NP class problems

Perfect clarification ✅ — you want **Class NP problems** (not NP-Hard).

That means:

> Problems where **it's easy to verify** a given solution (polynomial time) but **we don't know a polynomial-time way to find the solution**.

Okay, Abdullah — here are **two Class NP problems (not NP-Hard)** from Codeforces, LeetCode, and CodeChef — with:

- The **real problem statement (simplified from site)**
- The **reason it's NP (not NP-Hard)**
- The **algorithm (how to verify or attempt to solve)**
- **Simple code** (step-by-step, beginner friendly)

🧩 1️⃣ Codeforces — Problem 580C: *Kefa and Park*

🔗 https://codeforces.com/problemset/problem/580/C ↗

📝 Problem (simplified)

Kefa wants to visit some restaurant nodes in a park represented as a **tree** (a connected graph with no cycles).
Some vertices contain **cats**, others do not.
He starts at vertex 1.
He cannot go to a restaurant (leaf node) if there are more than **m consecutive vertices with cats** on his path from root to that restaurant.

Task: Count how many restaurants (leaf nodes) he can visit.

🎯 Why It's in Class NP

- It's a **decision/verification** problem:
  "Given a path, can Kefa reach a restaurant under the rules?"
  → You can **verify** in polynomial time if a path satisfies the $\leqslant$ m consecutive cats rule.
- It's not NP-Hard — you can check solutions quickly (depth-first search).

💡 Algorithm (DFS)

1. Build adjacency list for tree.
2. Keep count of consecutive cat nodes.
3. Start DFS from root (1).
    - If current node has a cat, increase count; else reset count to 0.
    - If count > m → invalid path.
    - If leaf node and count ⩽ m → increment answer.

🧠 Simple Code (C++)

cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, ans = 0;
vector<int> hasCat;
vector<vector<int>> adj;

void dfs(int node, int parent, int catCount) {
    if (hasCat[node]) catCount++;
    else catCount = 0;

    if (catCount > m) return; // too many cats in a row

    bool isLeaf = true;
    for (int child : adj[node]) {
        if (child != parent) {
            isLeaf = false;
            dfs(child, node, catCount);
        }
    }

    if (isLeaf) ans++; // leaf with valid path
}

int main() {
    cin >> n >> m;
    hasCat.resize(n + 1);
```

```
    adj.assign(n + 1, {});

    for (int i = 1; i <= n; i++) cin >> hasCat[i];

    for (int i = 0; i < n - 1; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(1, -1, 0);
    cout << ans;
}
```

✅ **Complexity:** O(n)
✅ **Verification (NP-style):** Given a path, you can check if it satisfies ⩽ m consecutive cats in linear time.

🧩 2️⃣ LeetCode — Problem 46: *Permutations*

🔗 https://leetcode.com/problems/permutations/ ↗

📝 Problem (simplified)

Given an array of distinct numbers `nums`, return all possible **permutations**.

Example:

```
csharp

Input: nums = [1,2,3]
Output:
[
 [1,2,3],
 [1,3,2],
 [2,1,3],
 [2,3,1],
 [3,1,2],
 [3,2,1]
]
```

🎯 Why It's Class NP

- The **decision version**:
  "Is there a permutation satisfying some property (e.g., sum of adjacent differences ⩽ k)?"
  — you can verify a given permutation in polynomial time.
- It's **not NP-Hard** — generating or verifying a permutation is easy (factorial complexity only due to enumeration).

💡 Algorithm (Backtracking)

1. Start with empty list `current`.
2. At each step, pick a number not used yet.
3. Recurse until permutation complete → store result.

🧠 Simple Code (Python)

python

```python
def permute(nums):
    res = []
    used = [False] * len(nums)

    def backtrack(path):
        if len(path) == len(nums):
            res.append(path[:])  # add copy
            return
        for i in range(len(nums)):
            if not used[i]:
                used[i] = True
                path.append(nums[i])
                backtrack(path)
                path.pop()
                used[i] = False

    backtrack([])
    return res
```

```
# Example
print(permute([1, 2, 3]))
```

✅ **Complexity:** O(n × n!)

✅ **Verification:** If someone gives you a permutation, checking if it's valid (contains all elements exactly once) is O(n).

🧩 3️⃣ CodeChef — Problem: *SUBINC (Count Increasing Subsequences)*

🔗 https://www.codechef.com/problems/SUBINC ↗

📝 Problem

Given an array `A` of length n, count the number of **non-decreasing contiguous subarrays**.

🎯 Why It's Class NP

- Verifying a proposed subarray is non-decreasing is O(n).
- It's not NP-Hard — can be solved efficiently using simple DP.

💡 Algorithm

1. Use a dynamic programming array `dp[i]` = number of non-decreasing subarrays ending at `i`.
2. If `A[i] >= A[i-1]`, then `dp[i] = dp[i-1] + 1`, else `dp[i] = 1`.
3. The answer = sum of all `dp[i]`.

🧠 Simple Code (Python)

```python
def countSubInc(A):
    n = len(A)
    dp = [1]*n
    ans = 1
    for i in range(1, n):
```

```
        if A[i] >= A[i-1]:
            dp[i] = dp[i-1] + 1
        ans += dp[i]
    return ans

# Example
print(countSubInc([1, 2, 2, 4]))  # Output: 9
```

✅ **Verification (NP style):** Checking a single subarray is non-decreasing = O(n).

✅ Summary Table

Platform	Problem	NP Feature	Verification Time	Solving Idea
Codeforces	Kefa and Park	Path validation	O(n)	DFS
LeetCode	Permutations	Permutation validity	O(n)	Backtracking
CodeChef	SUBINC	Check subarray order	O(n)	Dynamic Programming

---

Would you like me to make **slides (with visuals + short pitch)** for these three NP examples for your presentation?
I can organize it as **Slide 1: NP Definition → Slide 2–4: Each Problem** → Slide 5: Conclusion.