STRINGS:-

🌟 Pattern 1: Two Pointers

✅ **Idea:**

- Tum ek string (ya array) ke *2 indexes* maintain karte ho → **start** aur **end**.
- In dono se ek window ya subarray ya substring banate ho.
- Tum in pointers ko move karke check karte ho ki condition satisfy ho rahi hai ya nahi.

💥 Kab Use Karte Hain?

Problem Keyword Me Ho:	Pattern Idea
"without repeating"	Sliding window with set/map
"longest substring"	Two pointer + window size
"subarray/subsequence with sum/condition"	Two pointer window
"palindrome check"	Left = 0, Right = n - 1
"reverse in place"	Swap using left, right

⭐ "Jab bhi question mein start aur end ki boundary ya substring ki window ki baat ho, two-pointer approach almost hamesha kaam aati hai."

✅ Basic Idea of 2 Pointer:

```
start = 0
end = n - 1

while(start < end):
    check / swap / compare
    move start/end
```

🔎 Key Use Cases in Strings

1️⃣ Check if String is Palindrome

✅ **Brute Force Idea:**

- Pura string reverse karo aur compare karo original ke saath.

✅ **Time Complexity:**

- Brute: O(N)
- Optimized (two-pointer): O(N)

✅ **Note:**

⭐ "Interview mein hamesha two-pointer wali in-place approach likho – extra space nahi lagta."

Example Code:

```
bool isPalindrome(string s) {
    int start = 0, end = s.size() - 1;
    while (start < end) {
        if (s[start] != s[end]) return false;
        start++;
        end--;
    }
    return true;
}
```

✅ Pattern: Two pointer, move towards center.

2️⃣ Reverse String in Place

✅ **Brute Force Idea:**

- Ek naya array banao aur elements ko ulta fill karo.

✅ **Time Complexity:**

- Brute: O(N) time + O(N) space

- Optimized: O(N) time + O(1) space

✅ **Note:**

⭐ "Interview mein in-place swapping wali approach best hoti hai – space optimal."

Example Code:

```
void reverseString(vector<char>& s) {
    int left = 0, right = s.size() - 1;
    while(left < right){
        swap(s[left], s[right]);
        left++;
        right--;
    }
}
```

✅ Pattern: Two pointer, swap.

3️⃣ Remove Palindrome (with cleaning non-alnum and ignoring case)

✅ **Brute Force Idea:**

- Cleaned string bana ke reverse karo aur compare karo.

✅ **Time Complexity:**

- Brute: O(N) time + O(N) space
- Optimized (in-place two-pointer): O(N) time + O(1) space

✅ **Note:**

⭐ "Jab question mein non-alphanumeric ya case ignore bola ho, clean karna zaruri hai – isse interview mein edge cases bhi handle hote hain."

Example Code:

```
bool isPalindrome(string s) {
    int start = 0, end = s.size() - 1;
```

```
    while (start < end) {
        if (!isalnum(s[start])) { start++; continue; }
        if (!isalnum(s[end])) { end--; continue; }
        if (tolower(s[start]) != tolower(s[end])) return false;
        start++;
        end--;
    }
    return true;
}
```

✅ Pattern: Two pointer with cleaning.

4️⃣ Reverse Words in a String

✅ **Brute Force Idea:**

- Split karke words ka array banao → reverse order mein join karo.

✅ **Time Complexity:**

- Brute: O(N) time + O(N) space
- Optimized: O(N) time + O(1) space (in-place double reverse)

Example Code:

```
string reverseWords(string s) {
    reverse(s.begin(), s.end());
    int i = 0, n = s.size();
    string result;

    while (i < n) {
        while (i < n && s[i] == ' ') i++;
        int start = i;
        while (i < n && s[i] != ' ') i++;
        int end = i;

        if (start < end) {
```

```
            string word = s.substr(start, end - start);
            reverse(word.begin(), word.end());
            if (!result.empty()) result += " ";
            result += word;
        }
    }
    return result;
}
```

✅ 2-pointer for each word boundaries.

✅🧠 Pattern 1 Core Points to Remember

⭐ Always keep TWO indexes → start, end.

⭐ Define **window**.

⭐ Move them to grow/shrink window based on conditions.

✅ **Video Tip:**

⭐ "Jab tak tumhare dimaag mein 'start' aur 'end' ki picture nahi bani → two-pointer nahi samjha."

🎯 Pattern 1 Cheat Sheet

Goal	Pointer Use
Check Palindrome	Move inward comparing
Reverse String	Swap start/end
Reverse Words in Sentence	Reverse all, then each word
Sliding Window Longest Substring	Start/End → grow/shrink window
Check Anagram	Sort or freq array (not pure 2-pointer)

✅ **Pattern 1 Key Rule**

> ⭐ "When question says 'Check start and end of something' or 'Process whole string from both ends' → Think Two Pointers."

✅ **Pro Interview Tip:**

⭐ "Interview mein pehle brute force socho → phir two-pointer wala optimized approach likho – interviewer tumhari thinking process dekhta hai."

✅ PATTERN 2: Stack

✅ **Idea:**

- Open/close balance check
- Remove matching pairs
- Track previous characters

✅ **Keywords:**

- "parentheses"
- "remove duplicates"
- "balanced"

⭐ Example 1: Remove Outermost Parentheses

🧠 **Problem:**

`Input: "(()())(())"`

✅ Remove outermost → `"()()()"`

✅ **Intuition:**

- Count how deep you are in nesting
- Don't add outermost (when depth = 1)

✅ **Code:**

```
string removeOuterParentheses(string s) {
    string res;
    int depth = 0;
```

```
    for(char ch : s) {
        if(ch == '(') {
            if(depth > 0) res += '(';
            depth++;
        } else {
            depth--;
            if(depth > 0) res += ')';
        }
    }
    return res;
}
```

✅ **Explanation:**

- `depth` counts open brackets
- Ignore adding when depth is 0 or 1 (outermost)

⭐ Example 2: Remove Adjacent Duplicates

🧠 **Problem:**

`Input: "abbaca"`

✅ Output: `"ca"`

✅ **Intuition:**

- Stack holds characters
- Remove pair when same

✅ **Code:**

```
string removeDuplicates(string s) {
    stack<char> st;
    for(char ch : s) {
        if(!st.empty() && st.top() == ch) st.pop();
        else st.push(ch);
    }
    string res = "";
```

```
    while(!st.empty()) {
        res = st.top() + res;
        st.pop();
    }
    return res;
}
```

✅ **Explanation:**

- Push if not same as top
- Pop if same (remove pair)

✅ ⭐ When to think Stack?

- Matching symbols
- Remove pairs
- Balanced structures

✅ PATTERN 3: HashMap / Frequency Count

✅ **Idea:**

- Count characters
- Track mappings
- Compare counts

✅ **Keywords:**

- "anagram"
- "frequency"
- "isomorphic"

✅ **Brute Force Idea:**

- Sort and compare

✅ **Time Complexity:**

- Brute: O(N log N)
- Optimized HashMap: O(N)

⭐ Example 1: Check Anagram(dono string ke characters equal times repeat hone chiy)

🧠 **Problem:**

`s = "anagram", t = "nagaram"`

✅ Output: true

✅ **Code:**

```
bool isAnagram(string s, string t) {
    if(s.size() != t.size()) return false;
    vector<int> count(26, 0);
    for(char ch : s) count[ch - 'a']++;
    for(char ch : t) count[ch - 'a']--;
    for(int c : count) if(c != 0) return false;
    return true;
}
```

✅ **Explanation:**

- Count letters in s
- Subtract letters in t
- All zeros = anagram

⭐ Example 2: Isomorphic Strings (dono string ke corresponding character same character se hi relate krne chiy)

🧠 **Problem:**

`s = "egg", t = "add"`

✅ Output: true

✅ **Code:**

```
bool isIsomorphic(string s, string t) {
    unordered_map<char, char> m1, m2;
    for(int i = 0; i < s.size(); i++) {
        if((m1.count(s[i]) && m1[s[i]] != t[i]) ||
           (m2.count(t[i]) && m2[t[i]] != s[i]))
            return false;
        m1[s[i]] = t[i];
        m2[t[i]] = s[i];
    }
    return true;
}
```

✅ **Explanation:**

- Map s→t and t→s
- Check consistency

⭐ Example 3: Sort Characters by Frequency

🧠 **Problem:**

`s = "tree"`

✅ Output: `"eetr"` or `"eert"`

✅ **Code:**

```
string frequencySort(string s) {
    int freq[128] = {0};

    // Count frequency of each character
    for(char ch : s) freq[ch]++;
```

```
    // Result string
    string res = "";

    // Keep adding highest frequency char until all done
    while(!s.empty()) {
        char maxChar = 0;
        for(char c : s) {
            if(freq[c] > freq[maxChar]) maxChar = c;
        }
        res.append(freq[maxChar], maxChar);

        // Remove all occurrences of maxChar from s
        s.erase(remove(s.begin(), s.end(), maxChar), s.end());
        freq[maxChar] = 0;
    }
    return res;
}
```

✅ **Explanation:**

- Count frequencies
- Sort by count

✅ **When to think HashMap:**

- Count letters
- Compare mappings
- Frequency-based sorting

✅ PATTERN 4: Sliding Window

✅ **Idea:**

- Maintain window with start, end

- Check condition while moving window

✅ **Keywords:**

- "longest substring"
- "without repeating"
- "at most k distinct"

✅ **Brute Force Idea:**

- Check all substrings

✅ **Time Complexity:**

- Brute: $O(N^2)$
- Optimized: O(N)

⭐ Example 1: Longest Substring Without Repeat

🧠 **Problem:**

`Input: "abcabcbb"`

✅ Output: 3 ("abc")

✅ **Code:**

```
int lengthOfLongestSubstring(string s) {
    unordered_set<char> seen;
    int start = 0, end = 0, maxLen = 0;
    while(end < s.size()) {
        if(!seen.count(s[end])) {
            seen.insert(s[end]);
            maxLen = max(maxLen, end - start + 1);
            end++;
        } else {
            seen.erase(s[start]);
            start++;
        }
    }
```

```
    return maxLen;
}
```

✅ **Explanation:**

- Add unique chars
- Remove when repeat
- Track window length

⭐ Example 2: Longest Substring with at most K distinct

🧠 **Problem:**

`Input: "eceba", k=2`

✅ Output: 3 ("ece")

✅ **Code:**

```
int lengthOfLongestSubstringKDistinct(string s, int k) {
    unordered_map<char,int> count;
    int start = 0, maxLen = 0;
    for(int end = 0; end < s.size(); end++) {
        count[s[end]]++;
        while(count.size() > k) {
            count[s[start]]--;
            if(count[s[start]] == 0) count.erase(s[start]);
            start++;
        }
        maxLen = max(maxLen, end - start + 1);
    }
    return maxLen;
}
```

✅ **Explanation:**

- Maintain ≤ k distinct in window

- Shrink if over k

✅ **When to think Sliding Window?**

- Substring with condition
- Count distinct
- Longest/shortest satisfying window

✅ PATTERN 5: Sorting

✅ **Idea:**

- Sort chars/strings
- Compare after sorting

✅ **Brute Force Idea:**

- Compare unsorted strings

✅ **Time Complexity:**

- O(N log N)

✅ **Examples:**

- Check anagram (sort)
- Longest common prefix (sort array of strings)

✅ **Example: Longest Common Prefix**

```
string longestCommonPrefix(vector<string>& strs) {
    sort(strs.begin(), strs.end());
    string first = strs[0], last = strs.back();
    int i = 0;
    while(i < first.size() && i < last.size() && first[i] == last[i]) i++;
    return first.substr(0, i);
}
```

✅ **Explanation:**

- Sorting brings similar prefixes together

✅ Pattern 6: Expand Around Center

✅ Idea:

- Check palindromes by expanding from center

✅ Brute Force Idea:

- All substrings check karo

✅ Time Complexity:

- Brute: $O(N^3)$
- Optimized: $O(N^2)$

✅ Example: Longest Palindromic Substring

```
string longestPalindrome(string s) {
    int start = 0, end = 0;
    for(int i = 0; i < s.size(); i++) {
        int len1 = expand(s, i, i);//odd
        int len2 = expand(s, i, i+1);//even
        int len = max(len1, len2);
        if(len > end - start + 1) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substr(start, end - start + 1);
}

int expand(string &s, int l, int r) {
    while(l >= 0 && r < s.size() && s[l] == s[r]) {
        l--;
        r++;
    }
```

```
    return r - l - 1;
}
```

✅ **Explanation:**

- Try to grow palindrome from every center

✅⭐ Pattern 7: Observation / Greedy Insight

✅ 💡 **Idea:**

- Kabhi kabhi koi stack / two-pointer / hashmap nahi lagta.
- Question *hi* hint deta hai → "*kya observe ho raha hai?*"
- Bas woh *simple rule* ya *greedy choice* nikal ke apply karo.

✅ **Keywords:**

✅ "observe pattern"

✅ "simple rule"

✅ "greedy choice"

✅ "mathematical property"

✅ **Brute Force:**

- Usually O(N)

✅1️⃣ Largest Odd Number in a String

🧭 **Problem:**

> Given num = "3542708"✅ Find the *largest odd-ending substring*.

✅ **Observation:**

- Jo bhi suffix odd digit pe end kare wo valid.

- Right se scan → first odd → return prefix till there.

✅ **Code:**

```
string largestOddNumber(string num) {
    for(int i = num.size() - 1; i >= 0; i--) {
        if((num[i] - '0') % 2 != 0) return num.substr(0, i + 1);
    }
    return "";
}
```

✅ **Explanation:**

- Check last odd digit.
- All digits before it are valid.

⭐ No stack, no two pointers → just **observation**!

✅ 2️⃣ Maximum Nesting Depth of Parentheses

🧭 **Problem:**

> "(1+(2*3)+((8)/4))+1"✅ Output: 3

✅ **Observation:**

- Count open '(' → increase depth.
- Track max depth.
- No need for stack contents!

✅ **Code:**

```
int maxDepth(string s) {
    int depth = 0, maxD = 0;
    for(char ch : s) {
        if(ch == '(') {
            depth++;
            maxD = max(maxD, depth);
```

```
        } else if(ch == ')') {
            depth--;
        }
    }
    return maxD;
}
```

✅ **Explanation:**

- Just track how deep you go.
- Max depth = answer.

⭐ Very simple **observation**.

✅ 3️⃣ Roman to Integer

🧭 **Observation:**

- If smaller before bigger → subtract.
- Else → add.

✅ **Code Idea:**

```
I V X L C D M
1 5 10 50 100 500 1000
```

```
int romanToInt(string s) {
    unordered_map<char, int> val{{'I',1},{'V',5},{'X',10},{'L',50},{'C',100},{'D',50
0},{'M',1000}};
    int res = 0;
    for(int i = 0; i < s.size(); i++) {
        if(i + 1 < s.size() && val[s[i]] < val[s[i+1]])
            res -= val[s[i]];
        else
            res += val[s[i]];
    }
```

```
    return res;
}
```

✅ **Explanation:**

- Just check neighbor → subtract if needed.

✅ ⭐ **Pattern 7 Summary:**

> Whenever question says: "find X, given these simple rules" → think observation-based!✅ No extra data structure✅ Just greedy / property

---

✅✅ ⭐ Final Line for Notes:

> ⭐ Observation-Based Pattern:*Use question's rules directly!No fancy data structure — just logic/greedy choice.*

✅ **Examples:**

- Largest odd number in string
- Maximum nesting depth
- Roman to Integer
- Atoi parsing