SLIDING WINDOW/SOME TWO POINTER PROBLEMS.

1) Best Time to Buy and Sell Stock (121)

- `low` → buy day, `high` → sell day.
- If `prices[high] < prices[low]` → found better buying price → `low = high`.
- If `prices[high] >= prices[low]` → calculate profit.
- Update `maxProfit` whenever profit is positive.
- Always move `high` forward to explore future days.
- Idea: keep track of lowest so far, try selling at every future higher price.

.

```java
class Solution {
    public int maxProfit(int[] prices) {
        int low = 0;
        int high = 1;
        int maxp = 0;
        while(high < prices.length)
        {
            int calc = prices[high] - prices[low];
            if(calc < 0)
            {
                low = high;
                high++;
            }
            if(calc >= 0)
            {
                maxp = Math.max(maxp, calc);
                high++;
            }
        }
        return maxp;
    }
}
```

2) LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS. (3) (VERY IMPORTANT, U'VE BEEN DOING MISTAKE HERE)

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int low = 0;
        int high = 0;
        int maxlen = 0;
        Map<Character, Boolean> map = new HashMap<>();
        while(high < s.length())
        {
            while(map.containsKey(s.charAt(high)))
            {
                map.remove(s.charAt(low));
                low++;
                continue;
            }
            map.put(s.charAt(high), true);
            maxlen = Math.max(maxlen, high-low+1);
            high++;
        }
        return maxlen;
    }
}
```

Low and high points to initial.
Eg: if "abcabcbb" low points to a, high points to a. Len = 1
Low points to a, high points to b. Len = 2
Low points to a, high points to c. len = 3
Low points to a, high points to a, both are same!
Thus, we reduce the window from left till we get a valid substring. Then, low points to a (3rd index) and high also points to same one. And the cycle repeats till high reaches the end.

SIMPLE IDEA.

- `low` → start of window, `high` → end of window.
- Expand window by moving `high`.
- If duplicate appears → window becomes invalid.

- Remove characters from `low` until duplicate is gone (`while` loop).
- Once valid, update `maxLen = high - low + 1`.
- Idea: Expand when valid, shrink when duplicate appears.

3) PERMUTATION IN A STRING (NOT OPTIMAL, BUT WORKS ON LEETCODE AND EASIER TO REMEMBER)

```java
class Solution {
    public boolean checkInclusion(String s1, String s2) {

        if (s1.length() > s2.length()) return false;

        int ws = s1.length();
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < ws; i++) {
            sb.append(s2.charAt(i));
        }

        if (sort(sb.toString()).equals(sort(s1))) {
            return true;
        }

        for (int i = ws; i < s2.length(); i++) {
            sb.deleteCharAt(0);
            sb.append(s2.charAt(i));

            String temp1 = sort(sb.toString());
            String temp2 = sort(s1);
            if (temp1.equals(temp2))

                return true;

        }


        return false;

    }


    public String sort(String s) {

        char[] arr = s.toCharArray();

        Arrays.sort(arr);

        return new String(arr);

    }

}
```

🔥 Idea

- Permutation ⇒ same characters, same frequency.
- Window size = `s1.length()`.
- For each window in `s2`:
    - Sort window
    - Sort `s1`
    - Compare.
- If equal → permutation exists.

⚠️ Traps

- ❗ Forgetting to check first window.
- ❗ Not handling `s1.length() > s2.length()`.
- ❗ Sorting `s1` inside loop repeatedly (wasteful).
- ❗ Very slow for large inputs (O(n * m log m)).

4) MINIMUM SIZE SUBARRAY SUM (209)

```java
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int low = 0;
        int high = 0;
        int sum = 0;
        int maxlen = Integer.MAX_VALUE;
        while(high < nums.length)
        {
            sum = sum + nums[high];
            while(sum >= target)
            {
                sum = sum - nums[low];
                maxlen = Math.min(maxlen, high - low + 1);
                low++;
            }
            high++;
        }
        if(maxlen == Integer.MAX_VALUE)
        {
            return 0;
        }
        else return maxlen;
    }
}
```

🟢 209 - Minimum Size Subarray Sum

- Expand window → add to `sum`.
- When `sum ≥ target` → shrink from left.
- Update `minLen` while shrinking.
- Continue till end.

⚠️ traps

- Init `minLen = MAX_VALUE`.
- Return `0` if never updated.
- Works because all numbers are positive.