Leetcode

Hanhee Lee

June 9, 2025

Contents

1 General

1.1 Interviewer Considerations

Notes:

- How did the candidate **analyze** the problem?
- Did the candidate miss any special or **edge** cases?
- Did the candidate approach the problem **methodically** and logically?
- Does the candidate have a strong foundation in basic computer science **concepts**?
- Did the candidate produce **working code**? Did the candidate **test** the code?
- Is the candidate's code clean and easy to read and **maintain**?
- Can the candidate **explain** their ideas clearly?

1.2 Steps for Success During the Technical Interview

Summary:

1. **Clarify the question**
   (a) Understand what the question is asking and gather example inputs and outputs.
   (b) Clarify constraints such as:
      i. Can numbers be negative or repeated?
      ii. Are values sorted or do we need to sort them?
      iii. Can we assume input validity?
   (c) Asking clarifying questions shows communication skills and prevents missteps.
2. **Design a solution**
   (a) Avoid immediate coding; propose an initial approach and refine it.
   (b) Analyze the algorithm's time and space complexity.
   (c) Consider and address edge cases.
   (d) Think aloud to demonstrate logical reasoning and collaboration.
   (e) Discuss non-optimal ideas to show your thought process.
3. **Write your code**
   (a) Structure the solution using helper functions.
   (b) Confirm API details when uncertain.
   (c) Use your strongest programming language and full syntax.
   (d) Write complete, working code—not pseudocode.
4. **Test your code**
   (a) Validate your solution with 1–2 example test cases.
   (b) Walk through each line using inputs.
   (c) Do not assume correctness—prove it through testing.
   (d) Discuss any further optimizations and their trade-offs.

1.3 Common Mistakes to Avoid

Warning:

1. Starting to code without clarifying the problem.
2. Failing to write or discuss sample inputs and outputs.
3. Using pseudocode instead of fully functional code.
4. Misunderstanding the problem or optimizing prematurely.

1.4 Syntax

Summary:

1. `dict.items()`
   - Returns a view object that displays a list of a dictionary's key-value tuple pairs.
2. `sorted(iterable, key=..., reverse=...)`
   - `iterable`: The sequence or collection (e.g., list, dictionary view) to be sorted.
   - `key=...`: A function that extracts a comparison key from each element. Sorting is performed based on the result of this function.
     - `key=lambda x: x[0]`: Sort by the first element of each tuple.
     - `key=lambda x: x[1]`: Sort by the second element of each tuple.
   - `reverse=...`: A boolean value. If `True`, sorted in descending order; otherwise, sorted in ascending order (default is `False`).
3. `collections.Counter(iterable)`
   - Counts the frequency of each unique element in `iterable` and returns a dictionary-like object.
   - **Arguments:**
     - `iterable`: a sequence (e.g., list, string) or any iterable containing hashable elements.

1.5 Big-O Complexity Chart

Summary:

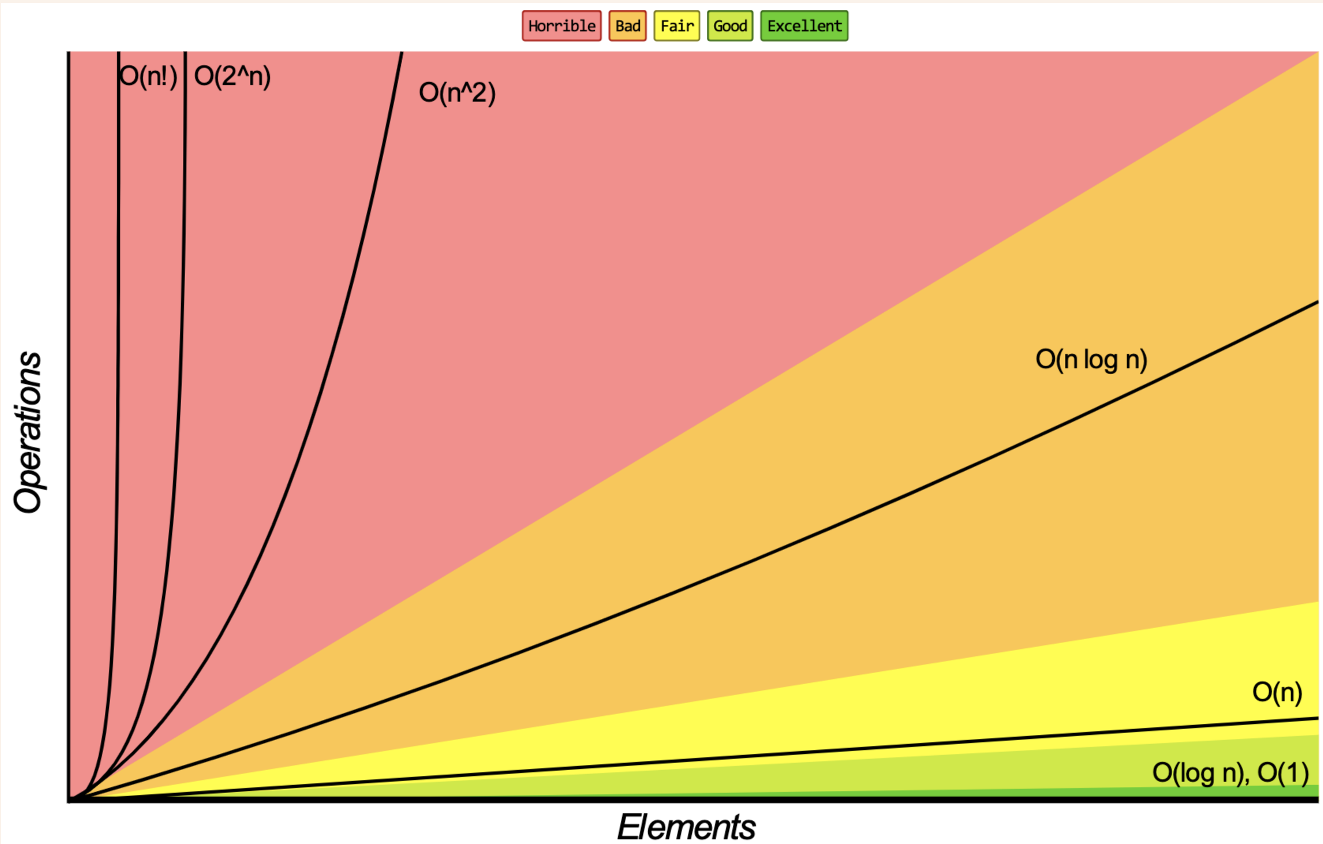

Figure 1

1.6 Common Data Structure Operations

Summary:

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figure 2

1.7 Array Sorting Algorithms

Summary:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 3

2 Arrays and Hashing

2.1 When to Use?

Summary:

- To count frequencies in $O(n)$ time.
- To check membership in constant time.
- To map keys to values (e.g., index, count, group).
- To group elements by shared features (e.g., anagrams).
- To detect duplicates efficiently.

2.2 Hashing

Algorithm:

```python
def solve_problem(nums):
    # Step 1: Initialize the hashmap (e.g., for frequency, index, or existence check)
    hashmap = {}

    # Step 2: Iterate over the array
    for i, num in enumerate(nums):
        # Step 3: Define your condition (e.g., check complement, existence, frequency)
        if some_condition_based_on_hashmap(num, hashmap):
            # Step 4: Return or process result as needed
            return result_based_on_condition

        # Step 5: Update the hashmap
        hashmap_update_logic(num, i, hashmap)

    # Step 6: Handle the case where the condition is never met
    return final_result_if_needed

# Helper functions (replace with actual logic based on the problem)
def some_condition_based_on_hashmap(num, hashmap):
    # Example: return (target - num) in hashmap
    pass

def hashmap_update_logic(num, i, hashmap):
    # Example: hashmap[num] = i
    pass
```

2.3 Common Problems

Summary:

Problem	Description:
**347. Top K Frequent Elements	Given an integer array nums and an integer k, return the k most frequent elements.

- Use a hashMap to count the frequency of each element.
- Sort the map by frequency and return the top k elements.

118. Pascal's Triangle	Given an integer numRows, return the first numRows of Pascal's triangle.

- **Initialize:** `res = [[1]]`.
- **Loop from numRows - 1:**
  - **Pad the PrevRow:** Create `dummy_row` by padding the last row in `res` with zeros at both ends.
  - **Loop 2 from len(prevRow) + 1:** For each position `i`, compute the value `dummy_row[i] + dummy_row[i+1]` and append it to the new row.

Summary:

Problem	**Description:**
73. Set Matrix Zeroes	Given an m x n integer matrix, if an element is 0, set its entire row and column to 0.

- **Record Zero Positions:** Iterate through all elements. If `matrix[i][j] == 0`, append `[i, j]` to list.
- **Row/Column Zeroing:** Set all elements in column `col_ind` to zero and all elements in row `row_ind` to zero using two helpers.

54. Spiral Matrix	Given an m x n matrix, return all elements of the matrix in spiral order.

- **Initialize:** Create an empty list `res`, set boundaries: `top, bottom, left, right`, and current pos (i, j).
- **Loop:** While top <= bottom and left <= right. Use helper functions to achieve the following:
  - Traverse from left to right along the top row and adjust top bdy and check if top > bottom.
  - Traverse from top to bottom along the right column and adjust right bdy and check if left > right.
  - Traverse from right to left along the bottom row and adjust bottom bdy and check if top > bottom.
  - Traverse from bottom to top along the left column and adjust left bdy and check if left > right.

3 Two Pointers

3.1 When to Use?

Summary:

- If we need to find a pair of elements that satisfy a condition.
- If we need to find a subarray that satisfies a condition.

3.2 Slow and Fast Pointers

Algorithm:

1.

3.2.1 Common Problems

Summary:

Problem	Description:
15. 3Sum	Given an array of integers, return all the triplets [nums[i], nums[j], nums[k]] s.t. i != j, i != k, and j != k. • **Tricks:**

3.3 Left and Right Pointers

Algorithm:

1. Initialize two pointers. Some common choices:
    - One at the front and one at the back of the array.
    - Both at the front of the array.
    - Both at the back of the array.

3.3.1 Common Problems

Summary:

Problem	Description:
15. 3Sum	Given an array of integers, return all the triplets [nums[i], nums[j], nums[k]] s.t. i != j, i != k, and j != k.

- **Tricks:**

125. Valid Palindrome	Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

- `s_new = ''.join(char.lower() for char in s if char.isalnum())` to remove non-alphanumeric and lowercase.
- Use front and back pointers. If they not equal, return False. If equal move both pointers.

167. Two Sum II - Input array is sorted	Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a target.

- Use front and back pointers. If > target, move back pointer left. If < target, move front pointer right.

11. Container With Most Water	Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

- **Initialization:**
  - Set `left = 0`, `right = len(height) - 1` to have the width as large as possible.
  - Initialize `maxWater = 0`.
- **Water Area Calculation:** Define helper function `amtWater(height, left, right)`:
  - Compute `height = min(height[left], height[right])`.
  - Compute `width = right - left`.
  - Return `height * width`.
- **Two-Pointer Strategy: While `left <= right`:**
  - Compute area between `left` and `right`, update `maxWater`.
  - Move the pointer at the shorter line inward:
    - If `height[left] <= height[right]`, increment `left`.
    - Else, decrement `right`.

Summary:

Problem	**Description:**
42. Trapping Rain Water	Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

- **Initialization:** If `height` is empty, return 0.
  - Set two pointers: `l = 0`, `r = len(height) - 1`.
  - Initialize `leftMax = height[l]`, `rightMax = height[r]`, `res = 0`.
- **Two-Pointer Traversal: While** `l < r`:
  - If `leftMax < rightMax`:
    - Increment `l`. Update `leftMax = max(leftMax, height[l])`.
    - Accumulate water: `res += leftMax - height[l]`.
  - Else:
    - Decrement `r`. Update `rightMax = max(rightMax, height[r])`.
    - Accumulate water: `res += rightMax - height[r]`.
- **Intuition:**
  - Always move the pointer at the side with the smaller maximum, since the water trapped at that point depends on the limiting side.
  - The water accumulated for a column is determined by the height of the column minus the min of the maximum heights on both sides.

4 Sliding Window

4.1 Fixed Sliding Window

Summary:

- Find a subarray/substring of a fixed size that satisfies a condition.
- Find the maximum or minimum of a subarray of a fixed size.

Algorithm:

```
initialize window_sum = 0
initialize max_result (or other required value)

# Set up initial window
for i in range(0, k):
    window_sum += arr[i]

max_result = window_sum # Initialize result

# Slide the window
for i in range(k, n):
    window_sum += arr[i] - arr[i - k] # Add new element and remove 1st element of prev window
    max_result = max(max_result, window_sum) (or other computation)

return max_result (or other required value)
```

4.1.1 Common Problems

Summary:

Problem	Description:
643. Maximum Average Subarray I	Given an integer array nums and an integer k, return the maximum average value of a subarray of length k.

- Follow template.

567. Permutation in String	Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.

- **Init:** Follow template with window_valid, freqMap_window, freqMap_s1, and fixed size k of len(s1). Rather than sum, get freq of chars.
- **Special Case:** If len(s1) $>$ len(s2), return False.
- **For:** Since contiguous, slide through s2 and update freqMap_window by adding new char and removing old char (make sure to del key if freq $= 0$).
- **Condition:** If freqMap_window == freqMap_s1, return True.

219. Contains Duplicate II	Given an integer array nums and an integer k, return true if there are two distinct indices i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

- **Init:** Follow template with window_freq and fixed size k.
- **Special Case:** If len(nums) < 2, return False.
- **Initial window:** Range(min(k+1, len(nums))) since first window can be smaller than k.

4.2 Dynamic Sliding Window

Summary:

- Find longest or shortest subarray/substring that satisfies a condition.

Algorithm:

```
initialize left = 0
initialize window_state (sum, count, frequency map, etc.)
initialize min_or_max_result

for right in range(n):
    update window_state to include arr[right] # Expand the window

    while window_state violates the condition:
        update min_or_max_result (if needed)
        update window_state to exclude arr[left] # Shrink the window
        move left pointer forward

return min_or_max_result
```

4.2.1 Common Problems

Summary:

Problem	**Description:**
121. Best Time to Buy and Sell Stock	Given an array where the ith element is the price of a stock on day i, find the maximum profit you can achieve. You may not engage in multiple transactions.

- **Buy low, sell high principle**
  - Use left = buy and right = sell, initialized at 0, 1.
  - If price[right] >= price[left], update max profit. Move right pointer since we can still sell for a profit.
  - If price[right] < price[left], move left pointer since we need to find a lower price to buy.
  - Continue until right pointer reaches the end of the array.

3. Longest Substring W/O Repeating Characters	Given a string s, find the length of the longest substring without repeating characters.

- **Init:** Follow template and use frequency map of chars for window_state.
- **While:** If a char is repeated, move left pointer to right by 1 and adjust freqMap until current char is unique.
- **Change:** Compare substring length outside of while with `max_res = max(max_res, right - left + 1)`.

424. Longest Repeating Character Replacement	Given a string s that consists of only uppercase English letters, you can replace any letter with another letter. Find the length of the longest substr containing the same letter after performing at most k replacements.

- **Init:** Follow template and use freqMap of chars for window_state.
- **While:** If the number of replacements needed exceeds k, i.e. `(r - l + 1) - max_freq > k`
  - Move left pointer to right by 1 and adjust freqMap until the condition is satisfied.
- **Change:** Compare substring length outside of while with `max_res = max(max_res, right - left + 1)`.

**76. Minimum Window Substring	Given two strings s and t, return the minimum window substr of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return ""

- **Init:** Set `left = 0`. Initialize `count_t` as frequency map of `t`, `count_s` for current window, and variables `have = 0`, `required = len(count_t)`, `res = [-1, -1]`, and `resLen = infty`.
- **For right in range(n):** Expand window by adding `s[right]` to `count_s`. If relevant char and frequency matches `count_t`, increment `have`.
  - **While have == required:**
    - Update result if current window is smaller w/ coordinates `res = [left, right]` and length `resLen = right - left + 1`.
    - Shrink window by ↓ `count_s[s[left]]`; if below `count_t`, decrement `have`; increment `left`.
- **Return:** `s[res[0]:res[1]+1]` if valid window found, else empty string.

Summary:

Problem	Description:
239. Sliding Window Maximum	Given an integer array nums and an integer k, return the maximum value in each sliding window of size k.

- Hi

5 Binary Search

Algorithm:

```python
def binary_search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

5.1 When to Use?

Summary:

- Use when the input is **sorted** or can be **monotonically mapped**.
- Common for problems involving **searching for a target**, **finding boundaries**, or **min/max constraints**.
- Works on arrays, answer ranges, or implicit search spaces with $\mathcal{O}(\log n)$ complexity.

5.1.1 Common Problems

Summary:

Problems	Description
875. Koko Eating Bananas	Given an array of piles and an integer h, find the minimum eating speed k such that Koko can eat all bananas in h hours.

- Use binary search to find the minimum k w/ l = 0, r = max(piles) since k must be in this range.
- Check if k is valid by calculating the total hours needed to eat all bananas.
  - for p in piles total_hours += math.ceil(p / k)
  - Compare total_hours with h. If hours <= h, update r = mid - 1 and res = mid. Else update l = mid + 1.

**153. Find Minimum in Rotated Sorted Array	Given a rotated sorted array, find the minimum element.

- Initialize `res = nums[0]` as a candidate minimum.
- Set binary search bounds: `l = 0`, `r = len(nums) - 1`.
- While `l <= r`:
  - If `nums[l] < nums[r]`, subarray is sorted; update `res = min(res, nums[l])` and break.
  - Compute midpoint `m = (l + r) // 2`, update `res = min(res, nums[m])`.
  - If `nums[m] >= nums[l]`, left half is sorted; search right: `l = m + 1`.
  - Else, pivot is in left half; search left: `r = m - 1`.
- Return `res` as the minimum element.

**33. Search in Rotated Sorted Array	Given a rotated sorted array, search for a target value. If found, return its index.

-

6 Linked List

Summary: Data structure for storing objects in linear order.

- **Object:** Data and a pointer to the next object.

6.1 When to Use?

Summary:

- Implement other DS: stacks, queues, hash tables.
- Dynamic memory allocation.

6.2 Singly Linked List

Algorithm:

```python
class Node:
    def __init__(self, data):
        self.data = data  # Value stored in the node
        self.next = None  # Pointer to the next node

class SinglyLinkedList:
    def __init__(self,data):
        self.head = Node(data)  # Head of the list

    def operations(self):
        pass
```

6.3 Operations

Summary:

Operation	Time Complexity (WC)
Search	$O(n)$
Insert	$O(n)$
Delete	$O(n)$
Access	$O(n)$

6.4 Doubly Linked List

Algorithm:

```python
class Node:
    def __init__(self, data):
        self.data = data  # Value stored in the node
        self.next = None  # Pointer to the next node
        self.prev = None  # Pointer to the previous node

class DoublyLinkedList:
    def __init__(self,data):
        self.head = Node(data)  # Head of the list

    def operations(self):
        pass
```

6.5 Operations

Summary:

Operation	Time Complexity (WC)
Search	$O(n)$
Insert	$O(n)$
Delete	$O(n)$
Access	$O(n)$

6.6 Circular Linked List

Algorithm:

```python
class Node:
    def __init__(self, data):
        self.data = data  # Value stored in the node
        self.next = None  # Pointer to the next node

class CircularLinkedList:
    def __init__(self,data):
        self.head = Node(data)  # Head of the list
        self.head.next = self.head  # Point to itself

    def operations(self):
        pass
```

6.7 Operations

Summary:

Operation	Time Complexity (WC)
Search	$O(n)$
Insert	$O(n)$
Delete	$O(n)$
Access	$O(n)$

Summary:

Problem	**Description:**
19. Remove Nth Node From End of List	Given a linked list, remove the Nth node from the end of the list.

- **Idea:** Use two pointers (slow and fast) to traverse the list by moving fast pointer N steps ahead.
  - **Conditional:** If fast is None, then this means N = length of list, therefore, remove head by returning head.next.
- Move both pointers until fast reaches the end.
  - Slow pointer will be at the node to be deleted, so store prev node and do appropriate adjustments.

2. Add Two Numbers	Given two non-empty linked lists representing two non-negative integers, add the two numbers and return a linked list.

- **Initialize:** Create a dummy node, use `cur` to traverse as `cur = dummy`.
  - Set `next_val = 0` to store carry between digit additions.
- **Helper Function:** `add_two_nodes(n1, n2, next_val)`
  - Extract values: `val1 = n1.val if n1 else 0`, `val2 = n2.val if n2 else 0`.
  - Compute sum:
    * `rem = (val1 + val2 + next_val) % 10`
    * `next_val = (val1 + val2 + next_val) // 10`.
  - Return updated carry and new node with value `rem`.
- **Traversal Loop:**
  - Continue while at least one of `l1`, `l2` exists or `next_val != 0`.
  - Call `add_two_nodes(l1, l2, next_val)` and link result to `cur.next`.
  - Advance `cur`, and move `l1`, `l2` to their next nodes if they exist.

23. Merge K Sorted Lists	Given an array of k linked lists, each list is sorted in ascending order. Merge all the lists into one sorted linked list and return it.

- **Initialization:** Create a min-heap `heap` to store tuples `(node.val, list_index, node)`.
  - Push the head of each non-empty list in `lists` into the heap.
  - Initialize dummy node `dummy` and pointer `cur = dummy`.
- **Merge Process: While** `heap` is not empty:
  - Pop the smallest element `(min_val, list_ind, node)` from the heap.
  - Append `node` to the result list using `cur.next = node`, then advance `cur`.
  - If `node.next` exists, push `(node.next.val, list_ind, node.next)` into the heap.
- **Return Result:** `return dummy.next` (skipping dummy head).

Summary:

Problem	**Description:**
287. Find the Duplicate Number	Given an array of integers, find the duplicate number.

- **Phase 1: Cycle Detection (Floyd's Tortoise and Hare)**
  - Initialize two pointers: `slow = 0`, `fast = 0`.
  - **Loop:** Move `slow = nums[slow]`, `fast = nums[nums[fast]]`.
  - Continue until `slow == fast`, indicating an intersection point within the cycle.
- **Phase 2: Cycle Entrance (Duplicate Finder)**
  - Initialize a new pointer: `slow2 = 0`.
  - **Loop:** Move both `slow = nums[slow]` and `slow2 = nums[slow2]`.
  - When `slow == slow2`, return the value at that index as the duplicate.
- **Rationale:**
  - Each number is a pointer to the next index; repeated values form a cycle.
  - Detecting the start of the cycle identifies the duplicate.

7 Stack and Queue

7.1 collections.deque

Summary:

```
from collections import deque
```

```
deque(iterable=None, maxlen=None)
```

- Creates a new deque object initialized with elements from `iterable`.
- **Arguments:** `iterable`: optional iterable of elements; `maxlen`: maximum number of elements.

```
d.append(x)
```

- Adds `x` to the right end of the deque.
- **Arguments:** `x`: element to append.

```
d.appendleft(x)
```

- Adds `x` to the left end of the deque.
- **Arguments:** `x`: element to append.

```
d.pop()
```

- Removes and returns the rightmost element.
- **Raises:** `IndexError` if deque is empty.

```
d.popleft()
```

- Removes and returns the leftmost element.
- **Raises:** `IndexError` if deque is empty.

```
d.extend(iterable)
```

- Appends elements from `iterable` to the right side.
- **Arguments:** `iterable`: iterable of elements to append.

```
d.extendleft(iterable)
```

- Appends elements from `iterable` to the left side (in reverse order).
- **Arguments:** `iterable`: iterable of elements to append.

```
d.rotate(n=1)
```

- Rotates the deque `n` steps to the right (left if negative).
- **Arguments:** `n`: number of steps to rotate.

```
d.clear()
```

- Removes all elements from the deque.
- **Postcondition:** deque is empty.

```
d.count(x)
```

- Counts occurrences of `x` in the deque.
- **Arguments:** `x`: element to count.

```
d.remove(value)
```

- Removes the first occurrence of `value`.
- **Raises:** `ValueError` if value is not present.

7.2 Stack

Summary: Data structure that follows Last-In-First-Out (LIFO) order for inserting and removing elements.

- **Array or Linked List:** Used to maintain the linear order of elements.
- **Top Pointer:** Points to the most recently inserted element.

7.2.1 When to Use?

Summary:

- Function call management using a call stack.
- Reversing sequences or backtracking algorithms.
- Syntax parsing and expression evaluation.

7.2.2 Operations

Summary:

Operation	Time Complexity (WC)
Push	$O(1)$
Pop	$O(1)$
Peek	$O(1)$
IsEmpty	$O(1)$

Algorithm:

```python
class Stack:
    def __init__(self):
        self.items = []  # Internal array to store elements

    def push(self, item):
        self.items.append(item)  # Add item to top

    def pop(self):
        if not self.is_empty():
            return self.items.pop()  # Remove and return top element

    def peek(self):
        if not self.is_empty():
            return self.items[-1]  # Return top element without removing

    def is_empty(self):
        return len(self.items) == 0
```

7.2.3 Common Problems

Summary:

Problem	**Description**
20. Valid Parentheses	Check if parentheses are balanced.

- **Initialization:**
  - Create an empty `stack` to track unmatched opening brackets.
  - Define a `closeToOpen` mapping from closing brackets to corresponding opening brackets.
- **String Traversal:**
  - For each character:
    * If it is a closing bracket:
      · Check if the stack is non-empty and the top of the stack matches the corresponding opening bracket. If yes, pop the stack; otherwise, return `False`.
    * If it is an opening bracket, push it onto the stack.
- **Final Check:** Return `True` if the stack is empty (all brackets matched), else `False`.
- **Key Insight:** The stack ensures that brackets are matched in the correct type and order, guaranteeing validity through last-in, first-out (LIFO) behavior.

150. Evaluate Reverse Polish Notation	Evaluate expression in postfix notation.

- **Initialization:** Create an empty `stack` to store intermediate operands and results.
- **Token Traversal:**
  - For each token in the input list:
    * If the token is an operator (`+`, `-`, `*`, `/`):
      · Pop the top two elements from the stack.
      · Apply the operator in the correct order (note: for subtraction and division, order matters).
      · Push the result back onto the stack.
    * If the token is a number:
      · Convert it to an integer and push it onto the stack.
- **Final Result:** Return the top element of the stack, which contains the final evaluated value.
- **Key Insight:** Reverse Polish Notation (postfix notation) allows expressions to be evaluated using a stack without parentheses by always applying operations to the two most recent operands.

22. Generate Parentheses	Generate all combinations of valid parentheses.

- **Initialization:** Create `stack` to build the current sequence and `res` list to store valid sequences.
- **Recursive Backtracking:** Define a recursive function `backtrack(openN, closedN)`:
  - If both `openN` and `closedN` equal `n`, a complete valid sequence is formed. Append it to `res`.
  - If `openN < n`, add an opening bracket `"("`, recurse, then backtrack by removing it.
  - If `closedN < openN`, add a closing bracket `")"`, recurse, then backtrack by removing it.
- **Initial Call:** Start the recursion with `openN = 0`, `closedN = 0`.
- **Return Result:** Return the list `res` containing all valid combinations.
- **Key Insight:** Maintain the constraint that at any point, the number of closing brackets must not exceed the number of opening brackets to ensure sequence validity.

Summary:

Problem	**Description**
739. Daily Temperatures	Find days until a warmer temperature.

- **Initialization:**
  - Create a result list `res` initialized with zeros, having the same length as `temperatures`.
  - Create an empty `stack` to store pairs of (temperature, index).
- **Array Traversal:**
  - Iterate through each temperature and its index:
    * While the stack is not empty and the current temperature is greater than the temperature at the top of the stack:
      · Pop the stack, and for the popped index, set the result as the difference between the current index and the popped index.
    * Push the current temperature and index onto the stack.
- **Return Result:** Return the result list `res`, which contains the number of days to wait for a warmer temperature for each day.
- **Key Insight:** A monotonic decreasing stack is used to efficiently find, for each day, the next day with a higher temperature, achieving linear $O(n)$ time complexity.

7.3 Queue

Summary: Data structure that follows First-In-First-Out (FIFO) order for inserting and removing elements.

- **Array or Linked List:** Used to store elements in sequence.
- **Front and Rear Pointers:** Track the ends for dequeue and enqueue operations.

7.3.1 When to Use?

Summary:

- Scheduling processes in operating systems.
- Handling asynchronous data (e.g., IO Buffers, Event Queues).
- Breadth-First Search in graphs or trees.

7.3.2 Operations

Summary:

Operation	Time Complexity (WC)
Enqueue	$O(1)$
Dequeue	$O(n)$
Peek	$O(1)$
IsEmpty	$O(1)$

Algorithm:

```python
class Queue:
    def __init__(self):
        self.items = []  # Internal array to store elements

    def enqueue(self, item):
        self.items.append(item)  # Add item to the rear

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)  # Remove and return the front element

    def peek(self):
        if not self.is_empty():
            return self.items[0]  # Return front element without removing

    def is_empty(self):
        return len(self.items) == 0
```

8 Trees

8.1 Binary Search Tree (BST)

Summary:

- A binary tree where for each node, left subtree values are smaller, and right subtree values are larger.
- **Balanced vs. Unbalanced:**

$$O(\log(n)) \text{ (balanced)} \leq O(h) \leq O(n) \text{ (unbalanced)}$$

Algorithm:

```python
class Node:
    def __init__(self, key):
        self.val = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def operations(self,_):
        pass
```

8.2 Operations

Summary:

Operation	Time Complexity
Search	$O(h)$
Insert	$O(h)$
Delete	$O(h)$
Find Min/Max	$O(h)$
In-order Traversal	$O(n)$
Pre-order Traversal	$O(n)$
Post-order Traversal	$O(n)$
Level-order Traversal	$O(n)$

8.2.1 Search

Algorithm:

```python
def search(self, key):
    current = self.root
    while current:
        if key == current.val:
            return current
        elif key < current.val:
            current = current.left
        else:
            current = current.right
    return None
```

8.2.2 Insert

Algorithm:

```python
def insert(self, key):
    def _insert(node, key):
        if node is None:
            return TreeNode(key)
        if key < node.val:
            node.left = _insert(node.left, key)
        elif key > node.val:
            node.right = _insert(node.right, key)
        return node
    self.root = _insert(self.root, key)
```

8.2.3 Delete

Algorithm:

```python
def delete(self, key):
    def _delete(node, key):
        if node is None:
            return None
        if key < node.val:
            node.left = _delete(node.left, key)
        elif key > node.val:
```

```python
            node.right = _delete(node.right, key)
        else:
            # Node with one child or no child
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            # Node with two children
            temp = self._find_min(node.right)
            node.val = temp.val
            node.right = _delete(node.right, temp.val)
        return node
    self.root = _delete(self.root, key)
```

8.2.4 Find Min

Algorithm:

```python
def find_min(self, node):
    while node.left is not None:
        node = node.left
    return node
```

8.2.5 Find Max

Algorithm:

```python
def find_max(self, node):
    while node.right is not None:
        node = node.right
    return node
```

8.2.6 DFS In-order Traversal (Left → Root → Right)

Definition: Visit the left subtree, then the root, and finally the right subtree.

- Used for retrieving elements in sorted order from a BST.

Algorithm:

```python
def inorder(node):
    if node:
        inorder(node.left)
        print(node.val)
        inorder(node.right)
```

8.2.7 DFS Pre-order Traversal (Root → Left → Right)

Definition: Visit the root first, then the left subtree, and finally the right subtree.

- Useful for copying or serializing the tree.

Algorithm:

```
def preorder(node):
    if node:
        print(node.val)
        preorder(node.left)
        preorder(node.right)
```

8.2.8 DFS Post-order Traversal (Left → Right → Root)

Definition: Visit the left subtree, then the right subtree, and finally the root.

- Useful for deleting or freeing nodes in memory.

Algorithm:

```
def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.val)
```

8.2.9 BFS Level-order Traversal (Top → Bottom, Left → Right)

Definition: Visit nodes level-level from top to bottom & left to right.

- Useful for finding shortest paths or visualizing layers of a tree.

Algorithm:

```
from collections import deque

def level_order(root):
    if not root:
        return

    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

8.2.10 Common Problems

Summary:

Problem	**Description:**
**226. Invert Binary Tree	Given a binary tree, invert it.

- **Base Case:** If `root` is `None`, return `None`.
- **Swap Subtrees:** Swap the left and right children of the current `root`.
- **Recursive Inversion:**
  - Recursively invert the left subtree by calling `invertTree(root.left)`.
  - Recursively invert the right subtree by calling `invertTree(root.right)`.
- **Return Result:** Return the current `root` after its subtrees have been inverted.

**104. Maximum Depth of Binary Tree	Given a binary tree, find its maximum depth.

- **Recursive DFS:**
  - **Base Case:** If `root` is `None`, return `0`.
  - **Recursive Depth Calculation:**
    - Recursively compute the maximum depth of the left subtree by calling `maxDepth(root.left)`.
    - Recursively compute the maximum depth of the right subtree by calling `maxDepth(root.right)`.
  - **Return Result:** Return 1 plus the maximum of the left and right subtree depths.

**543. Diameter of Binary Tree	Given a binary tree, find its diameter.

- **Initialization:** Initialize a variable `res = 0` to store the maximum diameter found.
- **Depth-First Search (DFS):**
  - If `root` is `None`, return 0.
  - Recursively compute the left subtree depth by calling `dfs(root.left)`.
  - Recursively compute the right subtree depth by calling `dfs(root.right)`.
  - Update `res` as the maximum of its current value and `left + right`.
  - Return $1 + \max(\texttt{left}, \texttt{right})$ to represent the height of the current subtree.
- **Result:**
  - Call `dfs(root)` to start the recursion from the root node.
  - Return the final value of `res`, which represents the diameter of the tree.

110. Balanced Binary Tree	Given a binary tree, check if it is height-balanced.

- **Recursive Depth-First Search (DFS):**
  - If `root` is `None`, return `[True, 0]` (tree is balanced with height 0).
  - Recursively check the left and right subtrees by calling `dfs(root.left)` and `dfs(root.right)`.
  - A node is balanced if:
    - Both left and right subtrees are balanced.
    - The height difference between the left and right subtrees is at most 1.
  - Return a list `[isBalanced, height]`, where:
    - `isBalanced` is a boolean indicating subtree balance.
    - `height` is $1 + \max(\texttt{left height}, \texttt{right height})$.
- **Return Result:** Return 1st element of the result from `dfs(root)`, indicating whether entire tree is bal.

Summary:

Problem	**Description:**
100. Same Tree	Given two binary trees, check if they are the same.

- **Base Cases:**
  - If both `p` and `q` are `None`, return `True`.
  - If only one of `p` or `q` is `None`, or their values differ, return `False`.
- **Recursive Comparison:**
  - Recursively check if the left subtrees `p.left` and `q.left` are identical.
  - Recursively check if the right subtrees `p.right` and `q.right` are identical.
  - Return `True` only if both left and right subtree comparisons return `True`.

235. Lowest Common Ancestor of a BST	Given a BST and two nodes, find their lowest common ancestor.

- **Initialization:** Set `cur` to the `root` node of the tree.
- **Iterative Traversal:** While `cur` is not `None`:
  - If both `p.val` and `q.val` are greater than `cur.val`, move to `cur.right`.
  - Else if both `p.val` and `q.val` are less than `cur.val`, move to `cur.left`.
  - Otherwise, `cur` is the split point where paths to `p` and `q` diverge, and thus `cur` is the lowest common ancestor (LCA).
- **Return Result:** Return the node `cur` when the split point is found.

102. Binary Tree Level Order Traversal	Given a binary tree, return its level order traversal.

- **Initialization:** Create an empty list `res` to store nodes level-by-level.
- **Depth-First Search (DFS):** Define a recursive function `dfs(node, depth)`:
  - If `node` is `None`, return immediately.
  - If `depth` equals the length of `res`, append a new empty list for this depth level.
  - Append `node.val` to the corresponding depth list.
  - Recursively call `dfs(node.left, depth + 1)` and `dfs(node.right, depth + 1)`.
- **Return Result:**
  - Call `dfs(root, 0)` to start traversal.
  - Return the list `res` containing all levels.

98. Validate Binary Search Tree	Given a binary tree, check if it is a valid BST.

- **Initialization:**
  - If `root` is `None`, return `True`.
  - Initialize a queue `q` with a tuple containing the root node and its valid range `(-∞, ∞)`.
- **Breadth-First Search (BFS) Traversal:** While the queue is not empty:
  - Dequeue a node along with its valid value bounds `(left, right)`.
  - If the node's value is not strictly between `left` and `right`, return `False`.
  - If the node has a left child, enqueue it with updated bounds `(left, node.val)`.
  - If the node has a right child, enqueue it with updated bounds `(node.val, right)`.
- **Return Result:** After completing traversal without violations, return `True`.

230. Kth Smallest Element in a BST	Given a BST and an integer k, find the kth smallest element.

- **Initialization:** Create an empty list `arr` to store node values in ascending order.
- **Depth-First Search (DFS) In-Order Traversal:** Define a recursive function `dfs(node)`:
  - If `node` is `None`, return immediately.
  - Recursively call `dfs(node.left)` to visit the left subtree.
  - Append `node.val` to `arr`.
  - Recursively call `dfs(node.right)` to visit the right subtree.
- **Return Result:** DFS starting from the `root`, return `arr[k-1]`, which is the k^{th} smallest element.

8.2.11 BST-based Sets and Maps

Summary:

- **BST Set**: Stores unique values in sorted order. Supports insert, search, delete.
- **BST Map**: Associates keys with values, maintaining keys in sorted order.
- Can be implemented using self-balancing trees (e.g., AVL, Red-Black Tree) for O(log n) operations.
- Useful for range queries, floor/ceiling lookups, and ordered iteration.

Algorithm:

```python
class BSTSet:
    def __init__(self):
        self.root = None

    def add(self, val):
        self.root = insert_bst(self.root, val)

    def contains(self, val):
        return search_bst(self.root, val) is not None

    def remove(self, val):
        self.root = delete_bst(self.root, val)

class BSTMap:
    def __init__(self):
        self.root = None

    def put(self, key, value):
        self.root = self._put(self.root, key, value)

    def _put(self, node, key, value):
        if not node:
            return TreeNode((key, value))
        if key < node.val[0]:
            node.left = self._put(node.left, key, value)
        elif key > node.val[0]:
            node.right = self._put(node.right, key, value)
        else:
            node.val = (key, value)
        return node

    def get(self, key):
        node = self.root
        while node:
            if key < node.val[0]:
                node = node.left
            elif key > node.val[0]:
                node = node.right
            else:
                return node.val[1]
        return None
```

9 Heaps and Priority Queues

9.1 Heap

Summary:

- A **heap** is a complete binary tree where each node follows the heap property:
  - **Max heap:** Largest key at root, where every parent node is **greater than or equal to** its children.
  - **Min heap:** Smallest key at root, where every parent node is **less than or equal to** its children.
- **Balanced Tree:** $h = \log n$
- **Indexing:** Given a node at index i in the array, assuming 1-based indexing:
  1. **Parent:** $\text{parent(i)} = \left\lfloor \frac{i}{2} \right\rfloor$
  2. **Left child:** $\text{leftchild(i)} = 2i$
  3. **Right child:** $\text{rightchild(i)} = 2i + 1$

9.2 Heapq

Summary:

```
heapq.heapify(x)
```

- Transforms a list `x` into a valid min-heap in-place.
- **Arguments:** `x`: list to be heapified.

```
heapq.heappush(heap, item)
```

- Inserts `item` into `heap` while maintaining the heap invariant.
- **Arguments:** `heap`: list representing a heap; `item`: element to insert.

```
heapq.heappop(heap)
```

- Removes and returns the smallest element from the heap.
- **Arguments:** `heap`: non-empty list representing a valid heap.

```
heapq.heappushpop(heap, item)
```

- Pushes `item` onto the heap, then pops and returns the smallest element.
- **Arguments:** `heap`: valid heap; `item`: element to insert.

```
heapq.heapreplace(heap, item)
```

- Pops and returns the smallest element, then inserts `item` into the heap.
- **Arguments:** `heap`: non-empty valid heap; `item`: element to insert.

```
heapq.nlargest(n, iterable)
```

- Returns the `n` largest elements from `iterable` in descending order.
- **Arguments:** `n`: number of elements; `iterable`: list or other iterable.

```
heapq.nsmallest(n, iterable)
```

- Returns the `n` smallest elements from `iterable` in ascending order.
- **Arguments:** `n`: number of elements; `iterable`: list or other iterable.

Notes:

- In general, when pushing tuples to a heap in Python:
  - The first element determines the primary priority.
  - If equal, subsequent elements serve as tie-breakers.

Algorithm:

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left(self, i):
        return 2 * i + 1

    def right(self, i):
        return 2 * i + 2

    def operations(self,_):
        pass
```

9.3 Operations

Summary:

Operation	Time Complexity
Insert	$O(\log n)$
Extract Min	$O(\log n)$
Get Min	$O(1)$
Heapify	$O(n)$
Build Heap	$O(n)$
Search	$O(n)$
Delete	$O(n)$
Heap Sort	$O(n \log n)$

9.3.1 Insert

Algorithm:

```
def insert(self, key):
    self.heap.append(key)
    i = len(self.heap) - 1
    while i != 0 and self.heap[self.parent(i)] > self.heap[i]:
        self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]
        i = self.parent(i)
```

9.3.2 Heapify

Algorithm: Restores the heap property by moving a node down the tree to its correct position.

```
def heapify(self, i):
    smallest = i
    l = self.left(i)
    r = self.right(i)

    if l < len(self.heap) and self.heap[l] < self.heap[smallest]:
        smallest = l
    if r < len(self.heap) and self.heap[r] < self.heap[smallest]:
        smallest = r

    if smallest != i:
        self.heap[i], self.heap[smallest] = self.heap[smallest], self.heap[i]
        self.heapify(smallest)
```

9.3.3 Extract Min

Algorithm:

```
def extract_min(self):
    if not self.heap:
        return None
    if len(self.heap) == 1:
        return self.heap.pop()
    root = self.heap[0]
    self.heap[0] = self.heap.pop()
    self.heapify(0)
```

```python
    return root
```

9.3.4 Get Min

Algorithm: Min-heap always has the smallest element at the root.

```python
def get_min(self):
    return self.heap[0] if self.heap else None
```

9.3.5 Build Heap

Algorithm:

```python
def build_heap(self, arr):
    self.heap = arr[:]
    for i in range(len(self.heap) // 2 - 1, -1, -1):
        self.heapify(i)
```

9.3.6 Search

Algorithm:

```python
def search(self, key):
    return key in self.heap
```

9.3.7 Delete

Algorithm:

```python
def delete(self, key):
    try:
        index = self.heap.index(key)
        self.heap[index] = self.heap[-1]
        self.heap.pop()
        if index < len(self.heap):
            self.heapify(index)
            parent = self.parent(index)
            while index > 0 and self.heap[parent] > self.heap[index]:
                self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]
                index = parent
                parent = self.parent(index)
    except ValueError:
        pass
```

9.3.8 Heap Sort

Algorithm:

```python
def heap_sort(self):
    sorted_list = []
    original = self.heap[:]
    while self.heap:
        sorted_list.append(self.extract_min())
    self.heap = original
```

```
    return sorted_list
```

9.3.9 Common Problem

Summary:

Problem	Description
621. Task Scheduler	Given a list of tasks and a cooldown period, find the least time to finish all tasks.

- Use `Counter` to count the frequency of each task.
- Use a max-heap to store tasks by frequency: `(-freq, task)`.
- Use a queue to track cooldowns: `(-freq, ready_time)`.
- While either heap or queue is non-empty:
  - Increment `time`.
  - If heap is non-empty, pop task, decrement frequency, and if not 0, add to queue with `time + n`.
  - If the front of the queue is ready (`ready_time == time`), pop and push it back into the heap.

Summary:

Problem	Description
239. Sliding Window Maximum	Given an array and a window size, find the maximum in each sliding window.

- **Initialization:**
  - Create an empty max-heap `heap` using negated values: `(-value, index)`.
  - Fill the initial window with the first `k` elements.
  - Append the maximum (top of the heap) to result: `res.append(-heap[0][0])`.
- **Sliding the Window:**
  - For each new index `i`, push `(-nums[i], i)` into the heap.
  - **Lazy Removal:** While the top element's index is outside the window (`heap[0][1]` $\leq$ `i - k`), remove it using `heapq.heappop()`.
    - Heap may contain elements outside the current window, but they are ignored. Only care about root element.
  - Append the current maximum to the result: `res.append(-heap[0][0])`.
- **Return:** The list `res` containing all sliding window maximums.

10 Graphs

Summary:

Algorithm	Time Complexity	Space Complexity
BFS	$O(V+E)$	$O(V)$
DFS	$O(V+E)$	$O(V)$
Topological Sort (DFS)	$O(V+E)$	$O(V)$

10.1 Breadth-First Search (BFS)

Summary:
- Use when exploring nodes layer-by-layer, typically in unweighted graphs or grids.
- Ideal for finding the shortest path, level order traversal, or minimum number of steps.
- Queue-based traversal ensures nodes are visited in order of increasing distance from the source.

Algorithm:

```python
from collections import deque

def bfs(start, graph):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft() # FIFO (BFS)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

10.1.1 Common Problems

Summary:

Problem	Description:
994. Rotting Oranges	Given a m x n grid of oranges, where 0 = empty cell, 1 = fresh orange, and 2 = rotten orange, and where fresh oranges rot if adjacent to a rotten orange, return the minimum time required for all oranges to become rotten.

- **Initialization:**
  - Create a queue `q` to store coordinates of initially rotten oranges.
  - Count total fresh oranges `fresh = 0`, and set `time = 0`.
  - Traverse `grid`:
    * If `grid[r][c] == 1`, increment `fresh`.
    * If `grid[r][c] == 2`, append `(r, c)` to `q`.
- **BFS Propagation: While `fresh > 0 and q`** is not empty:
  - Go through all rotten oranges in `q` and check their neighbors to mark it as rotten.
  - Increment `time` after each level (i.e. after processing all rotten oranges at the current level).
- **Return Result:** If `fresh == 0`, return `time`; else return `-1`.

**417. Pacific Atlantic Water Flow	Given an m x n matrix of non-negative integers representing the height of each cell, return the coordinates of cells that can flow to both the Pacific & Atlantic oceans.

- **Initialization:**
  - Define `ROWS, COLS` as the dimensions of the input matrix.
  - Define `directions` as the 4 possible adjacent moves (up, down, left, right).
  - Create two 2D boolean matrices: `pac` and `atl`, indicating cells reachable by Pacific and Atlantic oceans.
- **Construct Ocean Borders:**
  - Initialize Pacific ocean border with all top row and leftmost column coordinates.
  - Initialize Atlantic ocean border with all bottom row and rightmost column coordinates.
- **BFS Traversal Function:**
  - Perform breadth-first search (BFS) from all coordinates along each ocean's border.
  - For each visited cell, enqueue adjacent cells that:
    * Are within bounds.
    * Are not yet marked as reachable.
    * Have equal or greater height (ensuring water can flow from neighbor to current).
- **Mark Reachable Cells:**
  - Call `bfs(pacific, pac)` and `bfs(atlantic, atl)` to fill in reachable matrices.
- **Collect Intersection Points:** Iterate through all cells in the matrix.
  - If a cell is marked `True` in both `pac` and `atl`, append it to the result.
- **Return:** Return list of coordinates where water can flow to both oceans.

130. Surrounded Regions	Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

- **Border Traversal:** Iterate through all border cells (first and last rows, first and last columns).
  - Enqueue all 'O's found on the border into the queue `q`.
- **BFS Flood Fill:**
  - For each 'O' in the queue, mark it and all connected 'O's as visited by changing them to temporary symbol '#'.
  - Only move to valid neighbors that are within bounds, are 'O', and unvisited.
- **Final Transformation:** Traverse the entire board.
  - Change all remaining 'O's (not connected to border) to 'X' and all temporary '#' markers back to 'O'.
- **Key Insight:** Only 'O's connected to the border should be preserved.

10.2 Depth-First Search (DFS)

Summary:

- Use when traversing all nodes or paths in **trees**, **graphs**, or **matrices**.
- Ideal for problems involving **backtracking**, **recursion**, or exploring all **connected components**.
- Can be implemented recursively or iteratively with a stack.
- Maintain a `visited` set or matrix to avoid revisiting nodes.
- Useful for **topological sorting**, **cycle detection**, and **pathfinding**.

Algorithm:

```
from collections import deque

def dfs(start, graph):
    visited = set()
    stack = deque([start])
    visited.add(start)

    while stack:
        node = stack.pop()  # LIFO (DFS)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(neighbor)
```

Algorithm:

```
def dfs(node, visited):
    if node in visited:
        return

    visited.add(node)

    for neighbor in graph[node]:
        dfs(neighbor, visited)
```

10.2.1 Common Problems

Summary:

Problem	Description:
200. Number of Islands	Given a 2D grid of '1's (land) and '0's (water), count the number of islands.

- Use DFS or BFS to explore all connected '1's and mark them as visited.
- Increment the island count for each unvisited '1'.

695. Max Area of Island	Given a 2D grid of '1's (land) and '0's (water), find the maximum area of an island.

- Use DFS or BFS to explore all connected '1's and calculate the area.
- Keep track of the maximum area encountered during the traversal.

79. Word Search	Given a 2D board and a word, check if the word exists in the grid.

- Use DFS to explore all possible paths in the grid.
- Mark cells as visited to avoid revisiting.
- Backtrack if the current path does not lead to a solution.

133. Clone Graph	Given a reference to a node in a connected undirected graph, return a deep copy of the graph.

- **Initialization:** Create a hash map `oldToNew` to store mappings from original nodes to their cloned nodes.
- **DFS:** Define `dfs(node)` to recursively clone the graph.
  - **Base Case:** If `node` already in `oldToNew`, return the cloned node.
  - **Clone Creation:** Create a new `Node(node.val)`, store in `oldToNew`.
  - **Neighbor Cloning:** For each `neigh` in `node.neighbors`, recursively clone and append to `copy.neighbors`.
- **Entry Point:** Return `dfs(node)` if `node` is not `None`; otherwise return `None`.

10.3 Topological Sort

Summary:

- **Overview:** Produces a total ordering from partial ordering.
- **DAG:** $G = (V, E)$ must be a DAG to produce a valid topological sorting.
- Given a DAG, create a linear (total) order out of the partial order $\rightarrow$ "serialize" these events
  - **Intuition:** Arranges the vertices of a DAG in a linear order such that for every directed edge $u \rightarrow v$, vertex u appears before v.

Algorithm:

```
from collections import defaultdict, deque

def topological_sort(num_nodes, edges):
    # Build adjacency list and in-degree count
    graph = defaultdict(list)
    in_degree = [0] * num_nodes

    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    # Start with all nodes that have in-degree 0
    queue = deque([i for i in range(num_nodes) if in_degree[i] == 0])
    topo_order = []

    while queue:
        node = queue.popleft()
        topo_order.append(node)

        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If not all nodes are processed, there is a cycle
    if len(topo_order) != num_nodes:
        return []  # or raise an error

    return topo_order
```

11 Sorting

Summary:

Name	Space	Time (BC,AC,WC)	Prop. (In-place, Stable, D&C)
Comparison Based			
Merge Sort	$\Theta(n)$	$\Theta(n \log n)$	$\cancel{\text{I}}$P,S,DC
• **DS:** Auxiliary arrays			
Quick Sort	$\Theta(1)$, $\Theta(\log n)$,$\Theta(n)$	$\Theta(n \log n)$, $\Theta(n \log n)$,$\Theta(n^2)$	IP, $\cancel{\text{S}}$,DC
• **DS:** Original array (in-place) • Space complexity depends on implementation.			
Heap Sort	$\Theta(1)$	$\Theta(n \log n)$	IP, $\cancel{\text{S}}$,$\cancel{\text{D}}$C
• **DS:** Binary heap • **Time Complexity:** – n: Number of elements in the input array that need to be sorted. – $h = \log n$: Height of the binary heap (can be changed) • Space complexity doesn't include the input.			
Bubble Sort	$\Theta(1)$	$\Theta(n)$,$\Theta(n^2)$,$\Theta(n^2)$	IP,S,$\cancel{\text{D}}$C
• **DS:** Original array (in-place)			
Selection Sort	$\Theta(1)$	$\Theta(n^2)$	IP,$\cancel{\text{S}}$,$\cancel{\text{D}}$C
• **DS:** Original array (in-place)			
Insertion Sort	$\Theta(1)$	$\Theta(n)$,$\Theta(n^2)$,$\Theta(n^2)$	IP,S,$\cancel{\text{D}}$C
• **DS:** Original array (in-place)			
Non-Comparison Based			
Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$ if $k \gg O(n)$, $\Theta(n)$ if $k \leq O(n)$	$\cancel{\text{I}}$P, S,$\cancel{\text{D}}$C
• **DS:** Auxiliary arrays • **Assumption:** Elements are integers ranging from 0 to k. • n: Size of array • k: Range of numbers (i.e. $[0, ..., k]$)			
Radix Sort	$\Theta(n+k)$	$\Theta(d \cdot (n+k))$	$\cancel{\text{I}}$P, S,$\cancel{\text{D}}$C
• **DS:** Auxiliary arrays • **Assumption:** All elements have $\leq d$-digits • **One pass time complexity:** $O(n+k)$ (i.e. counting sort) • **Note:** If $k = d = O(1)$, then $O(n)$ time complexity. • n: Size of array • k: Range of numbers • d: Number of digits			

11.1 Stable, In-place, and Divide and Conquer

Definition:

- **Stable:** Relative order of ties is maintained.
  - e.g. $[2_a, 3, 2_b, 1] \to [1, 2_a, 2_b, 3]$
- **In-place sorting:** $O(1)$ extra space.
- **Lower bound on comparison-based sorting:** No CBS algorithm on **unrestricted** range is better than $\Omega(n \log n)$

11.2 Merge Sort

Algorithm:

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Divide
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # Conquer: Merge two sorted halves
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    # Merge two sorted lists
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

11.3 Quick Sort

Algorithm:

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]  # Choose middle element as pivot
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    # Recursively sort left and right, then concatenate
    return quick_sort(left) + middle + quick_sort(right)
```

12 Images

12.1 2D Convolution Operations

Notes:

1. **1. Output Dimensions**
   The output height and width of a 2D convolution are given by:

$$\text{out\_height} = \left\lfloor \frac{\text{in\_height} + 2 \cdot \text{padding}_h - \text{effective\_kernel}_h}{\text{stride}_h} \right\rfloor + 1$$

$$\text{out\_width} = \left\lfloor \frac{\text{in\_width} + 2 \cdot \text{padding}_w - \text{effective\_kernel}_w}{\text{stride}_w} \right\rfloor + 1$$

2. **2. Effective Kernel Size (with Dilation)**
   The effective kernel size when dilation is applied:

$$\text{effective\_kernel}_h = \text{kernel\_height} + (\text{kernel\_height} - 1) \cdot (\text{dilation}_h - 1)$$

$$\text{effective\_kernel}_w = \text{kernel\_width} + (\text{kernel\_width} - 1) \cdot (\text{dilation}_w - 1)$$

3. **3. Convolution Operation (Batch, Channel-aware)**
   The general convolution operation for a batch of input tensors is:

$$\text{output}[b, c_{\text{out}}, h_{\text{out}}, w_{\text{out}}] = \sum_{c_{\text{in}}} \sum_{k_h} \sum_{k_w} (\text{input}[b, c_{\text{in}}, h_{\text{in}} + k_h \cdot \text{dilation}_h, w_{\text{in}} + k_w \cdot \text{dilation}_w] \cdot \text{filter}[c_{\text{out}}, c_{\text{in}}, k_h, k_w])$$

   where:

$$h_{\text{in}} = h_{\text{out}} \cdot \text{stride}_h, \quad w_{\text{in}} = w_{\text{out}} \cdot \text{stride}_w$$

12.2 Common Problems

Summary:

Problem	Description
661. Image Smoother	Given an image represented by a 2D array, smooth the image by averaging the pixel values of each pixel and its neighbors.

- Loop through the cols and rows of the image, then
  - total sum for each pixel $= \sum_{x,y \in \text{neighbours}} \text{image}[x][y] = \sum_{x=i-1}^{i+1} \sum_{y=j-1}^{j+1} \text{image}[x][y]$
    - If x or y is out of bounds, ignore it.
  - count $= \sum_{x=i-1}^{i+1} \sum_{y=j-1}^{j+1} 1$
  - average = total sum//count
- result$[i][j]$ = average

Problem	Description
832. Flipping an Image	Given a binary matrix, flip the image horizontally and invert it.

- Loop through the rows of the image, then use .reverse() to flip the row horizontally.
- Double for loop to invert image (change 0 to 1 and 1 to 0).

Problem	Description
48. Rotate Image	Given an n x n 2D matrix, rotate the image 90 degrees clockwise.

- Transpose the matrix (swap rows and columns) if $i < j$, then matrix$[i][j] \xLeftrightarrow{\text{swap}}$ matrix$[j][i]$.
- Reverse each row.

Problem	Description
**835. Image Overlap	Given two images represented by 2D arrays, find the maximum overlap between the two images.

- Try all possible translations of img1.
- For each translation, calculate the overlap with img2.