughts**

This problem elegantly demonstrates backtracking with pruning and deduplication. Key learnings include:

- Proper use of a set to avoid duplicate subsequences.
- Careful management of sequence conditions (non-decreasing).
- Classic application of DFS exploration.

For better efficiency, using local sets at each recursion level can reduce unnecessary recursion when duplicates are present at the same depth.

Would you also like me to create a flowchart or tree diagram showing how the recursion works step-by-step for an example like [4,6,7]? 📈 It would be super helpful for your notes or content! 🚀 (Just say yes!)