🧱 Stack Interview Quick-Check Pattern

A visual and tactical guide to solving stack-based problems like a pro in interviews.

---

1. 🧠 When to Use a Stack?

Ask yourself:

- ❓ Do I need to **reverse**, **undo**, or **backtrack**?
- ❓ Do I need to **remember the last seen** item in order?
- ❓ Is the question asking for a **Next Greater/Smaller Element**?
- ❓ Am I evaluating **expressions**, **parentheses**, or **paths**?
- ❓ Do I need **two passes** with memory of the previous pass?

> 💡 Stack is ideal when you need **last-in, first-out (LIFO)** processing.

---

2. 🔍 Common Stack Use Cases

Problem Type	Stack Helps With…
Parentheses Matching	Push on `(`, pop on `)`
Valid Expression Evaluation	Postfix/Infix/Prefix parsing
Next Greater Element	Store indexes, pop until condition met
Histograms / Monotonic Stack	Maintain increasing or decreasing stack
Decode Strings	Push when `[` seen, pop on `]`
Backspace String Compare	Simulate stack for undoing characters
Daily Temperatures	Monotonic stack for warmer day lookup

---

3. 🔩 Must-Know Stack Templates

✅ Valid Parentheses

```
function isValid(s) {
  const stack = [], map = { ')': '(', '}': '{', ']': '[' };
  for (let char of s) {
    if ('({['.includes(char)) {
      stack.push(char);
    } else {
      if (stack.pop() !== map[char]) return false;
    }
  }
  return stack.length === 0;
}
```

✅ Evaluate Reverse Polish Notation

```
function evalRPN(tokens) {
  const stack = [];
  for (let token of tokens) {
    if ('+-*/'.includes(token)) {
      let b = stack.pop(), a = stack.pop();
      stack.push(eval(`${a}${token}${b}`));
    } else {
      stack.push(Number(token));
    }
  }
  return stack[0];
}
```

✅ Decode String ("3[a2[c]]")

```
function decodeString(s) {
  const numStack = [], strStack = [];
  let currStr = "", num = 0;
```

```
  for (let char of s) {
    if (!isNaN(char)) {
      num = num * 10 + Number(char);
    } else if (char === '[') {
      numStack.push(num);
      strStack.push(currStr);
      num = 0;
      currStr = "";
    } else if (char === ']') {
      currStr = strStack.pop() + currStr.repeat(numStack.pop());
    } else {
      currStr += char;
    }
  }

  return currStr;
}
```

✅ Next Greater Element

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

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

  return res;
}
```

✅ Largest Rectangle in Histogram

```
function largestRectangleArea(heights) {
  const stack = [], n = heights.length;
  let max = 0;

  for (let i = 0; i <= n; i++) {
    let h = i === n ? 0 : heights[i];
    while (stack.length && h < heights[stack[stack.length - 1]]) {
      let height = heights[stack.pop()];
      let width = stack.length === 0 ? i : i - stack[stack.length -
1] - 1;
      max = Math.max(max, height * width);
    }
    stack.push(i);
  }

  return max;
}
```

---

4. 🧱 Edge Cases to Watch For

- Empty input or only one item
- Mismatched parentheses or brackets
- Division by zero in expression problems
- Stack underflow (too many pops)
- Infinite loops when not managing index correctly
- Repeated operations in `decodeString` or similar

🧠 Tip: Simulate a stack *by hand* for at least 1 test case to verify logic.

---

5. 🧠 Mental Model for Stack Problems

Scenario	Stack Used To…

Process characters or symbols	Match, validate, group
Track state history	Undo, redo, reverse
Maintain order while comparing values	Monotonic Stack (next greater, etc.)
Encode/decode nested structures	Stack for every layer
Simulate recursive logic manually	Custom stack for function calls

---

🔁 Problem Solving Loop

1. 🧭 Am I checking **last-in, first-out** behavior?
2. ❓ Should I **push/pull based on character/symbol/value**?
3. 📦 Do I need to store **indexes** or **values**?
4. 🔁 Am I comparing current item to the **top of stack**?
5. 🧪 Have I tested open-ended or nested edge cases?

✅ Final Interview Checklist

- Is it truly a **stack** or just a simple array scan?
- Do I need to use a **monotonic increasing/decreasing stack**?
- Am I handling **multi-digit numbers** or **nested structures** properly?
- Any **overflow or underflow** potential?
- Am I cleaning up the stack properly at the end?