Day 5 : dsa babua pattern course :- two pointers problem

Problem 1 : 📘 Strobogrammatic Number

💡 Definition (Kya hota hai Strobogrammatic Number?)

Aisa number **jo 180° rotate karne par same dikhe**, usko **Strobogrammatic Number** bolte hain.
→ 180° rotation ka matlab:
Top-ultra se ghumane par digits apne valid rotated pair ke equal rehne chahiye.

✔ Valid Strobogrammatic Digits & Their Pairs:

Digit	After 180° Rotation	Valid?
0	0	✔
1	1	✔
8	8	✔
6	9	✔
9	6	✔

❌ **Invalid Digits (kabhi Strobogrammatic nahi hote):**

`2, 3, 4, 5, 7`

Yeh rotate hone ke baad koi valid digit nahi banate.

🧠 Concept / Intuition (Logic samajhna sabse important)

Aapko ek **string of digits** milta hai.
Aapko check karna hai → kya yeh number **rotate hone ke baad same** ban sakta hai?

Two Pointer Approach:

1. i = 0 left se start
2. j = n - 1 right se start
3. Check:

   mapping[s[i]] == s[j]
4.

5. Agar kisi point pe match nahi hua → **not strobogrammatic**

Mapping:

```
0 → 0
1 → 1
8 → 8
6 → 9
9 → 6
```

🔍 Example 1

Input: "69"

Left = '6' → rotated = '9'
Right = '9'
✔ match
So "69" is **strobogrammatic**

🔍 Example 2

Input: "818"

Indices:
0 → 8
1 → 1
2 → 8
Pairs:
8 ↔ 8 ✔
1 ↔ 1 ✔
Center element always valid (0,1,8) ✔
So → **Yes**

🔍 Example 3

Input: "12"

Left = 1 → maps to 1
Right = 2 → but 2 is invalid
❌ Not strobogrammatic

🧩 Algorithm (Step-by-step)

1. Create a **map** of valid strobogrammatic pairs
2. Use two pointers i and j
3. For each pair:
    - Check if s[i] exists in map
    - Check if map[s[i]] == s[j]
4. If koi mismatch → return false
5. Loop end hone par → return true

⏱️ Time & Space Complexity

⏳ Time: O(n)

Har iteration me i++ aur j--
→ ek hi pass me kaam done

📦 Space: O(1)

Map me sirf 5 entries → constant space

📝 Final Code (C++ Two Pointer Approach)

```cpp
class Solution {
public:
    bool isStrobogrammatic(string s) {
        unordered_map<char, char> mp = {
            {'0','0'}, {'1','1'}, {'8','8'},
            {'6','9'}, {'9','6'}
        };

        int i = 0, j = s.size() - 1;

        while(i <= j) {
            char L = s[i];
            char R = s[j];

            if(mp.find(L) == mp.end()) return false; // invalid digit

            if(mp[L] != R) return false; // pair mismatch

            i++;
            j--;
        }
        return true;
    }
};
```

✔ Revision Points

- Valid digits: **0,1,8,6,9**
- 6 ↔ 9 pair important
- Invalid digits: **2,3,4,5,7**
- Two pointers + mapping
- Time: O(n), Space: O(1)

Problem 2: Append Characters to String to Make Subsequence — 🎯 Problem Goal

Aapko **two strings S (source)** aur **T (target)** diye gaye hain.
Aapko **minimum characters** batाने हैं **jo S ke end me append karne padेंगे**, tāki **T, S ka subsequence ban jaye**.

💡 What is a Subsequence? (Quick Recap)

Subsequence = string created by
→ **Deleting ANY characters**

→ **Without changing the order**
Example:
S = "babuaDS"
Valid subsequences: "bauS", "bbu", "babua"
Order same rahe = subsequence
(positions skip ho sakte hain → delete allowed)

🧠 Core Idea / Intuition (Two Pointer)

Aapko check karna hai ki **T ka kitna part S me match ho sakta hai**
→ **starting se**, **order maintain** karte हुए.
Two Pointers:

- i → S ke characters traverse karega
- j → T ke characters match karega

Matching Rules:

- Agar S[i] == T[j] → **match → j++**
- Warna → S[i] ko ignore (delete) → **i++**
- Process continue until S exhausts.

Last me:
👉 **T ka jitna part match nahi hua = append karna padega**
Formula:

```
answer = T×length − j
```

🔍 Example Explained

S = "coaching"
T = "coding"

Matching step-by-step:

S	T	Match?	j (progress in T)
c	c	✔	1
o	o	✔	2
a	d	✘	2
c	d	✘	2
h	d	✘	2
i	d	✘	2
n	d	✘	2
g	d	✘	2

👉 S me "d" kabhi mila hi nahi
👉 T ke matched part = "co" → (length = 2)
👉 Remaining = "ding" → 4 characters
So answer = **4**

🧩 Algorithm — Clean Steps

1. Two pointers:
   - i = 0 (for S)
   - j = 0 (for T)
2. While i < n and j < m:
   - If characters match → j++
   - Always move i++
3. End me j = number of matched characters.
4. Append needed = m - j
   (jo T ka remaining part bacha hai)

⏱️ Time Complexity

`O(n + m)`
→ Ek pass S par, aur T pointer ek direction me badhta hai.

📦 Space Complexity

`O(1)`
→ No extra space.

📝 Final C++ Code (Clean)

cpp

```cpp
class Solution {
public:
    int appendCharacters(string s, string t) {
        int i = 0, j = 0;

        while(i < s.size() && j < t.size()) {
            if(s[i] == t[j]) j++;
            i++;
        }

        return t.size() - j;
    }
};
```

✔ Revision Notes (5-sec scan)

- Subsequence → delete allowed, order not change
- Two pointers → S scan, T match
- Last matched index = j
- Append = t×length - j
- Simple linear problem
- Mostly asked in interviews as warm-up

Problem 3 : 1650 — Lowest Common Ancestor of a Binary Tree III (Tree nodes have parent pointer)

🎯 Problem Goal

Aapko **binary tree** diya hai jisme har node ke paas

- left
- right
- parent
  pointer diya hua hai.

Aapko do nodes **p** aur **q** diye gaye hain.
👉 **Lowest Common Ancestor (LCA)** return karna hai.

💡 What is LCA? (Simple Definition)

For two nodes **p** and **q**:
LCA = wo lowest (i.e., sabse neeche wala) node jiske subtree me p aur q dono descendent ho.
Node can be a descendant of itself.
Example:

```
          5
        /   \
       3     6
      / \
     2   1
```

- LCA(2,1) = **3**
- LCA(3,1) = **3**
- LCA(3,6) = **5**

🧠 Approach 1 — Using Set (Brute but Good)

Intuition:

1. p se **parent chain** upar jao aur sab nodes ek **set** me daal do.
2. Ab q se **upar jao**:
   - jo pehla node **set me mil jaye** → that is LCA.

Why it works?

Kyuki dono nodes ka **first common parent** = LCA.

✔ Algorithm Steps (Approach 1)

1. Create a set.
2. Traverse upward from p:
   - Insert p into set.
   - Move p = p.parent
3. Traverse upward from q:
   - If (q in set) → return q
   - Else q = q.parent

⏱ Complexity:

```
Time:  O(height of tree)
Space: O(height)   // set me upar jaate nodes store honge
```

✔ Code (Approach 1)

```
class Solution {
public:
    Node* lowestCommonAncestor(Node* p, Node* q) {
        unordered_set<Node*> st;

        while(p != nullptr) {
            st.insert(p);
            p = p->parent;
        }

        while(q != nullptr) {
            if(st.count(q)) return q;
            q = q->parent;
        }

        return nullptr;
    }
};
```

⭐ Approach 2 — Two Pointer Trick (Best Approach)

Same technique as "Intersection of Two Linked Lists".

🧠 Beautiful Intuition

Parent pointers → nodes form **linked lists** going upward:
Example:

```
p → parent → parent → ... → root → null
q → parent → parent → ... → root → null
```

These two upward chains behave like **two linked lists**.
Like intersection of linked lists:

```
distance(p to root) = A
distance(q to root) = B
common tail path = C
```

If you traverse like:

```
p1 = p
```

```
q1 = q
```
Each step:

```
p1 = (p1 == NULL ? q : p1.parent)
q1 = (q1 == NULL ? p : q1.parent)
```
👉 Guaranteed that
p1 and q1 will meet exactly at LCA.
Why?
Both pointers travel:

```
A + C + B + C = same total distance
```

✔ Algorithm Steps (Approach 2)

1. Create pointers p1 = p and q1 = q.
2. Loop until p1 == q1:
    - Move p1 upward:

      p1 = p1 == NULL ? q : p1.parent
    -
    - Move q1 upward:

      q1 = q1 == NULL ? p : q1.parent
    -
3. When both meet → that's LCA.

🔥 Why This Approach Is Amazing?

- **No extra space**
- Elegant
- Fast
- Works like linked list intersection logic
- Guaranteed meeting point = LCA

⏱ Complexity

```
Time:  O(height)
Space: O(1)
```

✔ Final Best Code (Approach 2)

```cpp
class Solution {
public:
    Node* lowestCommonAncestor(Node* p, Node* q) {
        Node* p1 = p;
        Node* q1 = q;

        while(p1 != q1) {
            p1 = (p1 == nullptr ? q : p1->parent);
            q1 = (q1 == nullptr ? p : q1->parent);
        }

        return p1;  // or q1
    }
};
```

📝 Revision Notes (5-sec scan)

- LCA = lowest node having p & q as descendants
- Approach 1 → store parents of p → walk q upwards
- Approach 2 → two-pointer intersection trick
- Best approach = **Two Pointer**
- Space = O(1)