**🚀 String Hashing & Hash Table Applications**

**🔹 28. Implement strStr() (Rolling Hash Polynomial) – *Easy***

**Concept:**

We want to **find the first occurrence of a substring (needle) in a larger string (haystack)**.  
Naive approach is O(n·m), where we check each substring directly.  
👉 Instead, we use **Rolling Hash (Rabin-Karp)** with polynomial hashing.

**Rolling Hash Polynomial Formula:**

For a string s[0..m-1],

H = (s[0]·p^(m-1) + s[1]·p^(m-2) + ... + s[m-1]·p^0) % M

* p = base (prime, e.g., 31 or 53)
* M = large prime modulus (e.g., 1e9+7)

**Why Rolling Hash?**

* Lets us compute the next substring hash efficiently (O(1)).
* Avoids recomputing a fresh hash at every shift.

**Collision Handling:**

* Same hash ≠ always same substring (possible *collision*).
* Always **verify match by direct comparison**.
* Or use **double hashing** (two independent (p, M) pairs) to lower probability of false matches.

✅ **Takeaway:** Rabin–Karp with rolling hash = Efficient substring search.  
Time: **O(n + m)** expected, vs **O(n·m)** for naive.

**🔹 49. Group Anagrams – *Medium***

**Concept:**

Group words so that **rearrangements (anagrams) fall into the same bucket**.

Example:

["eat","tea","tan","ate","nat","bat"]  
→ [["eat","tea","ate"], ["tan","nat"], ["bat"]]

**Approaches:**

**1️⃣ Sorting Method (Simple & Reliable)**

* Sort each word alphabetically → use sorted string as key.
* Example: "eat" → "aet", "tea" → "aet".
* Dictionary groups: { "aet": ["eat","tea","ate"], "ant":["tan","nat"], ... }

⏱ Time: **O(N \* K log K)**

**2️⃣ Prime-Product Hashing (Math Trick ⚡)**

* Map each character → unique prime number (a=2, b=3, c=5,...).
* Word’s hash = product of primes of its characters.
* Example:

eat = 2\*11\*71  
tea = 71\*11\*2  
Both = 1562 → grouped together

⏱ Time: **O(N \* K)**  
⚠️ Risk: Integer overflow. Use modular math.

**3️⃣ Frequency Count Hashing (Most Efficient)**

* Represent each word as **26-length frequency vector**.
* Example: "eat" → [1,0,0,...,1,0,...,1,...]
* Use tuple of counts as hash key.

⏱ Time: **O(N \* K)**  
✅ No collisions — unique representation of anagrams.

**🎯 Key Learnings**

* **Rolling Hash (Rabin-Karp):** Substring search optimized with rolling computation.
* **Hashing in Anagrams:** Several strategies:
  + Sorting (easy, slower)
  + Prime-product (mathematical, risky for overflow)
  + Frequency vector (optimal, no collisions)
* **Hash tables:** Core to both problems → efficient lookups, collision handling.

✨ This is now clean, workshop-style content.  
Would you like me to **convert this into an HTML-ready doc styled with code blocks, diagrams, and highlights** (for teaching/presentation), or keep it as plain structured notes for study?