

Leetcode

Hanhee Lee

April 15, 2025

Contents

| | | |
|----------|--|-----------|
| 1 | General | 2 |
| 1.1 | Interviewer Considerations | 2 |
| 1.2 | Steps for Success During the Technical Interview | 2 |
| 1.3 | Common Mistakes to Avoid | 2 |
| 1.4 | Syntax | 3 |
| 2 | Arrays and Hashing | 4 |
| 2.1 | When to Use? | 4 |
| 2.2 | Hashing | 4 |
| 2.3 | Common Problems | 5 |
| 3 | Two Pointers | 7 |
| 3.1 | When to Use? | 7 |
| 3.2 | Slow and Fast Pointers | 7 |
| 3.2.1 | Common Problems | 7 |
| 3.3 | Left and Right Pointers | 8 |
| 3.3.1 | Common Problems | 8 |
| 4 | Sliding Window | 9 |
| 4.1 | Fixed Sliding Window | 9 |
| 4.1.1 | Common Problems | 10 |
| 4.2 | Dynamic Sliding Window | 11 |
| 4.2.1 | Common Problems | 12 |
| 5 | Binary Search | 13 |
| 5.1 | When to Use? | 13 |
| 5.1.1 | Common Problems | 14 |
| 6 | Linked List | 15 |
| 6.1 | When to Use? | 15 |
| 6.2 | Singly Linked List | 15 |
| 6.3 | Operations | 15 |
| 6.4 | Common Problems | 16 |
| 7 | Images | 18 |
| 7.1 | 2D Convolution Operations | 18 |
| 7.2 | Common Problems | 18 |
| 8 | Trees | 20 |
| 8.1 | Binary Search Tree (BST) | 20 |
| 8.2 | Operations | 21 |
| 8.2.1 | Common Problems | 23 |
| 8.2.2 | BST-based Sets and Maps | 24 |
| 8.3 | Breadth-First Search (BFS) | 25 |
| 8.3.1 | Common Problems | 26 |
| 8.4 | Depth-First Search (DFS) | 27 |

| | | |
|-------|---------------------------|----|
| 8.4.1 | Common Problems | 28 |
|-------|---------------------------|----|

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`).

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:

```
1 def solve_problem(nums):
2     # Step 1: Initialize the hashmap (e.g., for frequency, index, or existence check)
3     hashmap = {}
4
5     # Step 2: Iterate over the array
6     for i, num in enumerate(nums):
7         # Step 3: Define your condition (e.g., check complement, existence, frequency)
8         if some_condition_based_on_hashmap(num, hashmap):
9             # Step 4: Return or process result as needed
10            return result_based_on_condition
11
12        # Step 5: Update the hashmap
13        hashmap_update_logic(num, i, hashmap)
14
15    # Step 6: Handle the case where the condition is never met
16    return final_result_if_needed
17
18 # Helper functions (replace with actual logic based on the problem)
19 def some_condition_based_on_hashmap(num, hashmap):
20     # Example: return (target - num) in hashmap
21     pass
22
23 def hashmap_update_logic(num, i, hashmap):
24     # Example: hashmap[num] = i
25     pass
```

2.3 Common Problems

Summary:

| Problem | Description: |
|--------------------------------|--|
| 217. Contains Duplicate | Given an integer array <code>nums</code> , return <code>true</code> if any value appears at least twice. <ul style="list-style-type: none"> • Use a set to store the elements. If an element is already in the set, return <code>True</code>. • Otherwise, add it to the set. |
| 242. Valid Anagram | Given two strings <code>s</code> and <code>t</code> , return <code>true</code> if <code>t</code> is an anagram of <code>s</code> and <code>false</code> otherwise. <ul style="list-style-type: none"> • Use a <code>HashMap</code> to count the frequency of each character in <code>s</code> and <code>t</code>. • If the frequency maps are equal, return <code>True</code>. Otherwise, return <code>False</code>. |
| 1. Two Sum | Given an array of integers, return indices of the two numbers s.t. they add up to a specific target. <ul style="list-style-type: none"> • Tricks: <ul style="list-style-type: none"> – Use a <code>HashMap</code> to store the indices of the elements, <code>prevMap[nums[i]] = i</code> – For each element, check if the <code>target - nums[i]</code> is in the map. – If it is, return the index of the <code>target - nums[i]</code> (from <code>prevMap</code>) and <code>i</code>. Otherwise, add <code>target - nums[i]</code>. |
| **49. Group Anagrams | Given an array of strings, group the anagrams together. <ul style="list-style-type: none"> • Use a <code>HashMap</code> to store a tuple of count of each char as the key and the list of words as the value. • For each word, create a tuple of count of each char and add the word to the list in the map. • Finally, return the values of the map. |
| **347. Top K Frequent Elements | Given an integer array <code>nums</code> and an integer <code>k</code> , return the <code>k</code> most frequent elements. <ul style="list-style-type: none"> • Use a <code>HashMap</code> to count the frequency of each element. • Sort the map by frequency and return the top <code>k</code> elements. |
| 118. Pascal's Triangle | Given an integer <code>numRows</code> , return the first <code>numRows</code> of Pascal's triangle. <ul style="list-style-type: none"> • Initialize: <code>res = [[1]]</code>. • Loop from <code>numRows - 1</code>: <ul style="list-style-type: none"> – Pad the PrevRow: Create <code>dummy_row</code> by padding the last row in <code>res</code> with zeros at both ends. – Loop 2 from <code>len(prevRow) + 1</code>: For each position <code>i</code>, compute the value <code>dummy_row[i] + dummy_row[i+1]</code> and append it to the new row. |

Summary:**Problem**

73. Set Matrix Zeroes

Description:

Given an $m \times n$ 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 $m \times n$ 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. $i \neq j$, $i \neq k$, and $j \neq k$. |
| <ul style="list-style-type: none"> • Tricks: | |
| 125. Valid Palindrome | Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases. |
| <ul style="list-style-type: none"> • <code>s_new = ".join(char.lower() for char in s if char.isalnum())</code> 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. |
| <ul style="list-style-type: none"> • Use front and back pointers. If $>$ target, move back pointer left. If $<$ target, move front pointer right. | |

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. $i \neq j$, $i \neq k$, and $j \neq k$. |
| • Tricks: | |
| 125. Valid Palindrome | Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases. |
| • <code>s_new = ".join(char.lower() for char in s if char.isalnum())</code> 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. | |

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:

```
1 initialize window_sum = 0
2 initialize max_result (or other required value)
3
4 # Set up initial window
5 for i in range(0, k):
6     window_sum += arr[i]
7
8 max_result = window_sum # Initialize result
9
10 # Slide the window
11 for i in range(k, n):
12     window_sum += arr[i] - arr[i - k] # Add new element and remove 1st element of prev window
13     max_result = max(max_result, window_sum) (or other computation)
14
15 return max_result (or other required value)
16
```

4.1.1 Common Problems

Summary:

| Problem | Description: |
|---------------------------------|---|
| 643. Maximum Average Subarray I | Given an integer array <code>nums</code> and an integer <code>k</code> , return the maximum average value of a subarray of length <code>k</code> . <ul style="list-style-type: none"> Follow template. |
| 567. Permutation in String | Given two strings <code>s1</code> and <code>s2</code> , return true if <code>s2</code> contains a permutation of <code>s1</code> , or false otherwise. <ul style="list-style-type: none"> Init: Follow template with <code>window_valid</code>, <code>freqMap_window</code>, <code>freqMap_s1</code>, and fixed size <code>k</code> of <code>len(s1)</code>. Rather than sum, get freq of chars. Special Case: If <code>len(s1) > len(s2)</code>, return False. For: Since contiguous, slide through <code>s2</code> and update <code>freqMap_window</code> by adding new char and removing old char (make sure to del key if freq = 0). Condition: If <code>freqMap_window == freqMap_s1</code>, return True. |
| 219. Contains Duplicate II | Given an integer array <code>nums</code> and an integer <code>k</code> , return true if there are two distinct indices <code>i</code> and <code>j</code> in the array such that <code>nums[i] == nums[j]</code> and <code>abs(i - j) <= k</code> . <ul style="list-style-type: none"> Init: Follow template with <code>window_freq</code> and fixed size <code>k</code>. Special Case: If <code>len(nums) < 2</code>, return False. Initial window: <code>Range(min(k+1, len(nums)))</code> since first window can be smaller than <code>k</code>. |

4.2 Dynamic Sliding Window

Summary:

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

Algorithm:

```
1 initialize left = 0
2 initialize window_state (sum, count, frequency map, etc.)
3 initialize min_or_max_result
4
5 for right in range(n):
6     update window_state to include arr[right] # Expand the window
7
8     while window_state violates the condition:
9         update min_or_max_result (if needed)
10        update window_state to exclude arr[left] # Shrink the window
11        move left pointer forward
12
13 return min_or_max_result
```

4.2.1 Common Problems

Summary:

| Problem | Description: |
|---|--|
| 121. Best Time to Buy and Sell Stock | <p>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.</p> <ul style="list-style-type: none"> • Buy low, sell high principle <ul style="list-style-type: none"> – Use $left = buy$ and $right = sell$, initialized at 0, 1. – If $price[right] \geq price[left]$, 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 | <p>Given a string s, find the length of the longest substring without repeating characters.</p> <ul style="list-style-type: none"> • Init: Follow template and use frequency map of chars for <code>window_state</code>. • While: If a char is repeated, move left pointer to right by 1 and adjust <code>freqMap</code> until current char is unique. • Change: Compare substring length outside of while with <code>max_res = max(max_res, right - left + 1)</code>. |
| 424. Longest Repeating Character Replacement | <p>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.</p> <ul style="list-style-type: none"> • Init: Follow template and use <code>freqMap</code> of chars for <code>window_state</code>. • While: If the number of replacements needed exceeds k, i.e. $(r - l + 1) - \max_freq > k$ <ul style="list-style-type: none"> – Move left pointer to right by 1 and adjust <code>freqMap</code> until the condition is satisfied. • Change: Compare substring length outside of while with <code>max_res = max(max_res, right - left + 1)</code>. |
| **76. Minimum Window Substring | <p>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 ""</p> <ul style="list-style-type: none"> • Init: Set $left = 0$. Initialize <code>count_t</code> as frequency map of t, <code>count_s</code> for current window, and variables <code>have = 0</code>, <code>required = len(count_t)</code>, <code>res = [-1, -1]</code>, and <code>resLen = \infty</code>. • For right in range(n): Expand window by adding $s[right]$ to <code>count_s</code>. If frequency matches <code>count_t</code>, increment <code>have</code>. • While have == required: <ul style="list-style-type: none"> – Update result if current window is smaller. – Shrink window by decrementing <code>count_s[s[left]]</code>; if below <code>count_t</code>, decrement <code>have</code>; increment <code>left</code>. • Return: $s[res[0]:res[1]+1]$ if valid window found, else empty string. |
| 239. Sliding Window Maximum | <p>Given an integer array $nums$ and an integer k, return the maximum value in each sliding window of size k.</p> <ul style="list-style-type: none"> • Init: Use deque to store indices of elements in the current window. • For right in range(n): <ul style="list-style-type: none"> – Remove indices that are out of the current window. – Remove indices from the back of the deque while the current element is greater than the element at those indices. – Append the current index to the deque. – If the window size is reached, append the maximum (element at the front of the deque) to the result list. |

5 Binary Search

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.

Algorithm:

```
1 def binary_search(nums, target):
2     left, right = 0, len(nums) - 1
3
4     while left <= right:
5         mid = left + (right - left) // 2
6
7         if nums[mid] == target:
8             return mid
9         elif nums[mid] < target:
10            left = mid + 1
11        else:
12            right = mid - 1
13
14    return -1
```

5.1.1 Common Problems

Summary:

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:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data # Value stored in the node
4         self.next = None # Pointer to the next node
5
6 class SinglyLinkedList:
7     def __init__(self, data):
8         self.head = Node(data) # Head of the list
9
10    def operations(self):
11        pass
```

Listing 1: Singly Linked List in Python

Summary:

| Operation | Time Complexity |
|-----------|-----------------|
| Search | $O(n)$ |
| Insert | $O(1)$ |
| Delete | $O(1)$ |
| Access | $O(n)$ |

6.3 Operations

6.4 Common Problems

Summary:

| Problem | Description: |
|--|--|
| 206. Reverse Linked List | Given the head of a singly linked list, reverse the list and return the reversed list. |
| <ul style="list-style-type: none"> • Iterative: <ul style="list-style-type: none"> – Init: $\underbrace{\text{None}}_{\text{prev}} \rightarrow \underbrace{1}_{\text{cur}} \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ – While loop: <ul style="list-style-type: none"> * Temp: $\underbrace{\text{None}}_{\text{prev}} \rightarrow \underbrace{1}_{\text{cur}} \rightarrow \underbrace{2}_{\text{temp}} \rightarrow 3 \rightarrow 4 \rightarrow 5$ * Switch link: $\underbrace{\text{None}}_{\text{prev}} \leftarrow \underbrace{1}_{\text{cur}} \rightarrow \underbrace{2}_{\text{temp}} \rightarrow 3 \rightarrow 4 \rightarrow 5$ * Swap positions: $\underbrace{1}_{\text{prev} = \text{cur}} \rightarrow \underbrace{2}_{\text{cur} = \text{temp}} \rightarrow 3 \rightarrow 4 \rightarrow 5$ | |
| 21. Merge Two Sorted Lists | Given two sorted linked lists, merge them into one sorted list. |
| <ul style="list-style-type: none"> • Initialize a dummy head: <ul style="list-style-type: none"> – <code>dummy = ListNode()</code>: Placeholder node to simplify edge case handling. – <code>current = dummy</code>: Build the merged list step-by-step. • Iterate through both lists while neither is empty: <ul style="list-style-type: none"> – Compare current nodes: <ul style="list-style-type: none"> * If <code>list1.val <= list2.val</code>, attach <code>list1</code>'s node to <code>current.next</code>, and move <code>list1</code> forward. Otherwise, do the same for <code>list2</code>. – Move the <code>current</code> pointer forward. • Attach remaining nodes (if any): After the loop, only one of <code>list1</code> or <code>list2</code> may still have nodes left. <ul style="list-style-type: none"> – <code>current.next = list1</code> if <code>list1</code> else <code>list2</code> ensures the remainder is attached. • Return the merged list: return <code>dummy.next</code> returns actual start of the merged list. – Visualization: <ul style="list-style-type: none"> * Initial state: <ul style="list-style-type: none"> · List1: $\underbrace{1}_{\text{list1}} \rightarrow 2 \rightarrow 4$ · List2: $\underbrace{1}_{\text{list2}} \rightarrow 3 \rightarrow 4$ · Merged List: $\underbrace{\text{dummy}}_{\text{start}} \rightarrow$ * Step 1: Compare list1 and list2 ($1 \leq 1$), take list1 <ul style="list-style-type: none"> · <code>dummy</code> → $\underbrace{1}_{\text{current}} \rightarrow$ · list1 → $\underbrace{2}_{\text{list1}} \rightarrow 4$ · list2 unchanged: $\underbrace{1}_{\text{list2}} \rightarrow 3 \rightarrow 4$ * Step 2: Compare 2 and 1 ($2 > 1$), take list2 <ul style="list-style-type: none"> · <code>dummy</code> → 1 → $\underbrace{1}_{\text{current}} \rightarrow$ · list1 remains: $\underbrace{2}_{\text{list1}} \rightarrow 4$ · list2 → $\underbrace{3}_{\text{list2}} \rightarrow 4$ | |

Summary:

| Problem | Description: |
|---|--|
| 141. Linked List Cycle | Given a linked list, determine if it has a cycle in it. |
| <ul style="list-style-type: none"> • Floyd's Cycle Detection Algorithm: <ul style="list-style-type: none"> – Use two pointers (slow and fast) to traverse the list – If they meet, a cycle exists. • While Condition: While fast and fast.next are not None b/c fast moves twice as fast so will reach the end first if no cycle. <ul style="list-style-type: none"> – fast: Ensures fast is not None, so fast.next is safe. – fast.next: Ensures fast.next is not None, so fast.next.next is safe. | |
| 143. Reorder List | Given a linked list, reorder it in a specific pattern. Specifically, the pattern is to rearrange the list s.t. the first element is followed by the last element, and so on. |
| <ul style="list-style-type: none"> • Find middle of list using slow/fast pointers w/ Floyd's algorithm, but have fast start at 2nd node. • Reverse the second half of the list using 206. • Merge the two halves together. | |
| 19. Remove Nth Node From End of List | Given a linked list, remove the Nth node from the end of the list. |
| <ul style="list-style-type: none"> • Idea: Use two pointers (slow and fast) to traverse the list by moving fast pointer N steps ahead. <ul style="list-style-type: none"> – Conditional: If fast is None, then this means $N = \text{length of list}$, therefore, remove head by returning head.next. • Move both pointers until fast reaches the end. <ul style="list-style-type: none"> – Slow pointer will be at the node to be deleted, so store prev node and do appropriate adjustments. | |

7 Images

7.1 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:

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

7.2 Common Problems

Summary:

| Problem | Description |
|------------------------|--|
| 661. Image Smoother | <p>Given an image represented by a 2D array, smooth the image by averaging the pixel values of each pixel and its neighbors.</p> <ul style="list-style-type: none"> • Loop through the cols and rows of the image, then <ul style="list-style-type: none"> – 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 |
| 832. Flipping an Image | <p>Given a binary matrix, flip the image horizontally and invert it.</p> <ul style="list-style-type: none"> • 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). |
| 48. Rotate Image | <p>Given an n x n 2D matrix, rotate the image 90 degrees clockwise.</p> <ul style="list-style-type: none"> • Transpose the matrix (swap rows and columns) if $i < j$, then $\text{matrix}[i][j] \xleftrightarrow{\text{swap}} \text{matrix}[j][i]$. • Reverse each row. |
| **835. Image Overlap | <p>Given two images represented by 2D arrays, find the maximum overlap between the two images.</p> <ul style="list-style-type: none"> • Try all possible translations of img1. • For each translation, calculate the overlap with img2. |

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:**

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

- **In-order (Increasing order):** Visit the left subtree, then the root, and finally the right subtree.
- **Pre-order:** Visit the root first, then the left subtree, and finally the right subtree.
- **Post-order:** Visit the left subtree, then the right subtree, and finally the root.

Algorithm:

```
1 class Node:
2     def __init__(self, key):
3         self.val = key
4         self.left = None
5         self.right = None
6
7 class BST:
8     def __init__(self):
9         self.root = None
10
11     def operations(self, _):
12         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 | $O(n)$ |

Algorithm:

```

1 def search(self, key):
2     current = self.root
3     while current:
4         if key == current.val:
5             return current
6         elif key < current.val:
7             current = current.left
8         else:
9             current = current.right
10    return None

```

Algorithm:

```

1 def insert(self, key):
2     def _insert(node, key):
3         if node is None:
4             return TreeNode(key)
5         if key < node.val:
6             node.left = _insert(node.left, key)
7         elif key > node.val:
8             node.right = _insert(node.right, key)
9         return node
10    self.root = _insert(self.root, key)

```

Algorithm:

```

1 def delete(self, key):
2     def _delete(node, key):
3         if node is None:
4             return None
5         if key < node.val:
6             node.left = _delete(node.left, key)
7         elif key > node.val:
8             node.right = _delete(node.right, key)
9         else:
10            # Node with one child or no child
11            if node.left is None:
12                return node.right
13            elif node.right is None:
14                return node.left
15            # Node with two children
16            temp = self._find_min(node.right)
17            node.val = temp.val
18            node.right = _delete(node.right, temp.val)

```

```
19         return node
20     self.root = _delete(self.root, key)
```

Algorithm:

```
1 def _find_min(self, node):
2     while node.left is not None:
3         node = node.left
4     return node
```

Algorithm:

```
1 def inorder(self):
2     def _inorder(node):
3         if node:
4             _inorder(node.left)
5             print(node.val, end=' ')
6             _inorder(node.right)
7     _inorder(self.root)
```

8.2.1 Common Problems

Summary:

Problem

226. Invert Binary Tree

Description:

Given a binary tree, invert it.

- Base case: If the node is None, return.
- Swap left and right children of the current node.
- Recursively call the function on left and right children.

8.2.2 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:

```
1 class BSTSet:
2     def __init__(self):
3         self.root = None
4
5     def add(self, val):
6         self.root = insert_bst(self.root, val)
7
8     def contains(self, val):
9         return search_bst(self.root, val) is not None
10
11    def remove(self, val):
12        self.root = delete_bst(self.root, val)
13
14 class BSTMap:
15     def __init__(self):
16         self.root = None
17
18     def put(self, key, value):
19         self.root = self._put(self.root, key, value)
20
21     def _put(self, node, key, value):
22         if not node:
23             return TreeNode((key, value))
24         if key < node.val[0]:
25             node.left = self._put(node.left, key, value)
26         elif key > node.val[0]:
27             node.right = self._put(node.right, key, value)
28         else:
29             node.val = (key, value)
30         return node
31
32     def get(self, key):
33         node = self.root
34         while node:
35             if key < node.val[0]:
36                 node = node.left
37             elif key > node.val[0]:
38                 node = node.right
39             else:
40                 return node.val[1]
41         return None
```

8.3 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:

```
1 from collections import deque
2
3 def bfs(start, graph):
4     visited = set()
5     queue = deque([start])
6     visited.add(start)
7
8     while queue:
9         node = queue.popleft() # FIFO (BFS)
10
11         for neighbor in graph[node]:
12             if neighbor not in visited:
13                 visited.add(neighbor)
14                 queue.append(neighbor)
```

8.3.1 Common Problems

Summary:

8.4 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:

```
1 from collections import deque
2
3 def dfs(start, graph):
4     visited = set()
5     stack = deque([start])
6     visited.add(start)
7
8     while stack:
9         node = stack.pop() # LIFO (DFS)
10
11         for neighbor in graph[node]:
12             if neighbor not in visited:
13                 visited.add(neighbor)
14                 stack.append(neighbor)
```

Algorithm:

```
1 def dfs(node, visited):
2     if node in visited:
3         return
4
5     visited.add(node)
6
7     for neighbor in graph[node]:
8         dfs(neighbor, visited)
```

8.4.1 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. <ul style="list-style-type: none">• Use DFS or BFS to explore all connected '1's and mark them as visited.• Increment the island count for each unvisited '1'. |
| 79. Word Search | Given a 2D board and a word, check if the word exists in the grid. <ul style="list-style-type: none">• 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. |