
🔷 **WEEK 2 – DAY 8**

Stack, State Tracking, Security Logging

Sony evaluates:

- Whether you can **model state changes**
- Proper use of **LIFO**
- Ability to connect **DSA → real security features**
- Clear explanation before coding

---

1. Stack — Core Concept

Spoken Explanation (What You Say Aloud)

"A stack follows the LIFO principle—Last In, First Out. It is ideal for problems where the most recent operation must be processed first, such as nested structures, rollback operations, or state reversal."

When Sony Expects Stack Usage

- Syntax validation (parentheses, XML, JSON)
- Expression evaluation (infix → postfix)
- Undo / redo operations
- Call stack simulation
- **Monotonic stacks** for performance optimization (next greater element, histogram problems)

Warm-Up Interview Questions

1. Stack vs recursion?
2. Stack vs queue use cases?
3. Can stack be implemented using array?
4. Overflow/underflow handling?
5. Real-world examples?

1️⃣ **Stack vs Recursion?**

Spoken Answer:

"Recursion internally uses a stack. Explicit stacks avoid stack overflow and give better control over memory."

Aspect	Stack	Recursion
Memory	Explicit control	Implicit
Risk	Safe	Stack overflow
Debug	Easier	Harder

---

2 Stack vs Queue Use Cases?

Spoken Answer:

"Stack is LIFO—used for undo, parsing, backtracking. Queue is FIFO—used for scheduling, BFS, buffering."

---

3 Can Stack Be Implemented Using Array?

Spoken Answer:

"Yes. Arrays are commonly used with a pointer to the top. Dynamic resizing avoids overflow."

---

4 Overflow / Underflow Handling?

Spoken Answer:

"Overflow occurs when pushing to a full stack; underflow occurs when popping from an empty stack. We handle this using bounds checks or dynamic arrays."

---

5 Real-World Examples?

Spoken Answer:

"Browser back button, undo/redo in editors, function calls, syntax parsing in compilers."

---

2. Problem 1: Valid Parentheses

Interview Question

Given a string containing ()[]{}, determine if it is valid.

Pattern: Stack (LIFO)
Difficulty: Easy (Sony expects 100% correctness)

Spoken Explanation (What You Say Aloud)

“This problem is best solved using a stack because parentheses must be closed in the reverse order in which they are opened.
Whenever I see an opening bracket, I push it onto the stack.
Whenever I see a closing bracket, I check whether it matches the most recent opening bracket on the stack.
If it doesn’t match or the stack is empty, the string is invalid.
At the end, if the stack is empty, the string is valid.”

Key Insight (Memorize This Line)

“Every opening bracket must be closed in the correct order.”

Python Solution (WITH INPUT)

```
# Program: valid_parentheses_stack.py


def isValid(s):
    stack = []
    mapping = {')': '(', ']': '[', '}': '{'}


    for ch in s:
        if ch in mapping:
            if not stack or stack.pop() != mapping[ch]:
                return False
        else:
            stack.append(ch)


    return not stack




# INPUT
s = "()[]{}"
print(isValid(s))   # Output: True


s2 = "(]"
```

```
print(isValid(s2))  # Output: False
```

JavaScript Solution (WITH INPUT)

```
// Program: validParenthesesStack.js


function isValid(s) {
    const stack = [];
    const map = { ')': '(', ']': '[', '}': '{' };

    for (let ch of s) {
        if (map[ch]) {
            if (!stack.length || stack.pop() !== map[ch]) {
                return false;
            }
        } else {
            stack.push(ch);
        }
    }
    return stack.length === 0;
}


// INPUT
console.log(isValid("()[]{}")); // true
console.log(isValid("(]"));     // false
```

Complexity (Say This Confidently)

- **Time Complexity:** O(n)
- **Space Complexity:** O(n) (stack in worst case)

Sony Follow-Ups

1. Why stack instead of counters?

2. What if input has other characters?
3. How to validate HTML/XML tags?
4. Can recursion replace stack?
5. Streaming input?

---

1 Why stack instead of counters?

Spoken Answer:

"Counters only track quantity, not order. Parentheses validation depends on correct nesting order, which only a stack can enforce."

---

2 What if input has other characters?

Spoken Answer:

"We can simply ignore non-bracket characters or filter input before processing. The core logic remains unchanged."

---

3 How would you validate HTML / XML tags?

Spoken Answer:

"The same stack approach applies. Instead of characters, we push opening tag names and pop when we encounter matching closing tags."

---

4 Can recursion replace a stack?

Spoken Answer:

"Recursion uses the call stack implicitly, but explicit stacks are safer and avoid stack overflow for large inputs."

---

5 How would this work for streaming input?

Spoken Answer:

"We can process characters one by one and maintain the stack incrementally. If a mismatch occurs at any point, we can terminate early."

Sony One-Line Summary (Very Important)

"Stacks are essential when correctness depends on reversing or validating nested order."

---

3. Problem 2: Next Greater Element

Interview Question

Given an array, find the next greater element for each element.

Brute Force

- Nested loops → **O(n²)** ❌

Pattern: Monotonic Stack (Decreasing)

Spoken Explanation (What You Say Aloud)

"The brute-force approach checks every pair and takes quadratic time, which is not optimal.
Instead, I use a **monotonic decreasing stack** that stores indices of elements whose next greater element has not been found yet.
When a new element is greater than the stack's top, it becomes the next greater element for all smaller elements before it.
Each element is pushed and popped at most once, giving linear time complexity."

Key Insight (Say This Clearly)

"A decreasing stack allows us to resolve next greater elements as soon as a larger value appears."

Python Solution (WITH INPUT)

```
# File: next_greater_element_stack.py


def nextGreater(nums):
    stack = []
    result = [-1] * len(nums)


    for i in range(len(nums)):
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            result[idx] = nums[i]
        stack.append(i)
```

```
    return result



# INPUT
nums = [4, 5, 2, 25]
print(nextGreater(nums))  # Output: [5, 25, 25, -1]
```

JavaScript Solution (WITH INPUT)

```
// File: nextGreaterElementStack.js

function nextGreater(nums) {
    const stack = [];
    const res = new Array(nums.length).fill(-1);

    for (let i = 0; i < nums.length; i++) {
        while (stack.length && nums[i] > nums[stack[stack.length - 1]]) {
            const idx = stack.pop();
            res[idx] = nums[i];
        }
        stack.push(i);
    }
    return res;
}

// INPUT
console.log(nextGreater([4, 5, 2, 25])); // [5, 25, 25, -1]
```

Complexity (Say This in Interview)

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

Sony Follow-Ups

1. Why indices instead of values?
2. Circular array variation?
3. Next smaller element?
4. Stock span problem?
5. Real-world use case?

1️⃣ Why use indices instead of values?

Spoken Explanation (Say This Aloud)

"Using indices lets me place the answer directly at the correct position in the result array and correctly handle duplicate values.
If I stored only values, I would lose positional information."

Key Reason

- Handles **duplicates**
- Maps result to **exact index**
- Required for circular / span problems

(No separate code needed — applies to all stack solutions.)

2️⃣ Circular Array Variation

Interview Question

Array is circular. After the last element, we continue from the start.

Spoken Explanation

"To simulate circular behavior, I iterate the array twice using modulo indexing.
I only push indices during the first pass to avoid infinite loops."

Python Code (WITH INPUT)

Program Name: next_greater_circular.py

```
def nextGreaterCircular(nums):
    n = len(nums)
    res = [-1] * n
    stack = []
```

```
    for i in range(2 * n):
        curr = nums[i % n]
        while stack and nums[stack[-1]] < curr:
            idx = stack.pop()
            res[idx] = curr
        if i < n:
            stack.append(i)

    return res

# INPUT
nums = [1, 2, 1]
print(nextGreaterCircular(nums))  # Output: [2, -1, 2]
```

JavaScript Code (WITH INPUT)

Program Name: nextGreaterCircular.js

```
function nextGreaterCircular(nums) {
    const n = nums.length;
    const res = new Array(n).fill(-1);
    const stack = [];

    for (let i = 0; i < 2 * n; i++) {
        const curr = nums[i % n];
        while (stack.length && nums[stack[stack.length - 1]] < curr) {
            const idx = stack.pop();
            res[idx] = curr;
        }
        if (i < n) stack.push(i);
    }
    return res;
```

```
}

// INPUT
console.log(nextGreaterCircular([1, 2, 1])); // [2, -1, 2]
```

3️⃣ Next Smaller Element

Spoken Explanation

“This is the same pattern, but instead of looking for a greater value, I reverse the comparison and use a monotonic increasing stack.”

Python Code (WITH INPUT)

Program Name: next_smaller_element.py

```
def nextSmaller(nums):
    stack = []
    res = [-1] * len(nums)

    for i in range(len(nums)):
        while stack and nums[i] < nums[stack[-1]]:
            idx = stack.pop()
            res[idx] = nums[i]
        stack.append(i)

    return res

# INPUT
nums = [4, 5, 2, 10]
print(nextSmaller(nums))  # Output: [2, 2, -1, -1]
```

JavaScript Code (WITH INPUT)

Program Name: nextSmallerElement.js

```
function nextSmaller(nums) {
```

```
    const stack = [];
    const res = new Array(nums.length).fill(-1);

    for (let i = 0; i < nums.length; i++) {
        while (stack.length && nums[i] < nums[stack[stack.length - 1]]) {
            const idx = stack.pop();
            res[idx] = nums[i];
        }
        stack.push(i);
    }
    return res;
}

// INPUT
console.log(nextSmaller([4, 5, 2, 10])); // [2, 2, -1, -1]
```

4️⃣ Stock Span Problem

Interview Question

For each day, calculate how many consecutive previous days had a price less than or equal to today.

Spoken Explanation

“Stock span is a variation of next greater to the left.
I pop all prices smaller than or equal to the current price and calculate distance using indices.”

Python Code (WITH INPUT)

Program Name: stock_span.py

```
def stockSpan(prices):
    stack = []
    span = [0] * len(prices)

    for i in range(len(prices)):
        while stack and prices[stack[-1]] <= prices[i]:
```

```
            stack.pop()
        span[i] = i + 1 if not stack else i - stack[-1]
        stack.append(i)

    return span

# INPUT
prices = [100, 80, 60, 70, 60, 75, 85]
print(stockSpan(prices))  # Output: [1,1,1,2,1,4,6]
```

JavaScript Code (WITH INPUT)

Program Name: stockSpan.js

```
function stockSpan(prices) {
    const stack = [];
    const span = new Array(prices.length);

    for (let i = 0; i < prices.length; i++) {
        while (stack.length && prices[stack[stack.length - 1]] <= prices[i]) {
            stack.pop();
        }
        span[i] = stack.length === 0 ? i + 1 : i - stack[stack.length - 1];
        stack.push(i);
    }
    return span;
}

// INPUT
console.log(stockSpan([100,80,60,70,60,75,85])); // [1,1,1,2,1,4,6]
```

---

5 Real-World Use Cases (Sony Loves This)

Spoken Explanation

“Monotonic stacks are used in stock market analysis, CPU scheduling, weather forecasting, histogram area calculation, and performance monitoring systems.”

Examples

- Stock price prediction
- CPU load optimization
- Rainwater trapping
- Image histogram processing
- Temperature trend analysis

Sony Final One-Line Summary (Memorize)

“Monotonic stacks reduce future-comparison problems from quadratic to linear time.”

---

4. Sony-Level Stack Variations (High Probability)

Expect **at least one**:

1. Min stack
2. Evaluate postfix expression
3. Daily temperatures
4. Remove adjacent duplicates
5. Largest rectangle in histogram

1️⃣ Min Stack

Concept: Stack + auxiliary state

Spoken Explanation (What You Say Aloud)

“Along with the normal stack, I maintain the minimum value so that getMin() runs in O(1) time without scanning the stack.”

Key Insight

“Each stack node stores the minimum up to that point.”

Program Names

- Python: min_stack.py
- JavaScript: minStack.js

Python (WITH INPUT)

min_stack.py

```
class MinStack:
    def __init__(self):
        self.stack = []

    def push(self, val):
        min_val = val if not self.stack else min(val, self.stack[-1][1])
        self.stack.append((val, min_val))

    def pop(self):
        self.stack.pop()

    def top(self):
        return self.stack[-1][0]

    def getMin(self):
        return self.stack[-1][1]

# INPUT
ms = MinStack()
ms.push(3); ms.push(5); ms.push(2)
print(ms.getMin())  # 2
```

JavaScript (WITH INPUT)

```
// minStack.js
class MinStack {
    constructor() {
        this.stack = [];
    }
    push(val) {
        const minVal = this.stack.length === 0 ? val : Math.min(val, this.stack[this.stack.length - 1][1]);
        this.stack.push([val, minVal]);
    }
```

```
  pop() { this.stack.pop(); }
  top() { return this.stack[this.stack.length - 1][0]; }
  getMin() { return this.stack[this.stack.length - 1][1]; }
}

// INPUT
const ms = new MinStack();
ms.push(3); ms.push(5); ms.push(2);
console.log(ms.getMin()); // 2
```

Sony Follow-Ups

- Why not scan stack? → O(n) vs O(1)
- Space trade-off? → Extra memory for min tracking

---

2 Evaluate Postfix Expression

Concept: Expression evaluation using stack

Spoken Explanation

"Operands are pushed to the stack. On encountering an operator, I pop two operands, evaluate, and push the result back."

Program Names

- Python: evaluate_postfix.py
- JavaScript: evaluatePostfix.js

Python

```
# evaluate_postfix.py
def evalPostfix(tokens):
  stack = []
  for t in tokens:
    if t in "+-*/":
      b, a = stack.pop(), stack.pop()
      stack.append(int(eval(f"{a}{t}{b}")))
    else:
      stack.append(int(t))
```

```
    return stack[0]


# INPUT
print(evalPostfix(["2","1","+","3","*"]))  # 9
```

JavaScript

```
// evaluatePostfix.js
function evalPostfix(tokens) {
    const stack = [];
    for (let t of tokens) {
        if ("+-*/".includes(t)) {
            const b = stack.pop(), a = stack.pop();
            stack.push(Math.trunc(eval(a + t + b)));
        } else {
            stack.push(Number(t));
        }
    }
    return stack[0];
}


// INPUT
console.log(evalPostfix(["2","1","+","3","*"])); // 9
```

Sony Follow-Ups

- Why stack? → Natural operand-operator ordering
- Can handle streaming tokens? → Yes

3️⃣ Daily Temperatures

Concept: Monotonic decreasing stack

Spoken Explanation

"I store indices of unresolved temperatures. When a warmer day appears, I resolve all colder previous days."

Program Names

- Python: daily_temperatures.py
- JavaScript: dailyTemperatures.js

Python

```
# daily_temperatures.py
def dailyTemperatures(t):
    res = [0]*len(t)
    stack = []
    for i in range(len(t)):
        while stack and t[i] > t[stack[-1]]:
            idx = stack.pop()
            res[idx] = i - idx
        stack.append(i)
    return res


# INPUT
print(dailyTemperatures([73,74,75,71,69,72,76,73]))
```

JavaScript

```
// dailyTemperatures.js
function dailyTemperatures(t) {
    const res = new Array(t.length).fill(0);
    const stack = [];
    for (let i = 0; i < t.length; i++) {
        while (stack.length && t[i] > t[stack[stack.length - 1]]) {
            const idx = stack.pop();
            res[idx] = i - idx;
        }
        stack.push(i);
    }
    return res;
}
```

```
// INPUT
console.log(dailyTemperatures([73,74,75,71,69,72,76,73]));
```

Sony Follow-Ups

- Why indices? → Distance calculation
- Time complexity? → O(n)

4️⃣ Remove Adjacent Duplicates

Concept: Stack for state rollback

Spoken Explanation

"I push characters unless the top matches the current one, in which case I pop to remove duplicates."

Program Names

- Python: remove_adjacent_duplicates.py
- JavaScript: removeAdjacentDuplicates.js

Python

```
# remove_adjacent_duplicates.py
def removeDuplicates(s):
    stack = []
    for ch in s:
        if stack and stack[-1] == ch:
            stack.pop()
        else:
            stack.append(ch)
    return "".join(stack)

# INPUT
print(removeDuplicates("abbaca"))  # "ca"
```

JavaScript

```
// removeAdjacentDuplicates.js
function removeDuplicates(s) {
    const stack = [];
    for (let ch of s) {
```

```
    if (stack.length && stack[stack.length - 1] === ch) stack.pop();
    else stack.push(ch);
  }
  return stack.join("");
}

// INPUT
console.log(removeDuplicates("abbaca")); // "ca"
```

Sony Follow-Ups

- Streaming friendly? → Yes
- Space? → O(n)

---

5 Largest Rectangle in Histogram

Concept: Monotonic increasing stack

Spoken Explanation

"Each bar acts as the smallest bar in a rectangle.
When a smaller height appears, I compute areas for previous bars."

Program Names

- Python: largest_rectangle_histogram.py
- JavaScript: largestRectangleHistogram.js

Python

```
# largest_rectangle_histogram.py
def largestRectangle(heights):
  stack = []
  max_area = 0
  heights.append(0)

  for i, h in enumerate(heights):
    while stack and heights[stack[-1]] > h:
      height = heights[stack.pop()]
      width = i if not stack else i - stack[-1] - 1
```

```
            max_area = max(max_area, height * width)
        stack.append(i)
    return max_area

# INPUT
print(largestRectangle([2,1,5,6,2,3]))  # 10
```

JavaScript

```
// largestRectangleHistogram.js
function largestRectangle(heights) {
    const stack = [];
    let maxArea = 0;
    heights.push(0);

    for (let i = 0; i < heights.length; i++) {
        while (stack.length && heights[stack[stack.length - 1]] > heights[i]) {
            const h = heights[stack.pop()];
            const w = stack.length === 0 ? i : i - stack[stack.length - 1] - 1;
            maxArea = Math.max(maxArea, h * w);
        }
        stack.push(i);
    }
    return maxArea;
}

// INPUT
console.log(largestRectangle([2,1,5,6,2,3])); // 10
```

Sony Follow-Ups

- Why stack? → Avoids $O(n^2)$
- Real-world use? → Image processing, skyline analysis

Sony Final Stack Summary (Memorize)

“Whenever problems involve reversal, nesting, rollback, or nearest-greater logic, stack converts brute force into linear time.”

---

5. Project: Add Login Attempt Logging (Security-Focused)

Interview Framing

“Design login attempt logging to support security auditing and brute-force detection.”

Core Requirements

Log **every login attempt**, not just successes.

Fields to Log

- user_id / username
- timestamp
- IP address
- user agent
- status (SUCCESS / FAILURE)
- failure reason (optional)

Database Model Example (Flask / SQLAlchemy)

```
class LoginAttempt(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100))
    ip_address = db.Column(db.String(45))
    user_agent = db.Column(db.Text)
    status = db.Column(db.String(10))
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

Logging Helper

```
def log_login_attempt(username, status, ip, user_agent):
    attempt = LoginAttempt(
        username=username,
        status=status,
```

```
        ip_address=ip,
        user_agent=user_agent
    )
    db.session.add(attempt)
    db.session.commit()
```

Usage in Login API

```
@app.route("/login", methods=["POST"])
def login():
    username = request.json["username"]
    ip = request.remote_addr
    agent = request.headers.get("User-Agent")

    if authenticate(username):
        log_login_attempt(username, "SUCCESS", ip, agent)
        return {"status": "ok"}
    else:
        log_login_attempt(username, "FAILURE", ip, agent)
        return {"status": "failed"}, 401
```

Sony Likes You to Say

“Logging both successful and failed attempts enables anomaly detection and account-lockout policies.”

Interview Follow-Ups on Login Logging

1. How to prevent log flooding?
2. How to detect brute-force attacks?
3. How long to retain login logs?
4. PII considerations?
5. Async vs sync logging?