Sliding Window

- Contains duplicate II

```
class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        HashMap<Integer,Integer> map=new HashMap<>();

        for(int i=0;i<nums.length;i++)
        {
            if(map.containsKey(nums[i])==true &&
Math.abs(map.get(nums[i])-i)<=k)
            {
                    return true;
            }
            map.put(nums[i],i);
        }
        return false;
    }
}
```

Or

```
class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        HashSet<Integer> window=new HashSet<>();

        for(int i=0;i<nums.length;i++)
        {
            // if number already in window → duplicate found
            if(window.contains(nums[i])==true)
            {
                    return true;
            }

            // add current number to window
            window.add(nums[i]);

            // maintain window size ≤ k
            if(window.size()>k)
                window.remove(nums[i-k]);
        }
        return false;
    }
}
```

- Best time to buy and sell stock

```
class Solution {
    public int maxProfit(int[] prices) {

        int[] rightmax=new int[prices.length];

        rightmax[prices.length-1]=0;

        for(int i=prices.length-2;i>=0;i--)
        {
            rightmax[i]=Math.max(prices[i+1],rightmax[i+1]);
        }

        int profit=0;

        for(int i=0;i<prices.length;i++)
        {
            if(prices[i]<rightmax[i])
            {
                int  x=Math.abs(prices[i]-rightmax[i]);
                profit = (profit<x) ? x : profit;
            }
        }

        return profit;
    }
}
```

Or

```
class Solution {
    public int maxProfit(int[] prices) {

        int l=0,r=1,profit=0;

        while(r<prices.length)
        {

            // profit if we sell at 'right' and buy at 'left'
            if(prices[l]<prices[r])
            {
                int pr=prices[r]-prices[l];
                profit=Math.max(profit,pr);
            }

            else
                l=r;

            r++;
        }

        return profit;
    }
}
```

- Longest substring without repeating character

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        HashSet<Character> set=new HashSet<>();

        int l=0,r=0,c=0;

        while(r<s.length())
        {
            char ch=s.charAt(r);

            // if character not in set, add and expand window
            if(!set.contains(ch))
            {
                set.add(ch);
                c=Math.max(c,(r-l+1));
                r++;
            }

            // if duplicate found, shrink window from left
            else
            {
                set.remove(s.charAt(l));
                l++;
            }
        }

        return c;
    }
}
```

- Longest repeating character replacement

```
class Solution {
    public int characterReplacement(String s, int k) {
        int[] arr=new int[26];  // frequency map for all
alphabets

        int l=0,r=0,maxfreq=0,maxwindow=0;

        while(r<s.length())
        {
            arr[s.charAt(r)-'A']++;

            maxfreq=Math.max(maxfreq,arr[s.charAt(r)-'A']); //
maximum frequency

            int win=r-l+1;  // window size

            if(win-maxfreq > k)
            {
                arr[s.charAt(l)-'A']--;
                l++;    // shrink window
            }

            win=r-l+1;
            maxwindow=Math.max(maxwindow,win);
            r++;    // expand window
        }
        return maxwindow;
    }
}
```

- Permutation in string

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

        int [] freq1=new int[26];
        int [] freq2=new int[26];

        for(int i=0;i<s1.length();i++)
            freq1[s1.charAt(i)-'a']++;

        int l=0,r=0;

        while(r<s2.length())
        {
            freq2[s2.charAt(r)-'a']++;

            if(r-l+1 > s1.length())
            {
                freq2[s2.charAt(l)-'a']--;  // shrinkinhg window
size
                l++;
            }

            if(Arrays.equals(freq1,freq2))  // Checking if
frequency matches
                return true;

            r++;
        }
        return false;
    }
}
```

- Minimum size subarray sum

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {

        int l=0,r=0,s=0,min=Integer.MAX_VALUE;

        while(r<nums.length)
        {
            s+=nums[r];

            while(s >= target)
            {
                min=Math.min(min,r-l+1);
                s-=nums[l];
                l++;
            }
            r++;
        }

        return (min==Integer.MAX_VALUE) ? 0 : min;
    }
}
```

- Find K closest elements

```
class Solution {
    public List<Integer> findClosestElements(int[] arr, int k,
int x) {
        int l = 0, r = arr.length - k;  // binary search range
for window start

        while (l < r) {
            int mid = (l + r) / 2;
            // move towards side with smaller distance (prefer
left if equal)
            if (Math.abs(arr[mid] - x) <= Math.abs(arr[mid + k] -
x))
                r = mid;
            else
                l = mid + 1;
        }

        // collect k elements starting from final left index
        List<Integer> list = new ArrayList<>();
        for (int i = l; i < l + k; i++)
            list.add(arr[i]);

        return list;
    }
}
```

Or

```
class Solution {
    public List<Integer> findClosestElements(int[] arr, int k,
int x) {
        int l = 0, r = arr.length - 1;

        while (r-l >= k) {
            if (Math.abs(arr[l] - x) <= Math.abs(arr[r] - x))
```

```java
                r--;
            else
                l++;
        }

        List<Integer> list = new ArrayList<>();
        for (int i = l; i <= r; i++)
            list.add(arr[i]);

        return list;
    }
}
```

- Minimum Window Substring

```java
class Solution {
    public String minWindow(String s, String t) {

        if(s.length()<t.length())
            return "";

        // store frequency of each char in t
        HashMap<Character,Integer> map=new HashMap<>();
        for(int i=0;i<t.length();i++)
        {
            char ch=t.charAt(i);
            map.put(ch,map.getOrDefault(ch,0)+1);
        }
        int
unique=map.size(),start=0,end=0,min=Integer.MAX_VALUE,sidx=-1;

        while(end<s.length())
        {

            // Expansion phase - include current char and update
map
            char ch=s.charAt(end);
            if(map.containsKey(ch))
            {
                map.put(ch,map.get(ch)-1);
                if(map.get(ch)==0)  // one char requirement met
                    unique--;
            }

            // when all required chars are included
            while(unique==0)
            {
                int len=end-start+1;
                if(len<min) // update smallest window
                {
                    min=len;
                    sidx=start;
                }

                // shrink window from left
                ch=s.charAt(start);
                if(map.containsKey(ch))
                {
                    map.put(ch,map.get(ch)+1);
                    if(map.get(ch)>0)   // one char missing again
```

```
                        unique++;
                    }
                    start++;
                }

                end++;  // expand window

            }

            // return smallest substring if found
            return (sidx == -1) ? "" : s.substring(sidx, sidx + min);
        }
}
```

- Sliding window maximum

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        int[] res = new int[n - k + 1];
        Deque<Integer> dq = new ArrayDeque<>(); // stores indices

        for (int i = 0; i < n; i++) {
            // remove out-of-window indices
            if (!dq.isEmpty() && dq.peekFirst() <= i - k)
                dq.pollFirst();

            // remove smaller elements (they can’t be max)
            while (!dq.isEmpty() && nums[dq.peekLast()] <
nums[i])
                dq.pollLast();

            dq.offerLast(i); // add current index

            // record max once window is formed
            if (i >= k - 1)
                res[i - k + 1] = nums[dq.peekFirst()];
        }

        return res;
    }
}
```