**[202. Happy Number](https://leetcode.com/problems/happy-number/)**

/\*

\* We use two pointers:

slow moves one step at a time, fast moves two steps at a time.

If there’s a cycle, slow and fast will eventually meet. If fast becomes 1, then no cycle → happy number.

\* \*/

**491. Non-decreasing Subsequences**

Let me explain **clearly** where the **recursion level prevents duplicate elements** using a **step-by-step breakdown** with **detailed recursion states**.

**1. Identifying the Key Issue: Where Duplicates Occur?**

nums = [4, 6, 7, 7]

 The second 7 at index 3 is **a duplicate of the first 7 at index 2**.

 We must ensure that we do **not start a new subsequence with the second 7** at the same recursion level.

**2. How Are Duplicates Avoided?**

**Role of Set<Integer> used**

* This **set keeps track of numbers** used **at the current recursion level**.
* If a number is already in used, it is **skipped**.

**3. Step-by-Step Recursive Calls**

Let's simulate **how the recursion tree forms**, showing **where duplicates are skipped**.

**Initial Call**

findSubsequences([4,6,7,7])

* Calls: backtrack([4,6,7,7], 0, [], ans)

Here, we start from index 0, and our choices are {4, 6, 7, 7}.

**Recursion Level 1 (Starting with an empty list [])**

**Options: {4, 6, 7, 7}**

* **Choose 4** → [4]
* **Choose 6** → [6]
* **Choose 7** → [7]
* **Skip second 7 at index 3** (it is a duplicate at this level)

**Why?**

* Set<Integer> used = {4,6,7}
* Second 7 is skipped.

**Recursion Level 2 (Adding to [4])**

**backtrack(nums, 1, [4], ans)**

Options: {6, 7, 7}

* **Choose 6** → [4,6]
* **Choose 7** → [4,7]
* **Skip second 7 at index 3** (it is a duplicate at this level)

**Why?**

* Set<Integer> used = {6,7}
* Second 7 is skipped.

**Recursion Level 3 (Adding to [4,6])**

**backtrack(nums, 2, [4,6], ans)**

Options: {7, 7}

* **Choose 7** → [4,6,7]
* **Skip second 7 at index 3** (it is a duplicate at this level)

**Why?**

* Set<Integer> used = {7}
* Second 7 is skipped.

**Recursion Level 4 (Adding to [4,6,7])**

**backtrack(nums, 3, [4,6,7], ans)**

Options: {7}

* **Choose 7** → [4,6,7,7] (Valid!)

**Backtracking Happens**

Now we **backtrack** and remove elements one by one.

* [4,6,7,7] → [4,6,7] (Backtrack)
* [4,6,7] → [4,6] (Backtrack)
* [4,6] → [4] (Backtrack)

**4. Final Decision Tree**

[]

├── 4

│ ├── 6

│ │ ├── 7

│ │ │ ├── 7 (Valid)

│ │ │ └── (Backtrack)

│ │ └── (Skip second 7)

│ ├── 7

│ │ ├── 7 (Valid)

│ │ └── (Backtrack)

│ └── (Skip second 7)

├── 6

│ ├── 7

│ │ ├── 7 (Valid)

│ │ └── (Backtrack)

│ └── (Skip second 7)

├── 7

│ ├── 7 (Valid)

│ └── (Backtrack)

└── (Skip second 7)

**5. Recap: Where Exactly Are Duplicates Deleted?**

**Duplicate Prevention at Each Recursion Level**

At **each level of recursion**, we use:

Set<Integer> used = new HashSet<>();

* If nums[i] is already in used, **we skip it**.
* This **ensures that duplicate numbers are not added** within the same recursion level.

**6. Summary**

* **Recursive function generates subsequences.**
* **At each recursion level, a Set<Integer> tracks used numbers.**
* **Duplicates at the same level are skipped.**
* **Only unique increasing subsequences are considered.**

got confusion here Recursion Level 4 (Adding to [4,6,7]) backtrack(nums, 3, [4,6,7], ans) Options: {7} Choose 7 → [4,6,7,7] (Valid!) why it is valid ?

**1. Understanding Why [4,6,7,7] is Allowed**

At **Recursion Level 4**, we are at **index 3** and considering whether to add the last 7 in [4,6,7].

* The current sequence is **[4,6,7]**.
* We check the **next available number**, which is 7 at **index 3**.
* Since **7 is ≥ the last element (7)**, we are allowed to add it.

**2. Key Rules for Validity**

A number is **valid to add** if:

1. It maintains **increasing (or non-decreasing) order**.
2. It is **not a duplicate at the same recursion level**.

At this recursion level:

* The last element of path is 7.
* The next available number is also 7 (**equal to the last element**).
* **Since 7 >= 7**, it maintains increasing order.
* There are **no duplicates at this level** since it is the **first time we are seeing 7 at index 3**.

Thus, **[4,6,7,7] is valid**.

**3. When Would [4,6,7,7] Be Invalid?**

* **If the second 7 were already seen at the same recursion level**, it would be **skipped**.
* However, here we are **not in the same recursion level as the previous 7 selection**.
* Instead, we are in **Recursion Level 4**, after selecting [4,6,7] from **Recursion Level 3**.

**4. Key Distinction: Same Level vs. Next Level**

💡 **A duplicate number is only skipped if it appears at the same recursion level.**

**✅ Valid Case: (Moving to the next level)**

Recursion Level 3 → [4,6,7]

Recursion Level 4 → Add 7 → [4,6,7,7] ✅

* The second 7 is **not at the same recursion level** as the first 7.
* It appears **one level deeper in recursion**, so it is valid.

**❌ Invalid Case: (Same recursion level)**

If both 7s were options at the **same level**, the second one would be **skipped**:

Recursion Level 2 → [4]

Options: {6, 7, 7}

Choose 7 → [4,7]

Skip second 7 ❌ (Same level)

**5. Code Execution for [4,6,7,7]**

**✅ Valid Path:**

1. []
2. → [4]
3. → [4,6]
4. → [4,6,7]
5. → [4,6,7,7] ✅ **Valid because 7 >= 7**

**6. Final Answer**

🔹 **[4,6,7,7] is valid because:**

1. The second 7 **appears in the next level of recursion**.
2. It maintains **non-decreasing order**.
3. The **used set prevents skipping it incorrectly**.

**55. Jump Game**

**Edge Case (Why Not goal <= 0?)**

goal **never becomes negative** in this approach because:

1. We only update goal = i if i + nums[i] >= goal.
2. Since i starts from nums.length - 1 and decreases, goal **always moves leftward but never below 0**.

**Jump Game 2**

as long as end < nums.length - 1, it means:

We **haven’t yet reached** the last index. So we need to **make another jump**.

[**152. Maximum Product Subarray**](https://leetcode.com/problems/maximum-product-subarray/)

Here we have 3 cases

1) all +ve number….then multiply every number

2) Even –ve number…then multiply every number

3) odd –ve number….for this use case, we need to remove 1 –ve number then whole array will have even odd numbers. Then in this case we need to find prefix product till –ve number which we are skipping and suffix product after that number. Whoever is max from prefix and suffix will be max product.