2023000218

# ICP NOTES

LEETCODE PROBLEMS

Rajarshi Ghosh
9-25-2025

# Counting Bits / 338

```
for i in range(1, n+1):
    if offset * 2 == i:
        offset = i    # Update offset to new power of 2
    dp[i] = 1 + dp[i - offset]  # Add 1 to the bit count of i - offset

return dp
```

**Why It Works**

- Every number can be expressed as:

**i = offset + x**

- Where offset is the **largest power of 2 ≤ i**, and x < offset.
- Since offset has only one 1-bit, **dp[i] = 1 + dp[i - offset].**
- Dry Run for `n = 5` :

| i | offset | dp[i] | Explanation |
|---|---|---|---|
| 1 | 1 | 1 | 1 + dp[0] = 1 |
| 2 | 2 | 1 | offset updates to 2, dp[2] = 1 + dp[0] |
| 3 | 2 | 2 | dp[3] = 1 + dp[1] = 2 |
| 4 | 4 | 1 | offset updates to 4 |
| 5 | 4 | 2 | dp[5] = 1 + dp[1] = 2 |

**Output: [0, 1, 1, 2, 1, 2]**

**⏱ Time and Space Complexity**

- **Time:** O(n)
- **Space:** O(n)

# Missing Number / 268

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        n = len(nums)
        expected_sum = n * (n + 1) // 2
        actual_sum = sum(nums)
        return expected_sum - actual_sum
```

**Why It Works**

- The numbers are from 0 to n, but one number is missing.
- The **sum of first n natural numbers** (0 to n) can be computed with the formula:

**Expected_Sum = [n * (n + 1)] / 2**

- actual_sum is just the sum of the numbers present in the array.
- The **difference between the expected sum and actual sum** gives the missing number:

**missing=expected_sum−actual_sum**

**Dry Run (Example)**

Input: nums = [3, 0, 1]

| Step | n | expected_sum | actual_sum | missing (return value) | Explanation |
|---|---|---|---|---|---|
| 1 | 3 | 3∗4//2=63 * 4 // 2 = 63∗4//2=6 | 3+0+1=43+0+1 = 43+0+1=4 | 6−4=26 - 4 = 26−4=2 | 2 is the missing number |

Output: 2

**Time and Space Complexity**

- **Time Complexity:**
    - Calculating expected_sum → O(1)
    - Calculating actual_sum (using sum()) → O(n)

    **Total:** O(n)
- **Space Complexity:**
    - Only a few variables are used → O(1) (constant space)

# Number of 1 Bits / 191

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        res = 0
        while n:
            # res += n % 2
            n &= n - 1
            # n = n >> 1
            res += 1
        return res
```

**Why It Works**

- **Goal:** Count how many 1 bits are present in the binary representation of n.
- The trick: n & (n-1) removes the **rightmost set bit** from n.
    - Example: n = 10110 → n-1 = 10101 → n & (n-1) = 10100 (rightmost 1 removed)
- Each loop iteration removes one 1 bit and increments res.
- Loop continues until n becomes 0, meaning all 1 bits have been counted.

This approach is more efficient than checking every bit individually because it only loops once per set bit.

**Dry Run (Example)**

Input: n = 11 (binary = 1011)

| Iteration | n (binary) | n-1 (binary) | n & (n-1) | res | Explanation |
|---|---|---|---|---|---|
| Start | 1011 | 1010 | 1010 | 1 | Rightmost 1 removed |
| 2 | 1010 | 1001 | 1000 | 2 | Rightmost 1 removed |
| 3 | 1000 | 0111 | 0000 | 3 | Rightmost 1 removed, n = 0 → stop |

Output: 3 (because 11 has three 1's in its binary form)

**Time and Space Complexity**

- **Time Complexity:**
    - Runs once for every set bit in n → O(k), where k = number of 1-bits.
    - Worst case: O(log n) (when all bits are 1).
- **Space Complexity:** Only uses constant variables → O(1)

# Reverse Bits / 190

```
class Solution:
    def reverseBits(self, n: int) -> int:
        res = 0
        for i in range(32):
            bit = (n >> i) & 1
            res = res | (bit << (31 - i))
        return res
```

**Why It Works**

- **Goal:** Reverse the bits of a 32-bit unsigned integer.
- **Step-by-step logic:**
    1. (n >> i) & 1 extracts the i-th bit from the right.
    2. (bit << (31 - i)) places that bit into its **reversed position** (mirrored around center).
    3. res | ... sets that bit in res without affecting previously set bits.
- After looping through all 32 bits, res contains the bit-reversed number.

## Dry Run (Example)

Input: `n = 13` (`binary = 00000000000000000000000000001101`)

| i | Extracted Bit (n >> i) & 1 | Shift (bit << (31-i)) | res (after setting) |
|---|---|---|---|
| 0 | 1 | Bit goes to position 31 | 10000000000000000000000000000000 |
| 1 | 0 | Nothing added | 10000000000000000000000000000000 |
| 2 | 1 | Bit goes to position 29 | 10100000000000000000000000000000 |
| 3 | 1 | Bit goes to position 28 | 10110000000000000000000000000000 |
| ... | Remaining bits are 0 | No changes | Final Result |

Output: res = 2952790016 (binary = 10110000000000000000000000000000)

**Time and Space Complexity**

- **Time Complexity:**
    - Loop runs exactly 32 times → O(32) → O(1) (constant time)
- **Space Complexity:**
    - Only uses a few integer variables → O(1) (constant space)

# Middle of the Linked List / 876

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        return slow  # slow now points to the middle node
```

**Why It Works**

- Uses **two pointers**:
  - slow moves one step at a time.
  - fast moves two steps at a time.
- When fast reaches the end of the list:
  - slow will be at the middle.
- This works because slow progresses at **half the speed** of fast, so when fast has traveled the full length, slow has traveled half.
-

## Dry Run (Example)

Input: `head = [1, 2, 3, 4, 5]`

| Step | slow (value) | fast (value) | Explanation |
|---|---|---|---|
| Start | 1 | 1 | Both at head |
| 1st Iteration | 2 | 3 | slow moves 1 step, fast moves 2 steps |
| 2nd Iteration | 3 | 5 | slow moves 1 step, fast moves 2 steps |
| 3rd Iteration | - | - | fast.next is None → stop |

Output: `slow` points to `3` (the middle node).

**Time and Space Complexity**

- **Time Complexity:**
    - Both pointers traverse the list at most once → O(n)
- **Space Complexity:**
    - Only uses two pointers → O(1)

# Linked List Cycle / 141

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow is fast:  # Cycle detected
                return True
        return False  # No cycle
```

**Why It Works**

- Uses **Floyd's Cycle Detection Algorithm** (Tortoise and Hare method).
- slow moves one step, fast moves two steps.
- **Two possible outcomes:**
    1. If there is **no cycle**, fast will reach None (end of list) → return False.
    2. If there **is a cycle**, eventually fast will "lap" slow, and they will meet → return True.

This works because in a circular path, a faster pointer will always catch up to a slower pointer.

Dry Run (Example)

Input: head = [3, 2, 0, -4] with cycle connecting -4 → 2

| Step | slow (value) | fast (value) | Explanation |
| --- | --- | --- | --- |

| | | | |
|---|---|---|---|
| Start | 3 | 3 | Both at head |
| 1st Iteration | 2 | 0 | slow moves 1 step, fast moves 2 steps |
| 2nd Iteration | 0 | 2 | slow moves 1 step, fast moves 2 steps |
| 3rd Iteration | -4 | -4 | slow moves 1 step, fast moves 2 steps → **they meet → cycle found** |

Output: True

**Time and Space Complexity**

- **Time Complexity:**
  - Worst case: O(n) (both pointers traverse nodes until they meet or fast reaches the end)
- **Space Complexity:**
  - No extra data structures used → O(1)

# Linked List Cycle II / 142

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head:
            return None
        slow, fast = head, head
        # Phase 1: Detect cycle (Floyd's Tortoise & Hare)
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                break
        else:
            # No cycle detected (while loop exited normally)
```

```python
        return None

    # Phase 2: Find the cycle start
    fast = head
    while fast != slow:
        fast = fast.next
        slow = slow.next
    return fast
```

**Why It Works**

This uses **Floyd's Cycle Detection Algorithm** with an additional step to find the **start node of the cycle**.

1. **Phase 1 – Detect the Cycle:**
   - slow moves 1 step at a time, fast moves 2 steps.
   - If slow and fast meet, a cycle exists.
   - If fast reaches None, there is no cycle → return None.
2. **Phase 2 – Locate the Start of Cycle:**
   - Reset fast to head.
   - Move both slow and fast one step at a time.
   - The node where they meet is the **entry point of the cycle**.

**Why Phase 2 Works:**

- When slow and fast first meet, slow has traveled k steps into the cycle.
- Distance from head to cycle start = a, distance from cycle start to meeting point = k.
- Resetting one pointer to head ensures they both meet at cycle_start after a steps.

## Dry Run (Example)

Input: `head = [3, 2, 0, -4]` with cycle connecting `-4 → 2`

| Step | slow (value) | fast (value) | Explanation |
|---|---|---|---|
| Start | 3 | 3 | Both at head |
| Iter 1 | 2 | 0 | slow +1, fast +2 |
| Iter 2 | 0 | 2 | slow +1, fast +2 |
| Iter 3 | -4 | -4 | slow +1, fast +2 → **meet → cycle detected** |
| Phase 2 Start | slow = -4, fast reset = 3 | - | Reset fast to head |
| Phase 2 Iter 1 | slow = 2, fast = 2 | - | They meet at `2`, cycle start found |

Output: Node with value `2` (start of cycle)

**Time and Space Complexity**

- **Time Complexity:**
    - Phase 1 takes O(n) in worst case (detecting cycle).
    - Phase 2 takes at most O(n) more steps to find start.
      **Total:** O(n)
- **Space Complexity:**
    - Only uses two pointers → O(1)

# Reverse Linked List / 206

```
class Solution:
  def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
    # Recursive approach: Time = O(n), Space = O(n) (due to recursion stack)
    if not head:
      return None  # Base case: empty list
    newHead = head
    if head.next:
      # Recursively reverse the rest of the list
      newHead = self.reverseList(head.next)
      # Reverse the current link
      head.next.next = head
      head.next = None
    return newHead
```

**Why It Works**

- **Recursion unwinds from the last node back to the first node.**
- Each call reverses **one link**:
    1. newHead = self.reverseList(head.next) reverses the sublist after head.
    2. head.next.next = head points the next node back to head.
    3. head.next = None breaks the old forward link.
- When recursion fully unwinds, newHead points to the new head of the reversed list.

**Dry Run (Example)**

Input: head = [1, 2, 3]

| Recursion Level | head | newHead after recursion | Operation Performed |
|---|---|---|---|
| Call 1 | 1 | returned from reversing [2,3] | 2.next → 1, 1.next = None |
| Call 2 | 2 | returned from reversing [3] | 3.next → 2, 2.next = None |
| Call 3 | 3 | head.next is None → return 3 | Base case reached |

Output: newHead = 3 → 2 → 1 → None

**Time and Space Complexity**

- **Time Complexity:**
  - Each node is visited once → O(n)
- **Space Complexity:**
  - Recursion stack holds up to n frames in worst case → O(n)

# Remove Nth Node from End of List / 19

```
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        dummy = ListNode(0)      # Dummy node simplifies edge cases
        dummy.next = head
        ahead = behind = dummy
        # Move ahead pointer n+1 steps ahead
        for _ in range(n + 1):
            ahead = ahead.next
        # Move both pointers until ahead reaches the end
        while ahead:
            ahead = ahead.next
            behind = behind.next
        # behind is just before the node we want to remove
        behind.next = behind.next.next
        return dummy.next  # Return new head
```

**Why It Works**

- **Dummy Node:** Handles edge cases cleanly (e.g., removing the head itself).
- **Two-pointer approach:**
    1. Move ahead pointer n+1 steps forward.
    2. Move ahead and behind together until ahead reaches the end.
    3. At this point, behind points to the node **just before** the node to remove.
    4. Skip the target node by behind.next = behind.next.next.

This guarantees exactly one pass through the list, maintaining O(n) time complexity.

**Dry Run (Example)**

Input: head = [1, 2, 3, 4, 5], n = 2

| Step | ahead (val) | behind (val) | Action |
|---|---|---|---|
| Init | dummy | dummy | Setup |
| Move ahead 3 steps (n+1) | ahead points to node 3 | behind still dummy | Create gap of n between pointers |
| Iter 1 | ahead → 4 | behind → 1 | Both move one step |
| Iter 2 | ahead → 5 | behind → 2 | Both move one step |
| Iter 3 | ahead → None | behind → 3 | Stop loop |
| Remove | behind.next = behind.next.next → 4 skips 4 | New list: 1 → 2 → 3 → 5 | |

Output: head = [1, 2, 3, 5]

**Time and Space Complexity**

- **Time Complexity:**
    - Single traversal of the list → O(n)
- **Space Complexity:**
    - Constant extra space (dummy + pointers) → O(1)

# Valid Parentheses / 20

```
class Solution:
    def isValid(self, s: str) -> bool:
        hashmap = {')': '(', '}': '{', ']': '['}
        stk = []
        for c in s:
            if c not in hashmap:
                stk.append(c)  # Opening bracket
            else:
                if not stk or stk.pop() != hashmap[c]:
                    return False
        return not stk  # True if stack is empty (all matched)
```

**Why It Works**

- **Stack-based approach:**
    1. Push every opening bracket ((, {, [) onto the stack.
    2. When encountering a closing bracket, check:
        - If stack is empty → invalid (no opening match).
        - If top of stack is not the matching opening bracket → invalid.
        - Otherwise pop the opening bracket from the stack (pair matched).
    3. At the end, if the stack is empty → all brackets matched → return True.
       Otherwise → there are unmatched opening brackets → return False.

## Dry Run (Example)

Input: `s = "({[]})"`

| Step | c | Stack Before | Action | Stack After |
|---|---|---|---|---|
| 1 | ( | [] | Push | [(] |
| 2 | { | [(] | Push | [(, {] |
| 3 | [ | [(, {] | Push | [(, {, [] |
| 4 | ] | [(, {, [] | Pop (match [ ) | [(, {] |
| 5 | } | [(, {] | Pop (match { ) | [(] |
| 6 | ) | [(] | Pop (match ( ) | [] |

Output: `True` (all brackets matched, stack empty)

**Time and Space Complexity**

- **Time Complexity:**
  - Each character is processed once → O(n)
- **Space Complexity:**
  - Stack holds at most n characters in worst case → O(n)

# Find Median from Data Stream / 295

```
import heapq

class MedianFinder:

    def __init__(self):

        # small = max heap (store as negative numbers)

        # large = min heap

        self.small, self.large = [], []

    def addNum(self, num: int) -> None:

        # Step 1: Push into max-heap (small)

        heapq.heappush(self.small, -num)

        # Step 2: Ensure order property: max(small) <= min(large)

        if self.small and self.large and (-self.small[0] > self.large[0]):

            val = -heapq.heappop(self.small)

            heapq.heappush(self.large, val)

        # Step 3: Balance sizes (difference can't exceed 1)

        if len(self.small) > len(self.large) + 1:

            val = -heapq.heappop(self.small)

            heapq.heappush(self.large, val)

        elif len(self.large) > len(self.small) + 1:

            val = heapq.heappop(self.large)

            heapq.heappush(self.small, -val)

    def findMedian(self) -> float:

        if len(self.small) > len(self.large):

            return -self.small[0]

        elif len(self.large) > len(self.small):

            return self.large[0]
```

```
    else:
        return (-self.small[0] + self.large[0]) / 2
```

**Why It Works**

- **Maintain two heaps:**
    - small (max-heap) stores the smaller half of numbers.
    - large (min-heap) stores the larger half of numbers.
- **Insertion logic:**

1. Push number into small (max-heap using negative values).
2. Ensure **order property:** largest in small ≤ smallest in large.
3. Balance sizes so the difference in length ≤ 1.

- **Median calculation:**
    - If one heap is larger → median is the top of that heap.
    - If heaps are equal → median = average of tops.

This allows **O(log n)** insertion and **O(1)** median retrieval.

**Dry Run (Example)**

Input: [1, 2, 3]

| Step | small (max-heap) | large (min-heap) | Median |
|---|---|---|---|
| Add 1 | [ -1 ] | [] | 1 |
| Add 2 | [ -1 ] | [ 2 ] | (1+2)/2 = 1.5 |
| Add 3 | [ -2, -1 ] | [ 3 ] | 2 |

**Time and Space Complexity**

- **addNum():** O(log n) → heap insertion and balancing
- **findMedian():** O(1) → just access heap tops
- **Space Complexity:** O(n) → all numbers stored in heaps

# Spiral Matrix / 54

```
from typing import List

class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        res = []
        if not matrix:
            return res
        left, right = 0, len(matrix[0])
        top, bottom = 0, len(matrix)
        while left < right and top < bottom:
            # Traverse top row
            for i in range(left, right):
                res.append(matrix[top][i])
            top += 1
            # Traverse right column
            for i in range(top, bottom):
                res.append(matrix[i][right - 1])
            right -= 1
            if not (left < right and top < bottom):
                break
            # Traverse bottom row (right to left)
            for i in range(right - 1, left - 1, -1):
                res.append(matrix[bottom - 1][i])
            bottom -= 1
            # Traverse left column (bottom to top)
            for i in range(bottom - 1, top - 1, -1):
                res.append(matrix[i][left])
            left += 1
        return res
```

**Why It Works**

- **Maintain boundaries:** top, bottom, left, right define the current rectangle to traverse.
- **Traversal order (spiral):**
    1. Top row → left to right
    2. Right column → top to bottom
    3. Bottom row → right to left
    4. Left column → bottom to top
- After each traversal, **shrink boundaries** to move inward.
- Repeat until left >= right or top >= bottom, meaning the entire matrix is traversed.

## Dry Run (Example)

Input:

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

| Step | Operation | Added to res | Boundaries (top,bottom,left,right) |
|---|---|---|---|
| 1 | Top row | 1,2,3 | top=1, bottom=3, left=0, right=3 |
| 2 | Right column | 6,9 | top=1, bottom=3, left=0, right=2 |
| 3 | Bottom row | 8,7 | top=1, bottom=2, left=0, right=2 |
| 4 | Left column | 4 | top=2, bottom=2, left=1, right=2 |
| 5 | Top row | 5 | top=3, bottom=2, left=1, right=2 → exit loop |

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

**Time and Space Complexity**

- **Time Complexity:**
    - Each element visited once → O(m × n) for an m×n matrix
- **Space Complexity:**
    - Result array stores all elements → O(m × n)

# Set Matrix Zeroes / 73

```
from typing import List

class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        ROWS, COLS = len(matrix), len(matrix[0])
        rowZero = False  # Whether the first row needs to be zeroed
        # Step 1: Use first row/col as markers
        for r in range(ROWS):
            for c in range(COLS):
                if matrix[r][c] == 0:
                    matrix[0][c] = 0
                    if r > 0:
                        matrix[r][0] = 0
                    else:
                        rowZero = True
        # Step 2: Set zeroes based on markers (skip first row/col)
        for r in range(1, ROWS):
            for c in range(1, COLS):
                if matrix[0][c] == 0 or matrix[r][0] == 0:
                    matrix[r][c] = 0
        # Step 3: Zero first column if needed
        if matrix[0][0] == 0:
            for r in range(ROWS):
                matrix[r][0] = 0
        # Step 4: Zero first row if needed
        if rowZero:
            for c in range(COLS):
                matrix[0][c] = 0
```

**Why It Works**

- **Goal:** Set entire row and column to 0 if a cell is 0, **in-place**.
- **Strategy:** Use **first row and first column as markers** to avoid extra space.
    1. Loop through matrix; if matrix[r][c] == 0, mark matrix[0][c] = 0 and matrix[r][0] = 0.
        - Special handling for first row using rowZero boolean.
    2. Loop through matrix (excluding first row/col); if row or column marker is 0, set the cell to 0.
    3. Zero first column if matrix[0][0] == 0.
    4. Zero first row if rowZero is True.

This avoids using extra O(m+n) space for marker arrays.

Dry Run (Example)

matrix = [

[1,1,1],

[1,0,1],

[1,1,1]

]

| **Step** | **Operation** | **Matrix State** |
|---|---|---|
| Step 1 | Mark zeros | [1,0,1],[0,0,1],[1,1,1] → first row/col markers set |
| Step 2 | Set zeros based on markers | [1,0,1],[0,0,0],[1,0,1] |
| Step 3 | First column check | matrix[0][0] != 0 → no change |
| Step 4 | First row check | rowZero=False → no change |

**Time and Space Complexity**

- **Time Complexity:**
    - Two nested loops over matrix → O(m × n)
- **Space Complexity:**
    - Only uses variables for markers → O(1) (in-place)

# Two Sum

```python
from typing import List

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        prevMap = {}  # value -> index
        for i, n in enumerate(nums):
            diff = target - n
            if diff in prevMap:
                return [prevMap[diff], i]
            prevMap[n] = i
```

**Why It Works**

- **Goal:** Find indices of two numbers whose sum equals target.
- **Hash map approach:**
    1. Iterate through array, for each number n:
        - Compute diff = target - n.
        - Check if diff exists in prevMap.
            - If yes → return [prevMap[diff], i].
        - Otherwise, store n in prevMap with its index.
- Ensures **one-pass solution** with O(1) lookup for complements.

## Dry Run (Example)

Input: `nums = [2, 7, 11, 15], target = 9`

| i | n | diff = target - n | prevMap | Action |
|---|---|---|---|---|
| 0 | 2 | 7 | {} | Store 2 → prevMap = {2:0} |
| 1 | 7 | 2 | {2:0} | 2 found → return [0,1] |

Output: `[0,1]`

**Time and Space Complexity**

- **Time Complexity:**
    - Single pass through array → O(n)
- **Space Complexity:**
    - Hash map stores up to n elements → O(n)

# Contains Duplicate

```
from typing import List

class Solution:

    def containsDuplicate(self, nums: List[int]) -> bool:

        seen = set()

        for n in nums:

            if n in seen:

                return True

            seen.add(n)

        return False
```

**Why It Works**

- **Goal:** Check if the array contains any duplicates.
- **Approach:**
    1. Initialize an empty set seen.
    2. Iterate through each number n:
        - If n is already in seen, a duplicate exists → return True.
        - Otherwise, add n to seen.
    3. If the loop completes without returning, no duplicates exist → return False.
- **Rationale:** Sets allow O(1) average lookup and insertion, making detection efficient.

## Dry Run (Example)

Input: `nums = [1, 2, 3, 1]`

| Step | n | seen before | Action | Duplicate Found? |
|---|---|---|---|---|
| 1 | 1 | {} | Add 1 → seen = {1} | No |
| 2 | 2 | {1} | Add 2 → seen = {1,2} | No |
| 3 | 3 | {1,2} | Add 3 → seen = {1,2,3} | No |
| 4 | 1 | {1,2,3} | 1 in seen → return True | Yes |

Output: `True`

**Time and Space Complexity**

- **Time Complexity:** O(n) → each number checked once
- **Space Complexity:** O(n) → set stores up to n elements

# Valid Anagram

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        return sorted(s) == sorted(t)
```

**Why It Works**

- **Goal:** Determine if t is an anagram of s (same letters, same frequency).
- **Approach:**
  1. Sorting both strings puts the same characters in the same order.
  2. If sorted versions are equal → all letters match in frequency → anagram.
  3. Otherwise → not an anagram.

## Dry Run (Example)

Input: `s = "listen", t = "silent"`

| Step | Operation | Result |
|---|---|---|
| 1 | sorted(s) | ['e','i','l','n','s','t'] |
| 2 | sorted(t) | ['e','i','l','n','s','t'] |
| 3 | Compare | Equal → return True |

Output: `True`

**Time and Space Complexity**

- **Time Complexity:** O(n log n) → sorting each string of length n
- **Space Complexity:** O(n) → storing sorted lists

# Group Anagrams / 49

```
from typing import List
from collections import defaultdict
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        res = defaultdict(list)  # mapping: char count -> list of anagrams
        for s in strs:
            count = [0] * 26  # 26 letters for 'a' to 'z'
            for c in s:
```

```
            count[ord(c) - ord("a")] += 1
        res[tuple(count)].append(s)
    return list(res.values())
```

**Why It Works**

- **Goal:** Group words that are anagrams together.
- **Approach:**
    1. Use a **character count array** of size 26 to represent each word.
        - count[i] = number of occurrences of letter chr(i + ord('a')).
    2. Convert the count array to a **tuple** (hashable) and use it as a key in a dictionary.
    3. Words with the same character counts are grouped together in the same list.
- Efficient because it avoids sorting each string, which would take O(k log k) per word (k = length of word).

## Dry Run (Example)

Input: `strs = ["eat","tea","tan","ate","nat","bat"]`

| Word | Character Count Key | Added to Dictionary |
|---|---|---|
| "eat" | (1,0,0,...,1,1,0...) | {"eat"} |
| "tea" | (1,0,0,...,1,1,0...) | {"eat","tea"} |
| "tan" | (1,0,0,...,1,0,1...) | {"tan"} |
| "ate" | (1,0,0,...,1,1,0...) | {"eat","tea","ate"} |
| "nat" | (1,0,0,...,1,0,1...) | {"tan","nat"} |
| "bat" | (1,1,0,...,1,0,0...) | {"bat"} |

Output: `[["eat","tea","ate"], ["tan","nat"], ["bat"]]`

**Time and Space Complexity**

- **Time Complexity:** $O(n \times k)$
    - n = number of words, k = max length of a word
    - Counting characters is O(k) per word, appending to dictionary is O(1)
- **Space Complexity:** $O(n \times k)$
    - Storage for dictionary and result lists

# Top K Frequent Elements / 347

```
from typing import List

class Solution:

    def topKFrequent(self, nums: List[int], k: int) -> List[int]:

        count = {}

        freq = [[] for _ in range(len(nums) + 1)]

        # Count frequency of each number

        for n in nums:

            count[n] = 1 + count.get(n, 0)

        # Bucket sort: place numbers into their frequency index

        for n, c in count.items():

            freq[c].append(n)

        res = []

        # Traverse buckets from highest freq to lowest

        for i in range(len(freq) - 1, 0, -1):

            for n in freq[i]:

                res.append(n)

                if len(res) == k:

                    return res
```

**Why It Works**

- **Goal:** Find k elements that appear most frequently in the array.
- **Approach:**
    1. Count the frequency of each element using a dictionary (count).
    2. Use **bucket sort**:
        - Create a list of lists freq where freq[i] contains elements that appear exactly i times.
    3. Traverse freq from **highest frequency to lowest**:
        - Collect elements until we have k elements.
- Efficient because we avoid full sorting by frequency and exploit the limited range (max frequency ≤ n).

**Dry Run (Example)**

Input: nums = [1,1,1,2,2,3], k = 2

1. Count frequencies → count = {1:3, 2:2, 3:1}
2. Bucket sort → freq = [[], [3], [2], [1], [], [], []] (indices = frequencies)
3. Traverse freq from high →
   - i=3 → add 1 → res = [1]
   - i=2 → add 2 → res = [1,2] → k=2 reached → return [1,2]

Output: [1,2] (order can vary among same-frequency elements)

**Time and Space Complexity**

- **Time Complexity:** O(n)
  - Counting frequencies → O(n)
  - Bucket sort insertion → O(n)
  - Collect top k → O(n) in worst case
- **Space Complexity:** O(n)
  - Dictionary + bucket array + result list

# Is Subsequence / 392

```
class Solution:
  def isSubsequence(self, s: str, t: str) -> bool:
    i, j = 0, 0
    while i < len(s) and j < len(t):
      if s[i] == t[j]:
        i += 1
      j += 1  # always move j forward
    return i == len(s)
```

**Why It Works**

- **Goal:** Determine if s is a subsequence of t.
- **Approach:** Two-pointer technique:
  1. i tracks position in s, j tracks position in t.
  2. Iterate through t (j moves forward always):
     - If t[j] == s[i], move i forward (match found).
     - If not, just move j forward.

3. After traversal, if i == len(s), all characters of s were found in order → True. Otherwise → False.

- **Rationale:** Only order matters; characters in between are ignored.

## Dry Run (Example)

Input: `s = "abc", t = "ahbgdc"`

| i | j | s[i] | t[j] | Action |
|---|---|---|---|---|
| 0 | 0 | a | a | match → i=1, j=1 |
| 1 | 1 | b | h | no match → j=2 |
| 1 | 2 | b | b | match → i=2, j=3 |
| 2 | 3 | c | g | no match → j=4 |
| 2 | 4 | c | d | no match → j=5 |
| 2 | 5 | c | c | match → i=3, j=6 |

Output: `i = len(s) = 3` → `True`

**Time and Space Complexity**

- **Time Complexity:** O(len(t)) → single pass through t
- **Space Complexity:** O(1) → only pointers i and j used

# Product of Array Except Self / 238

```
from typing import List

class Solution:

    def productExceptSelf(self, nums: List[int]) -> List[int]:

        res = [1] * len(nums)

        prefix = 1

        for i in range(len(nums)):

            res[i] = prefix

            prefix *= nums[i]

        postfix = 1

        for i in range(len(nums) - 1, -1, -1):

            res[i] *= postfix

            postfix *= nums[i]

        return res
```

**Why It Works**

- **Goal:** Compute res[i] = product of all nums[j] except nums[i] **without division**.
- **Approach:** Use **prefix and postfix products**:
    1. **Prefix pass:** res[i] = product of all elements before i.
        - Keep a running prefix product.
    2. **Postfix pass:** Multiply res[i] by product of all elements after i.
        - Keep a running postfix product from the end.
- This ensures each res[i] = product of all elements except itself.

## Dry Run (Example)

Input: `nums = [1,2,3,4]`

| i | prefix | res[i] | prefix after |
|---|---|---|---|
| 0 | 1 | 1 | 1*1=1 |
| 1 | 1 | 1 | 1*2=2 |
| 2 | 2 | 2 | 2*3=6 |
| 3 | 6 | 6 | 6*4=24 |

res after prefix: [1,1,2,6]

Postfix pass:

| i | postfix | res[i] | postfix after |
|---|---|---|---|
| 3 | 1 | 6*1=6 | 1*4=4 |
| 2 | 4 | 2*4=8 | 4*3=12 |
| 1 | 12 | 1*12=12 | 12*2=24 |
| 0 | 24 | 1*24=24 | 24*1=24 |

Output: [24,12,8,6]

**Time and Space Complexity**

- **Time Complexity:** O(n) → two passes through the array
- **Space Complexity:** O(1) extra (res array does not count as extra)

# Sum of Two Integers / 371

```
class Solution:
    def getSum(self, a: int, b: int) -> int:
        while b != 0:
            c = a & b      # carry: bits where both a and b are 1
            a = a ^ b      # sum without carry (XOR)
            b = c << 1     # shift carry left to add in next iteration
        return a
```

**Why It Works**

- **Goal:** Compute a + b without using + or -.
- **Approach:** Simulate bitwise addition.
    1. a ^ b gives sum of bits without considering carry (like binary addition ignoring carry).
    2. a & b gives positions where carry will be generated (both bits are 1).
    3. Left shift the carry (c << 1) to add it in the next higher bit position.
    4. Repeat until carry (b) becomes zero → no more carry to add.
- This is exactly how hardware performs binary addition.

## Dry Run (Example)

Input: `a = 5 (0101), b = 3 (0011)`

| Step | a (bin) | b (bin) | c = a&b | a = a^b | b = c<<1 |
|---|---|---|---|---|---|
| Init | 0101 | 0011 | - | - | - |
| 1 | 0101 | 0011 | 0001 | 0110 | 0010 |
| 2 | 0110 | 0010 | 0010 | 0100 | 0100 |
| 3 | 0100 | 0100 | 0100 | 0000 | 1000 |
| 4 | 0000 | 1000 | 0000 | 1000 | 0000 |

Output: `a = 1000 (8)`

**Time and Space Complexity**

- **Time Complexity:** O(1) (bounded by number of bits → 32 iterations max for 32-bit integers)
- **Space Complexity:** O(1) (uses only variables a, b, c)