

## VALID PALINDROME II Idea:

Two pointers from both ends.

On first mismatch, try skipping left or right once and check palindrome.

## Pattern:

Two pointers, greedy check on mismatch.

## Traps:

Reversing string repeatedly, extra string creation, forgetting only one deletion allowed.

```
class Solution {
    public boolean validPalindrome(String s) {
        int left = 0;
        int right = s.length() - 1;
        boolean ispal = false;
        StringBuilder sb = new StringBuilder(s);
        if (sb.reverse().toString().equals(s))
            return true;
        while (left < right) {
            if (s.charAt(left) == s.charAt(right)) {
                left++;
                right--;
            } else {
                StringBuilder sb1 = new StringBuilder(s);
                sb1.deleteCharAt(left);
                StringBuilder sb2 = new StringBuilder(s);
                sb2.deleteCharAt(right);
                if (isPal(sb1.toString()) || isPal(sb2.toString()))
                    return true;
                else {
                    return false;
                }
            }
        }
        return true;
    }

    public boolean isPal(String s) {
        StringBuilder st = new StringBuilder(s);
        if (st.reverse().toString().equals(s))
            return true;
        else
            return false;
    }
}
```

## MERGE STRINGS ALTERNATIVELY (1768)

```
class Solution {
    public String mergeAlternately(String word1, String word2) {
        StringBuilder sb = new StringBuilder();
        int turn = 0;
        while(turn < Math.min(word1.length(), word2.length()))
        {
            sb.append(word1.charAt(turn));
            sb.append(word2.charAt(turn));
            turn++;
        }
        if(word1.length() > word2.length())
        {
            String sub = word1.substring(turn);
            sb.append(sub);
        }
        else
        {
            String sub = word2.substring(turn);
            sb.append(sub);
        }
        return sb.toString();
    }
}
```

### Idea:

Use one index and append characters alternately until one word ends.  
After that, append the remaining part of the longer word.

---

### Pattern:

Two pointers (same index for both strings).  
Simple string building with leftover handling.

---

### Traps:

Using -1 in substring start index.  
Appending remaining from the wrong string.  
Overcomplicating with deletions or extra StringBuilders.

## MERGE SORTED ARRAY (88)

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int last = m + n - 1;
        while (m > 0 && n > 0)
        {
            if(nums1[m-1] >= nums2[n-1])
            {
                nums1[last] = nums1[m-1];
                m--;
            }
            else if(nums2[n-1] >= nums1[m-1])
            {
                nums1[last] = nums2[n-1];
                n--;
            }
            last--;
        }
        while(n > 0)
        {
            nums1[last] = nums2[n-1];
            n--;
            last--;
        }
    }
}
```

### Idea:

Start filling from the back.  
Compare last valid elements of both arrays and place the larger one at the end of `nums1`.  
Move pointers backward.

---

### Pattern:

Two pointers from end.  
Greedy comparison.  
In-place merge.

---

### Traps:

Not handling equality case.  
Wrong loop condition.  
Forgetting to copy remaining elements of `nums2`.

### REMOVE DUPLICATES IN PLACE EDITION

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int num = nums[0];
        int i = 0;
        int count = 1;
        int j = 1;
        while(i < nums.length)
        {
            if(num != nums[i])
            {
                num = nums[i];
                count++;
            }
            i++;
        }
        int unq = nums[0];
        int back = 1;
        int frnt = 1;
        while(frnt < nums.length)
        {
            if(unq != nums[frnt])
            {
                nums[back] = nums[frnt];
                unq = nums[frnt];
                back++;
            }
            frnt++;
        }
        return count;
    }
}
```

## Idea:

First pass counts how many unique elements are present.

Second pass shifts unique elements to the front using two pointers (**back** for placement, **frnt** for scanning).

---

## Pattern:

Two pointers on sorted array.

One pointer scans, one pointer builds the unique portion in-place.

---

## Traps:

Forgetting array is sorted.

Writing at wrong index (**frnt** instead of **back**).

Not resetting tracking variable (**unq**) before second pass.

## TWO SUM - INPUT ARRAY IS SORTED (167)

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int low = 0;
        int high = numbers.length-1;
        while(low<high)
        {
            if(numbers[low]+numbers[high]==target)
            {
                return new int[]{low+1, high+1};
            }
            else if(numbers[low]+numbers[high] > target)
            {
                high--;
            }
            else
            {
                low++;
            }
        }
    }
}
```

```

        return new int[]{low+1, high+1};
    }
}

```

## ROTATE ARRAY (189)

### TWO WAYS

ONE WAY -: TAKE EXTRA MEMORY, FROM THE ORIGINAL ARRAY, ADD IN THE NEW MEMORY'S  $(i+k \% n)$   $i+k$  because simple logic basically we have to shift the element  $k$  times so it goes forward  $k$  times,  $\%n$  for circular implementation so if we reach last element, it comes to the front.

ANOTHER OPTIMAL WAY -: Reverse the given array, and then we have two parts of reverse which we can reverse using a custom function (reverse)

```

public class Solution {
    public void rotate(int[] nums, int k) {
        int n = nums.length;
        k = k % n;
        reverse(nums, 0, n - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, n - 1);
    }

    private void reverse(int[] nums, int l, int r) {
        while (l < r) {
            int temp = nums[l];
            nums[l] = nums[r];
            nums[r] = temp;
            l++;
            r--;
        }
    }
}

```

## 3SUM

IDEA -: SIMILAR IDEA TO 2 POINTER USING SORTED ARRAY

[1,2,3,4,5,6,7,8,9]

In this first apply loop for all the elements, till arr.length and inside them for j, k perform two sum so  $i + j + k == \text{sum}$  and no matchings must be there meaning check for  $i \neq i-1$  and  $j \neq j-1$  and  $k \neq k-1$

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> outerlist = new ArrayList<>();
        for(int i=0; i<nums.length; i++)
        {
            if(i > 0 && nums[i]==nums[i-1]) continue;
            int j = i+1;
            int k = nums.length-1;
            while(j < k)
            {
                int sum = nums[i] + nums[j] + nums[k];
                if(sum > 0) k--;
                else if(sum < 0) j++;
                else if(sum==0)
                {
                    List<Integer> li = new ArrayList<>();
                    li.add(nums[i]);
                    li.add(nums[j]);
                    li.add(nums[k]);
                    outerlist.add(li);
                    j++;
                    k--;
                    while(j < k && nums[j]==nums[j-1]) j++;
                    while(j < k && nums[k]==nums[k+1]) k--;
                }
            }
        }
        return outerlist;
    }
}
```

## CONTAINER WITH MOST WATER

IDEA - EASY ONE, SIMPLY CALCULATE HEIGHT (ALREADY THE ARRAY), WIDTH IS HIGH - LOW, PUT THE MAX ONE IN AREA, AND GO FOR LOW++ IF HEIGHT OF LOW IS LESS THAN HEIGHT OF HIGH ( WE WANT TO GET THE MAX AREA SO SIMPLY SEARCHING) AND ELSE HIGH --;

```
class Solution {
    public int maxArea(int[] height) {
        int low = 0;
```

```

int high = height.length - 1;
int ans = 0;
while(low <= high)
{
    int width = high - low;
    int heig = Math.min(height[low], height[high]);
    int area = width * heig;
    ans = Math.max(ans, area);
    if(height[low] < height[high]) low++;
    else high--;
}
return ans;
}
}

```

## BOATS TO SAVE PEOPLE (881)

**IDEA -:** sort the array, apply similar to twosum logic. If sum <= limit, then up the count, and search for other people to send in boat by doing low++, high --. Mark them as visited, so that later we want to find out the individual heavy ones who couldn't be sent as pairs. If sum > limit then high --; later on in the same loop find the individual heavy ones.

```

class Solution {
    public int numRescueBoats(int[] people, int limit) {
        Arrays.sort(people);
        int count = 0;
        int low = 0;
        int high = people.length - 1;
        int[] done = new int[people.length];
        while(low < high)
        {
            int sum = people[low] + people[high];
            if(sum <= limit)
            {
                count++;
                people[low] = -1;
                people[high] = -1;
                low++;
                high--;
            }
            if(sum > limit) high--;
        }
        for(int i=0; i < people.length; i++)
        {

```

```
        if(people[i] != -1) count++;  
    }  
    return count;  
}  
}
```