DSA + LeetCode: Additional Session

Patterns, Implementation, Complexity

Session Agenda

Agenda

Warm-up Quiz

Deep Dives

Rapid Pattern Drills

Warm-up

Warm-up quiz

Answer 15 questions.

Reply format: 1 / 2 / 3 / 4

Question 1

You need the maximum sum of any contiguous subarray of length k.
Which approach is best?

1) Two pointers

2) Fixed size sliding window

3) Recursion

4) Greedy

Reply format: 1 / 2 / 3 / 4

Answer 1

You need the maximum sum of any contiguous subarray of length k.
Which approach is best?

1) Two pointers

2) Fixed size sliding window

3) Recursion

4) Greedy

Correct answer: 2

Why: Window size is fixed, so keep one running sum. Slide the window by subtracting the left element and adding the new right element. This checks all windows without recomputing sums.

Reply format: 1 / 2 / 3 / 4

Question 2

You need the shortest contiguous subarray whose sum is *at least* a given target.
All numbers are positive. Which approach is best?

1) Fixed size sliding window

2) Variable size sliding window

3) Sorting plus two pointers

4) DFS

Reply format: 1 / 2 / 3 / 4

Answer 2

You need the shortest contiguous subarray whose sum is at least target.
All numbers are positive. Which approach is best?

1) Fixed size sliding window

2) Variable size sliding window

3) Sorting plus two pointers

4) DFS

Correct answer: 2

Why: Because values are positive, expanding right only increases sum. Once sum reaches target, shrink from the left while it still meets the target to minimize length.

Reply format: 1 / 2 / 3 / 4

Question 3

You need to check whether a sorted array has two numbers that sum to X.
Which approach is best?

1) Two pointers from both ends

2) Sliding window

3) HashSet only

4) Recursion

Reply format: 1 / 2 / 3 / 4

Answer 3

You need to check whether a sorted array has two numbers that sum to X.
Which approach is best?

1) Two pointers from both ends

2) Sliding window

3) HashSet only

4) Recursion

Correct answer: 1

Why: Start left at the smallest and right at the largest. If the sum is too small, move left to increase it. If the sum is too large, move right to decrease it. Sorting makes pointer moves safe.

Reply format: 1 / 2 / 3 / 4

Question 4

In a variable size sliding window, when should you move the left pointer?

1) When the window becomes invalid for the condition

2) Every time you move the right pointer

3) Only when you find a new best answer

4) Only when right reaches the end

Reply format: 1 / 2 / 3 / 4

Answer 4

In a variable size sliding window, when should you move the left pointer?

1) When the window becomes invalid for the condition

2) Every time you move the right pointer

3) Only when you find a new best answer

4) Only when right reaches the end

Correct answer: 1

Why: Move left to restore validity or to shrink while staying valid. This is why shrinking is often inside a while loop, not a single if, because the window may need multiple left moves.

Reply format: 1 / 2 / 3 / 4

Question 5

You need the maximum sum of any contiguous subarray.
Some numbers can be negative. Which idea is most useful?

1) Restart the running sum when it becomes harmful

2) Always expand the window until the end

3) Always sort the array first

4) Always use recursion

Reply format: 1 / 2 / 3 / 4

Answer 5

You need the maximum sum of any contiguous subarray.
Some numbers can be negative. Which idea is most useful?

1) Restart the running sum when it becomes harmful

2) Always expand the window until the end

3) Always sort the array first

4) Always use recursion

Correct answer: 1

Why: If the running sum becomes negative, carrying it forward can only hurt future sums. Starting fresh at the next element gives the best chance to increase the maximum.

Reply format: 1 / 2 / 3 / 4

Question 6

What is an invariant in an algorithm?

1) A rule that stays true after every step

2) A variable that always increases

3) A data structure choice

4) A proof by contradiction

Reply format: 1 / 2 / 3 / 4

Answer 6

What is an invariant in an algorithm?

1) A rule that stays true after every step

2) A variable that always increases

3) A data structure choice

4) A proof by contradiction

Correct answer: 1

Why: An invariant is a condition you maintain throughout the algorithm. It lets you reason about correctness step by step, for example window validity or best progress so far.

Reply format: 1 / 2 / 3 / 4

Question 7

You scan an array left to right and track the farthest position you can reach so far.
This tracking is most closely related to which idea?

1) Greedy invariant

2) Binary search midpoint

3) Stack monotonicity

4) Backtracking branching

Reply format: 1 / 2 / 3 / 4

Answer 7

You scan an array left to right and track the farthest position you can reach so far.
This tracking is most closely related to which idea?

1) Greedy invariant

2) Binary search midpoint

3) Stack monotonicity

4) Backtracking branching

Correct answer: 1

Why: Greedy solutions often maintain a progress measure that stays correct after each step. Farthest reachable is one such measure that updates using local information.

Reply format: 1 / 2 / 3 / 4

Question 8

In dynamic programming, a good definition of dp[i] is usually:

1) The best answer for the prefix ending at i

2) The best answer for the suffix starting at i

3) The best answer after sorting

4) The answer using recursion only

Reply format: 1 / 2 / 3 / 4

Answer 8

In dynamic programming, a good definition of dp[i] is usually:

1) The best answer for the prefix ending at i

2) The best answer for the suffix starting at i

3) The best answer after sorting

4) The answer using recursion only

Correct answer: 1

Why: DP typically builds solutions for prefixes. dp[i] stores the best result up to index i and uses earlier dp states to compute the next state without recomputing subproblems.

Reply format: 1 / 2 / 3 / 4

Question 9

A DP recurrence depends only on dp[i-1] and dp[i-2].
What is a common optimization?

1) Use only two variables instead of an array

2) Use recursion instead of iteration

3) Use sorting before DP

4) Use a HashMap for dp

Reply format: 1 / 2 / 3 / 4

Answer 9

A DP recurrence depends only on dp[i-1] and dp[i-2].
What is a common optimization?

1) Use only two variables instead of an array

2) Use recursion instead of iteration

3) Use sorting before DP

4) Use a HashMap for dp

Correct answer: 1

Why: If only the last two states are needed, store two rolling values instead of the full dp array.
This keeps the recurrence but reduces extra space.

Reply format: 1 / 2 / 3 / 4

Question 10

Why is binary search O(log n)?

1) It scans all elements once

2) It halves the search space each step

3) It uses hashing

4) It uses recursion

Reply format: 1 / 2 / 3 / 4

Answer 10

Why is binary search O(log n)?

1) It scans all elements once

2) It halves the search space each step

3) It uses hashing

4) It uses recursion

Correct answer: 2

Why: Each comparison discards about half the remaining candidates. The number of times you can halve n until it becomes 1 grows like log base 2 of n.

Reply format: 1 / 2 / 3 / 4

Question 11

What is the time complexity of inserting an element at index 0 in an ArrayList?

1) O(1)

2) O(log n)

3) O(n)

4) O(n log n)

Reply format: 1 / 2 / 3 / 4

Answer 11

What is the time complexity of inserting an element at index 0 in an ArrayList?

1) O(1)

2) O(log n)

3) O(n)

4) O(n log n)

Correct answer: 3

Why: ArrayList uses a contiguous array internally. Inserting at the front requires shifting existing elements right to make space, which touches about n elements.

Reply format: 1 / 2 / 3 / 4

Question 12

What is the time complexity of contains(x) in a LinkedList?

1) O(1)

2) O(log n)

3) O(n)

4) O(n log n)

Reply format: 1 / 2 / 3 / 4

Answer 12

What is the time complexity of contains(x) in a LinkedList?

1) O(1)

2) O(log n)

3) O(n)

4) O(n log n)

Correct answer: 3

Why: LinkedList does not support random access. contains must walk node by node from the head until it finds x or reaches the end.

Reply format: 1 / 2 / 3 / 4

Question 13

HashMap get(key) is O(1) on average.
What is the main reason?

1) It stores keys in sorted order

2) Hashing maps the key to a bucket index

3) It always uses binary search

4) It always scans all keys

Reply format: 1 / 2 / 3 / 4

Answer 13

HashMap get(key) is O(1) on average.
What is the main reason?

1) It stores keys in sorted order

2) Hashing maps the key to a bucket index

3) It always uses binary search

4) It always scans all keys

Correct answer: 2

Why: A hash function maps the key to a bucket index, so lookup goes directly to a small bucket. With good distribution and resizing, bucket size stays small on average.

Reply format: 1 / 2 / 3 / 4

Question 14

When can HashMap lookup degrade toward O(n)?

1) When the array is sorted

2) When many keys collide into the same bucket

3) When values are large

4) When keys are integers

Reply format: 1 / 2 / 3 / 4

Answer 14

When can HashMap lookup degrade toward O(n)?

1) When the array is sorted

2) When many keys collide into the same bucket

3) When values are large

4) When keys are integers

Correct answer: 2

Why: If many keys land in the same bucket, lookup must compare against many entries in that bucket.
With heavy collisions, bucket scanning can approach linear time.

Reply format: 1 / 2 / 3 / 4

Question 15

Which is contiguous?

1) Subsequence

2) Subarray

3) Both

4) Neither

Reply format: 1 / 2 / 3 / 4

Answer 15

Which is contiguous?

1) Subsequence

2) Subarray

3) Both

4) Neither

Correct answer: 2

Why: Subarray uses consecutive elements with no gaps. Subsequence can skip elements while keeping order. This difference affects whether sliding window applies.

Reply format: 1 / 2 / 3 / 4

Deep Dives

Problem Intro

Container With Most Water

`https://leetcode.com/problems/container-with-most-water/`

What you are solving

You are given an array of heights. Pick two indices.

Water held is limited by the shorter height and the distance.

Return the maximum water possible.

Pattern focus

Two pointers

Key idea to remember

Area = min(height[left], height[right]) * (right - left)

Brute Force Approach

Container With Most Water

Try every pair of lines and compute the area. Track the maximum.

```
public int maxArea(int[] height) {
    int n = height.length;
    int best = 0;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int h = Math.min(height[i], height[j]);
            int w = j - i;
            best = Math.max(best, h * w);
        }
    }

    return best;
}
```

Time and space

Time: O(n^2) because we check all pairs

Space: O(1)

Optimized Approach

Container With Most Water

Start with two pointers at the ends. Compute area.

Move the pointer at the shorter height inward.

Width decreases each step, so only increasing the limiting height can help.

```
public int maxArea(int[] height) {
    int left = 0;
    int right = height.length - 1;
    int best = 0;

    while (left < right) {
        int h = Math.min(height[left], height[right]);
        int w = right - left;
        best = Math.max(best, h * w);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return best;
}
```

Time and space

Time: O(n) because each pointer moves forward at most n times

Space: O(1)

Problem Intro

Longest Substring Without Repeating Characters

`https://leetcode.com/problems/longest-substring-without-repeating-characters/`

What you are solving

Given a string, return the length of the longest substring

that contains no repeating characters.

Substring means contiguous.

Pattern focus

Variable size sliding window

Key idea to remember

Maintain a window that always has unique characters

Brute Force Approach

Longest Substring Without Repeating Characters

For each start index, expand forward while characters stay unique.

```
public int lengthOfLongestSubstring(String s) {
    int n = s.length();
    int best = 0;

    for (int i = 0; i < n; i++) {
        boolean[] seen = new boolean[128];
        for (int j = i; j < n; j++) {
            char c = s.charAt(j);
            if (seen[c]) break;
            seen[c] = true;
            best = Math.max(best, j - i + 1);
        }
    }

    return best;
}
```

Time and space

Time: O(n^2) in the worst case

Space: O(1) fixed character set

Optimized Approach

Longest Substring Without Repeating Characters

Expand right. If a duplicate appears, shrink left until unique again.

Track the maximum window length after each expansion.

```
import java.util.HashSet;
import java.util.Set;

public int lengthOfLongestSubstring(String s) {
    Set<Character> set = new HashSet<>();
    int left = 0;
    int best = 0;

    for (int right = 0; right < s.length(); right++) {
        char c = s.charAt(right);

        while (set.contains(c)) {
            set.remove(s.charAt(left));
            left++;
        }

        set.add(c);
        best = Math.max(best, right - left + 1);
    }

    return best;
}
```

Time and space

Time: O(n) each character enters and leaves the set at most once

Space: O(min(n, charset))

Problem Intro

Jump Game

`https://leetcode.com/problems/jump-game/`

What you are solving

Each value is your maximum jump length from that index.

Start at index 0. Return true if you can reach the last index.

Pattern focus

Greedy

Key idea to remember

Track the farthest index you can reach so far

Brute Force Approach

Jump Game

Try all reachable next indices from each position. This branches heavily.

```
public boolean canJump(int[] nums) {
    return dfs(nums, 0, new boolean[nums.length]);
}

private boolean dfs(int[] nums, int i, boolean[] visiting) {
    if (i >= nums.length - 1) return true;
    if (visiting[i]) return false;

    visiting[i] = true;
    int furthest = Math.min(nums.length - 1, i + nums[i]);

    for (int next = i + 1; next <= furthest; next++) {
        if (dfs(nums, next, visiting)) return true;
    }

    visiting[i] = false;
    return false;
}
```

Time and space

Time: Can be very large due to branching

Space: O(n) recursion depth

Optimized Approach

Jump Game

Maintain reach, the farthest index reachable so far.

If i ever exceeds reach, you are stuck.

Update reach with max(reach, i + nums[i]).

```
public boolean canJump(int[] nums) {
    int reach = 0;

    for (int i = 0; i < nums.length; i++) {
        if (i > reach) return false;
        reach = Math.max(reach, i + nums[i]);
    }

    return true;
}
```

Time and space

Time: O(n) single pass

Space: O(1)

Problem Intro

House Robber

`https://leetcode.com/problems/house-robber/`

What you are solving

You cannot rob two adjacent houses.

Return the maximum money you can rob.

Pattern focus

Dynamic programming

Key idea to remember

At each house, decide take it or skip it

Brute Force Approach

House Robber

At each index, choose take or skip. This repeats subproblems.

```
public int rob(int[] nums) {
    return helper(nums, 0);
}

private int helper(int[] nums, int i) {
    if (i >= nums.length) return 0;

    int take = nums[i] + helper(nums, i + 2);
    int skip = helper(nums, i + 1);

    return Math.max(take, skip);
}
```

Time and space

Time: Exponential due to repeated branching

Space: O(n) recursion depth

Optimized Approach

House Robber

dp[i] means best money up to index i.

dp[i] = max(dp[i-1], dp[i-2] + nums[i]).

Keep two rolling values for O(1) space.

```
public int rob(int[] nums) {
    int prev2 = 0;
    int prev1 = 0;

    for (int x : nums) {
        int take = prev2 + x;
        int skip = prev1;
        int cur = Math.max(take, skip);

        prev2 = prev1;
        prev1 = cur;
    }

    return prev1;
}
```

Time and space

Time: O(n) single pass

Space: O(1)

Pattern Drills

Problem Intro

Minimum Size Subarray Sum

`https://leetcode.com/problems/minimum-size-subarray-sum/`

What you are solving

Given positive integers and a target, find the minimum length

of a contiguous subarray whose sum is at least target.

If none exists, return 0.

Pattern focus

Variable size sliding window

Fill-in Code

Minimum Size Subarray Sum

```
public int minSubArrayLen(int target, int[] nums) {
    int left = 0;
    int sum = 0;
    int best = Integer.MAX_VALUE;

    for (int right = 0; right < nums.length; right++) {
        sum += nums[right];

        // TODO: shrink while sum >= target
        // Update best using current window length
        // Move left forward and reduce sum
    }

    return best == Integer.MAX_VALUE ? 0 : best;
}
```

Hint: Use while (sum >= target) to shrink and update best

Solution

Minimum Size Subarray Sum

```
public int minSubArrayLen(int target, int[] nums) {
    int left = 0;
    int sum = 0;
    int best = Integer.MAX_VALUE;

    for (int right = 0; right < nums.length; right++) {
        sum += nums[right];

        while (sum >= target) {
            best = Math.min(best, right - left + 1);
            sum -= nums[left];
            left++;
        }
    }

    return best == Integer.MAX_VALUE ? 0 : best;
}
```

Time and space

Time: O(n) each index moves forward at most once

Space: O(1)

Problem Intro

Longest Repeating Character Replacement

`https://leetcode.com/problems/longest-repeating-character-replacement/`

What you are solving

Given a string and k, you can replace at most k characters.

Return the longest substring that can become all one character.

Pattern focus

Variable size sliding window

Fill-in Code

Longest Repeating Character Replacement

```
public int characterReplacement(String s, int k) {
    int[] freq = new int[26];
    int left = 0;
    int maxFreq = 0;
    int best = 0;

    for (int right = 0; right < s.length(); right++) {
        int idx = s.charAt(right) - 'A';
        freq[idx]++;

        // TODO: update maxFreq

        // TODO: shrink while (right - left + 1) - maxFreq > k
        // Shrink by decrementing freq at left and moving left forward

        // TODO: update best
    }

    return best;
}
```

Hint: Window invalid when windowSize - maxFreq > k

Solution

Longest Repeating Character Replacement

```
public int characterReplacement(String s, int k) {
    int[] freq = new int[26];
    int left = 0;
    int maxFreq = 0;
    int best = 0;

    for (int right = 0; right < s.length(); right++) {
        int idx = s.charAt(right) - 'A';
        freq[idx]++;
        maxFreq = Math.max(maxFreq, freq[idx]);

        while ((right - left + 1) - maxFreq > k) {
            freq[s.charAt(left) - 'A']--;
            left++;
        }

        best = Math.max(best, right - left + 1);
    }

    return best;
}
```

Time and space

Time: O(n) each pointer moves forward at most once

Space: O(1)

Problem Intro

Gas Station

`https://leetcode.com/problems/gas-station/`

What you are solving

Each station provides gas[i]. Moving to next costs cost[i].

Find a start index to complete the circle, or return -1.

Pattern focus

Greedy

Fill-in Code

Gas Station

```
public int canCompleteCircuit(int[] gas, int[] cost) {
    int total = 0;
    int tank = 0;
    int start = 0;

    for (int i = 0; i < gas.length; i++) {
        int gain = gas[i] - cost[i];

        // TODO: update total and tank

        // TODO: if tank < 0, reset start to i + 1 and tank to 0
    }

    // TODO: return -1 if total < 0 else return start
    return -1;
}
```

Hint: If a segment fails, none of its indices can be the start

Solution

Gas Station

```
public int canCompleteCircuit(int[] gas, int[] cost) {
    int total = 0;
    int tank = 0;
    int start = 0;

    for (int i = 0; i < gas.length; i++) {
        int gain = gas[i] - cost[i];
        total += gain;
        tank += gain;

        if (tank < 0) {
            start = i + 1;
            tank = 0;
        }
    }

    return total < 0 ? -1 : start;
}
```

Time and space

Time: O(n) single pass

Space: O(1)

That’s a wrap!