ChatGPT

Stack-Based Solutions: In-Depth Explanations for Six NeetCode Problems

1. Valid Parentheses (Stack for Matching Brackets)

Brute Force Thinking: A naïve solution repeatedly removes matching pairs of brackets until no more can be removed. For example, given `s = "({[]})"`, one could iteratively delete `()`, `{}`, or `[]` substrings as they appear. If the string becomes empty at the end, it was valid; otherwise not [1] [2]. This approach works but runs in quadratic time (each removal scans the string) [3].

Optimal Stack Approach: Valid parentheses follow a *last-opened, first-closed* order, which is exactly the LIFO behavior of a stack [4]. We traverse the string and use a stack to track open brackets. When an opening bracket (`(`, `{`, `[`) appears, push it onto the stack. When a closing bracket appears, check the stack: the most recent opening must match the type of this closing bracket [5]. If it matches, pop the stack; if it doesn't match (or the stack is empty when a closing bracket comes), the string is invalid [6] [7]. In the end, if the stack is empty, all brackets were matched in correct order, so the string is valid [8].

Step-by-Step Dry Run: Consider `s = "({[]})"`. We iterate through each character and manipulate the stack: 1. Read `'('` – an opening bracket. **Action:** push it. Stack now: `["("]`. 2. Read `'{'` – opening bracket. **Action:** push. Stack: `["(", "{"]`. 3. Read `'['` – opening bracket. **Action:** push. Stack: `["(", "{", "["]` (top is `"["`). 4. Read `']'` – a closing bracket. **Action:** check stack top, which is `"["`. It matches the type for `']'`, so pop it [9]. Stack after pop: `["(", "{"]`. 5. Read `'}'` – closing bracket. **Action:** check stack top (`"{"`); it matches `'}'`, so pop it. Stack: `["("]`. 6. Read `')'` – closing bracket. **Action:** check stack top (`"("`); it matches `')'`, so pop it. Stack: `[]` (empty).

At the end, the stack is empty and we've successfully matched every opening with the correct closing bracket in order. We return **True**.

JavaScript Implementation:

```
function isValid(s) {
  const stack = [];
  const closeToOpen = { ')': '(', ']': '[', '}': '{' };
  for (let c of s) {
    if (!(c in closeToOpen)) {
      // c is an opening bracket
      stack.push(c);                    // push opens onto stack
    } else {
      // c is a closing bracket
      if (stack.length === 0) return false;         // no matching opening
      if (stack[stack.length - 1] !== closeToOpen[c]) return false;  // type
```

```
mismatch
      stack.pop();                    // pop matched opening
    }
  }
  return stack.length === 0;  // valid if no unmatched openings remain
}
```

Time Complexity: O(n), where n is the length of the string (each character is pushed/popped at most once) [10].
Space Complexity: O(n) in the worst case (all openings pushed onto the stack).

Common Pitfalls: - *Missing Stack Check:* Always check that the stack isn't empty before peeking or popping. If a closing bracket appears when the stack is empty, it's an invalid string. Forgetting this check can cause runtime errors (e.g. popping an empty stack) [11]. - *Type Mismatch:* Ensure you match the correct types of brackets. A common bug is pushing to the stack on a closing bracket rather than popping, or using the wrong mapping. Using a map `closeToOpen` for lookup (closing -> opening) helps avoid mixing up pairs [12]. - *Unclosed Openings:* After processing all characters, any remaining open brackets in the stack mean the string is invalid. For example, `"(()"` would leave an unmatched `'('` on the stack [13]. - *Using the Wrong Data Structure:* Note that a **stack** (LIFO) is required here. A queue (FIFO) would not work because parentheses must close in reverse order of opening [14].

Alternate Approaches: Besides the stack solution, one could use string replacement repeatedly (the brute-force removal method) or even recursion to match pairs, but these are far less efficient and not typical in interviews [3]. The stack method is the standard O(n) solution.

---

2. Min Stack (Stack for Tracking Minimum)

Problem: Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. Standard stack operations are O(1), but naïvely tracking the minimum would be O(n) if we scan the stack on each `getMin()`.

Naïve Idea: On each `getMin()`, one could iterate through all elements to find the smallest. For example, to implement `getMin()` you might pop all items to find the min and then push them back [15]. This indeed yields the correct minimum but takes O(n) time per call [16], which is too slow if `getMin` is called frequently.

Optimal Stack Approach – Two Stacks: We can do better by storing extra information as we go. The key is to maintain, alongside the main stack, a *min-stack* that always holds the current minimum at the corresponding depth [17]. Each time we push a value, we also push an entry on the min-stack representing the minimum value **up to that point** [18]. This way, the top of the min-stack is always the minimum of all values in the main stack. When we pop, we pop from both stacks, so the min-stack always stays in sync [19].

- **Push operation:** Push the value onto the main stack. Then push the new minimum onto the min-stack. The new minimum is `min(val, currentMin)` – i.e. either the new value or the previous minimum if that is smaller [20].

- **Pop operation:** Pop from both the main stack *and* the min-stack (ensuring the min-stack discards the min value for the popped level) [21] .
- **Top operation:** Peek the top of the main stack.
- **GetMin operation:** Peek the top of the min-stack, which is the minimum element in the stack [22] .

Example Dry Run: Push a sequence of values and see how the two stacks evolve:

- Start with an empty stack. `minStack` is empty as well.
- `push(5)` : Main stack becomes `[5]`. min-stack becomes `[5]` (the min so far is 5).
- `push(2)` : Main stack `[5, 2]`. min-stack `[5, 2]` (min so far becomes 2, since `min(2,5)=2`).
- `push(8)` : Main stack `[5, 2, 8]`. min-stack `[5, 2, 2]` (the new value 8 is not smaller than current min 2, so we push 2 again as the min so far remains 2).
- `push(1)` : Main stack `[5, 2, 8, 1]`. min-stack `[5, 2, 2, 1]` (new min is 1, since `min(1,2)=1`).
- Now `getMin()` → look at top of min-stack, which is `1` . Indeed 1 is the smallest of {5,2,8,1}.
- `pop()` → removes top of main stack (which was 1) and also pops top of min-stack (which was 1). Now the current minimum (new top of min-stack) is `2` . Thus after popping 1, `getMin()` would correctly return 2.
- `top()` → returns top of main stack (which is now 8).

Through these operations, each push/pop updates the min-stack appropriately, so `getMin()` is always O(1).

JavaScript Implementation:

```
class MinStack {
  constructor() {
    this.stack = [];
    this.minStack = [];
  }
  push(val) {
    this.stack.push(val);
    // Compute new min: if minStack is empty, val is the min. Otherwise compare
val with current min (top of minStack).
    const newMin = this.minStack.length === 0
                   ? val
                   : Math.min(val, this.minStack[this.minStack.length - 1]);
    this.minStack.push(newMin);
  }
  pop() {
    this.stack.pop();
    this.minStack.pop();
  }
  top() {
    return this.stack[this.stack.length - 1];
  }
  getMin() {
```

```
    return this.minStack[this.minStack.length - 1];
  }
}
```

Time Complexity: All operations run in O(1) time on average [23] (push, pop, top, and getMin each do a constant amount of work).
Space Complexity: O(n) for storing up to n values in the main stack and n values in the min-stack [23]. (We use two stacks of the same length, so it's 2n space, i.e. O(n).)

Common Pitfalls: - *Forgetting to Synchronize Stacks:* If you push to one stack and not the other (or pop from one and not the other), the two stacks get out of sync. For example, some implementations try to save space by pushing to `minStack` only when the new value is <= current min. This can work, but you must then pop from `minStack` **only** when the popped value equals the current min [24]. It's safer in interviews to push a min value for every push (as above) to avoid mistakes. - *Handling Edge Cases:* Be careful with initial conditions. When the stack is empty, be sure to initialize the min value correctly (push the value itself as the min for the first element). Also, ensure `pop()` and `top()` are only called when the stack isn't empty (per problem constraints). - *One-Stack Trickiness:* An alternative one-stack approach encodes the current minimum within the stack by storing *differences* or sentinel values (for instance, push `(val - currentMin)` and update the currentMin when a new minimum is pushed) [25]. While clever (and space-saving), this method is tricky to implement correctly and handle on pops, especially with potential integer overflow issues [26]. It's generally beyond the expectations of a typical interview unless you bring it up; sticking to the two-stack method is recommended for clarity.

---

3. Evaluate Reverse Polish Notation (Stack for Expression Evaluation)

Problem: We are given an array of tokens representing a mathematical expression in **Reverse Polish Notation** (RPN, or postfix notation). For example: `["2","1","+","3","*"]` represents the expression $(2 + 1) * 3 = 9$. The task is to evaluate the expression and return the result. In RPN, operators come *after* their operands, so the expression can be evaluated using a stack.

Naïve Idea: One brute-force approach is to scan the tokens repeatedly and perform operations as soon as an operator with two preceding operands is found. For instance, you could walk through the list, find the first operator, compute the result of its two operands, replace the three tokens (two operands + operator) with the single result, and repeat until one number remains [27] [28]. This approach will work but is quite inefficient (O(n^2) in the worst case) since each operation may require shifting tokens in the array.

Stack Approach: A stack is an ideal structure for evaluating postfix expressions [29]. The algorithm is straightforward: 1. Create an empty stack. 2. Iterate through each token in the input array: - If the token is a number (operand), push its integer value onto the stack [30]. - If the token is an operator (`+`, `-`, `*`, or `/`): - Pop the **top two** numbers from the stack. Let's call the first one popped `a` and the second one popped `b` – note that since we popped from stack, `a` was the **last** number and `b` was the one before it. Now we perform the operation `b (op) a` [31]. - Push the result of that operation back onto the stack. 3.

After processing all tokens, the stack will have a single number, which is the result of the expression. Pop and return it.

Using the stack ensures that whenever we encounter an operator, the top of the stack holds its two corresponding operands (the most recent numbers that haven't been used yet), which is exactly what we need in postfix evaluation [29] .

Step-by-Step Dry Run: Consider the token array: `["2","1","+","3","*"]`. We walk through the tokens and use a stack to evaluate: - Start with an empty stack `[]`. - Token `"2"` – a number. **Action:** push 2. Stack becomes `[2]`. - Token `"1"` – a number. **Action:** push 1. Stack: `[2, 1]`. - Token `"+"` – an operator. **Action:** pop twice to get operands. We pop 1 (`a=1`), then pop 2 (`b=2`). Now compute `b + a = 2 + 1 = 3`. Push the result `3` onto stack. Stack now: `[3]` [30] . - Token `"3"` – a number. **Action:** push 3. Stack: `[3, 3]`. - Token `"*"` – an operator. **Action:** pop twice for operands. Pop 3 (`a=3`), then pop 3 (`b=3`). Compute `b * a = 3 * 3 = 9`. Push result `9`. Stack: `[9]`. - End of tokens. The stack has `[9]`, so output **9**, which matches the expected result.

Another example: `["4","13","5","/","+"]` represents $4 + (13/5) = 6$. The stack process would compute `13/5 = 2` (integer division truncating toward zero) and then `4 + 2 = 6`.

JavaScript Implementation:

```
function evalRPN(tokens) {
  const stack = [];
  for (const token of tokens) {
    if (token === '+' || token === '-' || token === '*' || token === '/') {
      // It's an operator: pop two operands
      const a = stack.pop();      // right operand
      const b = stack.pop();      // left operand
      let result;
      if (token === '+') {
        result = b + a;
      } else if (token === '-') {
        result = b - a;
      } else if (token === '*') {
        result = b * a;
      } else if (token === '/') {
        // Note: truncate toward 0
        result = b / a;
        result = result < 0 ? Math.ceil(result) : Math.floor(result);
        // Alternatively: result = Math.trunc(b / a);
      }
      stack.push(result);
    } else {
      // It's a number: push onto stack
      stack.push(Number(token));
    }
```

```
    }
    return stack.pop();
}
```

Important Implementation Details: In the code above, notice that for subtraction and division we popped `a` then `b` and computed `b - a` or `b / a`. This is critical – it ensures the left operand is the one that was lower on the stack (earlier in the token stream). Mixing up the order (doing `a - b` or `a / b`) would yield the wrong result [32]. We also ensure division truncates toward zero. In JavaScript, using `Math.trunc(b/a)` achieves this (we manually did it with `Math.floor/Math.ceil` depending on sign). This matches the problem's requirement (e.g., in C++/Java, `-7/2 = -3` when truncated toward zero, not -4) [33].

Time Complexity: O(n) for n tokens, since we process each token exactly once and stack operations are O(1) each [34].
Space Complexity: O(n) for the stack in the worst case (if the expression is something like all numbers then an operator at the end, the stack grows to size n/2) [34].

Common Pitfalls: - *Operand Order:* As noted, subtraction and division are not commutative. If you pop in the wrong order, you'll get `a - b` instead of `b - a`. Always remember: the second value popped was the left operand in the original expression, since it was pushed earlier [32]. - *Integer Division:* In many languages, dividing integers can truncate differently. In Python, for example, `-7 // 2 = -4` (floor division), but the expected RPN result is `-3`. Make sure to truncate toward zero, not toward $-\infty$. In our JavaScript solution, we handled this by using `Math.trunc` (or its equivalent) for the division case [33]. - *Negative Number Tokens:* Tokens can include negative numbers like `"-3"`, which are not operators. Be careful in checking tokens – e.g., checking `if (token[0] == '-')` alone is wrong because it would treat `"-3"` as an operator. We explicitly check if the token is one of the four operator symbols exactly (in code we used `===` comparisons). This avoids misinterpreting negative values [35]. - *Stack Underflow:* Though the problem guarantees a valid RPN expression, it's good practice to ensure that when an operator appears, the stack has at least two operands to pop. A well-formed RPN will always satisfy this, but an extra check can save you from runtime errors in case of bad input.

Alternate Approaches: It's possible to evaluate RPN using recursion (treating the expression as a binary tree where each operator has two subtree values to evaluate) or by first converting to an infix expression, but those approaches are more complex for this task. The stack method is by far the simplest and most efficient way to evaluate postfix expressions [29].

---

4. Daily Temperatures (Monotonic Stack for Next Greater Element)

Problem: We have an array of daily temperatures (integers). For each day, we need to determine how many days one would have to wait until a warmer temperature occurs. If there is no future day with a higher temperature, we put 0 for that day. For example, given `temperatures = [73, 74, 75, 71, 69, 72, 76, 73]`, the answer should be `[1, 1, 4, 2, 1, 1, 0, 0]` [36].

Brute Force: A straightforward approach is, for each day *i*, scan forward to find the next day *j* where temperature[j] > temperature[i]. Then answer[i] = j - i. If no such j, answer[i] = 0. This requires nested loops:

for day 0, you might scan all the way to the end, for day 1 a bit less, etc. In the worst case (e.g., a decreasing sequence), this is $O(n^2)$ time [37] [38], which is too slow for large inputs.

Optimal Stack Approach – Monotonic Decreasing Stack: This problem can be solved in one pass using a *monotonic stack* (a stack that maintains a certain order of elements). Here, we want to efficiently find the next greater (warmer) temperature for each day [39]. We can maintain a stack of **indices** of days, such that the temperatures of those days are in *decreasing* order in the stack.

- We iterate through the days. For each day `i` with temperature `t`:
- While the stack is not empty **and** `t` is higher than the temperature at the top index of the stack, it means day `i` is the next warmer day for the day at the stack's top. We pop the top index `j` from the stack and set `answer[j] = i - j` (the difference in days) [40] [41].
- After popping all smaller (cooler or equal) temperatures from the stack, we push the current day `i` onto the stack.
- If the stack is not empty at the end, those indices have no warmer day ahead, and they remain 0 in the answer (we typically initialize the answer array with 0s, so this is handled).

This works because each day index is pushed exactly once and popped at most once. The stack will always contain indices of days in strictly decreasing temperature order – as soon as a warmer day comes, it resolves (pops) all previous days that are cooler and waiting for a warmer day [39] [42]. This guarantees an efficient solution (linear time).

Example Dry Run: Suppose `temperatures = [70, 71, 68, 72]` for a simpler illustration: - Start with empty stack, `answer = [0, 0, 0, 0]`. - Day 0: temp = 70. Stack is empty, so no previous day to compare. **Push index 0**. Stack = `[0]` (meaning day0 is waiting for a warmer day). - Day 1: temp = 71. Compare with day at top of stack (day0: temp 70). 71 > 70, so **pop 0** from stack and set `answer[0] = 1 - 0 = 1` (day1 is 1 day after day0 and is warmer) [42]. Stack is now empty. Push day1. Stack = `[1]`. - Day 2: temp = 68. Compare with day1 on stack (temp 71). 68 is not warmer than 71, so we cannot resolve day1 yet. **Push index 2**. Stack = `[1, 2]` (day1 and day2 are waiting; note temp[2]=68 is cooler than temp[1]=71, so stack order is still "decreasing" from 71 to 68). - Day 3: temp = 72. Compare with top of stack (day2: temp 68). 72 > 68, so pop day2 and set `answer[2] = 3 - 2 = 1`. Stack now = `[1]`. Now compare 72 with new top (day1: temp 71). 72 > 71, so pop day1 and set `answer[1] = 3 - 1 = 2`. Stack now empty. **Push day3**. Stack = `[3]`. - End of array. Stack has `[3]`, meaning day3 has no warmer future day, so `answer[3]` remains 0 (which it was initialized to). - Final `answer = [1, 2, 1, 0]`. We can verify: day0 waits 1 day (to day1, 71°F) which is correct; day1 waits 2 days (to day3, 72°F); day2 waits 1 day (to day3); day3 has no warmer day.

JavaScript Implementation:

```
function dailyTemperatures(temperatures) {
  const n = temperatures.length;
  const answer = new Array(n).fill(0);
  const stack = []; // will store indices of days, stack maintained in
decreasing temp order
  for (let i = 0; i < n; i++) {
    // While current day is warmer than the day at top of stack, pop and set
```

```
answer
    while (stack.length > 0 && temperatures[i] >
temperatures[stack[stack.length - 1]]) {
      const prevDay = stack.pop();
      answer[prevDay] = i - prevDay;    // found a warmer day for prevDay
    }
    stack.push(i);  // push current day index onto stack
  }
  return answer;
}
```

Time Complexity: O(n) – each index is pushed once and popped at most once, so the while-loop across the entire run will iterate at most n times in total [43] .
Space Complexity: O(n) for the stack in the worst case (if temperatures are in decreasing order, the stack grows to hold all indices).

Common Pitfalls: - *Using* `>=` *instead of* `>`*:* We only pop while the current temperature is strictly greater than the temperature at stack top [44] . If you use `>=`, you would pop on equal temperatures as well, which is incorrect – equal temperature is not "warmer", and that day should *not* resolve the previous day's wait. Using `>=` could cause you to skip a valid future warmer day or break the monotonic property incorrectly. - *Mixing up indices and values:* It's important that we push **indices** onto the stack, not the temperature values. This way, when we find a warmer temperature at day `i`, we know exactly which day index to resolve by popping. A common mistake is to push the temperature itself, then later not know which index to calculate the difference with. If you do store values, you lose track of positions needed for the answer [45] . In our code, we stored indices (and used them to compute day differences). - *Not initializing the answer array:* Make sure to initialize the result array with 0s. Days that never see a warmer future day should remain 0 by default (as our code does by filling with 0s initially). - *Off-by-one errors:* When computing the difference `i - prevDay`, ensure it's correct (in our logic `i` is the index of the next warmer day and `prevDay` is the index of the current day being resolved, so `i - prevDay` is the number of days in between). Off-by-one mistakes can happen if you, say, use `i - prevDay - 1` incorrectly.

Alternate Approach – Backward Traversal/Dynamic Programming: Another clever solution is to traverse from rightmost day backward, using previously computed results to skip ahead. For each day i (going right-to-left), you can jump to the next day j = i+1, and while that day isn't warmer, use `answer[j]` to hop directly to the next candidate day (`j += answer[j]`). This effectively skips segments of days in one go [46] . This approach also achieves O(n) time in the average case and uses O(1) extra space (besides the output array) [47] . However, it's a bit harder to derive and prone to its own pitfalls, so the monotonic stack method is usually easier to explain and implement under interview pressure.

---

5. Car Fleet (Monotonic Stack for Comparing Arrival Times)

Problem: We have `n` cars, each with a starting position and speed, all moving toward a target position on a one-lane road. A *car fleet* is a group of cars that eventually cluster together and travel at the same speed to the target (because a faster car catches up to a slower car ahead, and then it has to slow down). We need

to find out how many car fleets will arrive at the destination. In other words, if you send all cars toward the target, how many distinct groups will form? A single car alone is also considered a fleet.

Key Observation: If we calculate each car's **time** to reach the target (time = (target - position) / speed), we can determine fleet formation by comparing these times [48]. A car with a greater or equal arrival time than a car ahead of it (closer to target) will catch up **before** the target and form a fleet with that car [49] [50]. However, a car that would arrive faster (sooner) than the car ahead cannot pass it (one-lane road), so it will instead catch up and slow down, effectively adopting the slower car's arrival time.

Sorting: It's logical to sort cars by starting position in descending order (from closest to target to farthest) [51] [52]. That way, when we iterate, we are considering the front-most car first and then each car behind it in order. Any car can only catch up to cars ahead of it (with greater position), not behind.

After sorting, we iterate through the sorted list and use a stack to keep track of fleets: - Compute each car's arrival time = `(target - position) / speed`. - Use a stack to store the **arrival times of fleets** formed so far (the top of the stack will be the arrival time of the fleet that is ahead of the current car in iteration) [53]. - For each car in sorted order: - Push its arrival time onto the stack, assuming it forms a new fleet. - Then, check the stack: if the arrival time of this new car is **less than or equal to** the fleet time just ahead (the next car closer to target), it means this car catches up to that fleet before reaching target. **It should not form a new fleet**, so we pop the smaller/equal time off and effectively merge this car into the fleet ahead [54]. In other words, the fleet arrival time remains the larger time (slower fleet). - If the new car's time is greater (meaning it can't catch up to the fleet ahead), then it stays on the stack as a new fleet. - In the end, the number of fleet times on the stack is the number of fleets.

Example: Suppose `target = 10` and we have positions `[4, 1, 0, 7]` with speeds `[2, 2, 1, 1]` (as in one of the examples). First, pair them and sort by position descending: we get cars in order of starting position 7, 4, 1, 0. Their arrival times: - Car at 7 (speed 1): time = $(10-7)/1 = 3$ hours. - Car at 4 (speed 2): time = $(10-4)/2 = 3$ hours. - Car at 1 (speed 2): time = $(10-1)/2 = 4.5$ hours. - Car at 0 (speed 1): time = $(10-0)/1 = 10$ hours.

Now iterate in this order: - Car at 7: stack = [3]. (It starts a new fleet.) - Car at 4: time = 3. The top of stack is 3 (fleet ahead has time 3). Our car's time (3) $\leq$ 3, meaning it catches up exactly at the target (or earlier if it were smaller) – it joins the fleet with time 3 instead of creating a new one [55]. We pop the new time off immediately (or simply don't count it as new fleet). Stack remains [3]. - Car at 1: time = 4.5. Top of stack is 3, and 4.5 > 3, so this car cannot catch up to the fleet ahead (which arrives sooner). It forms a new fleet. Push 4.5. Stack = [3, 4.5]. - Car at 0: time = 10. Top is 4.5, and 10 > 4.5, so this car also cannot catch up to the fleet ahead. It becomes a new fleet. Push 10. Stack = [3, 4.5, 10].

Stack size = 3, so there are 3 car fleets. Indeed, in this scenario: the car at 7 and 4 form one fleet (arrive together in 3h), the car at 1 is alone (arrives at 4.5h), and the car at 0 is alone (arrives much later at 10h).

JavaScript Implementation:

```
function carFleet(target, position, speed) {
  // Pair up position and speed for sorting
  const cars = position.map((pos, i) => [pos, speed[i]]);
```

```
  cars.sort((a, b) => b[0] - a[0]);  // sort descending by position

  const stack = [];
  for (const [pos, sp] of cars) {
    const time = (target - pos) / sp;
    stack.push(time);                    // assume new fleet
    if (stack.length >= 2 && stack[stack.length - 1] <= stack[stack.length -
2]) {
      // Current car catches up to the fleet ahead
      stack.pop();  // merge with fleet ahead, discard current time
    }
  }
  return stack.length;
}
```

In this implementation, we push each car's time, then immediately check if it should merge with the fleet in front. If `time <= stack[stack.length-2]`, it means this car's fleet merges into the one ahead (we pop it back out) [56] [57]. The stack thereby only retains the largest time in each fleet (which is the fleet's overall arrival time).

Time Complexity: Sorting takes O(n log n). The subsequent single pass is O(n). Overall O(n log n) for n cars [58].
Space Complexity: O(n) for the array of car pairs and the stack to hold fleet times.

Common Pitfalls: - *Sorting Order:* Make sure to sort by position **descending** (from target backward) [59]. If you sort the other way (ascending), you'll process cars from the back, and it becomes much harder to determine fleet formation (you'd end up counting fleets incorrectly because you'd encounter slower cars after faster ones and logic breaks). - *Using* `<` *vs* `<=`: Use `<=` when comparing times to decide fleet merging [55]. If a car's arrival time is exactly equal to the fleet ahead, they will meet exactly at the target and still count as one fleet (the problem considers catching up "at the moment of arrival" as joining the fleet). Using `<` (strictly less) would mistakenly count such cases as separate fleets. In our code, we used `<=` to merge in both equal and smaller time scenarios. - *Precision/Division Issues:* In languages with integer division, be careful to use floating point for time calculation. For example in Java, `(target - pos) / speed` using integers would truncate the time. Two cars with times 3.5 and 3.1 could both become 3 due to truncation and incorrectly merge. In Java or C++, cast to double before division [60]. In JavaScript, this isn't an issue since division of numbers yields a float by default. Just be mindful of using proper precision so that comparisons are accurate. - *Interpreting the Output:* Remember that the stack (or the count we end up with) represents the number of fleets, not necessarily the number of cars left – multiple cars can collapse into one fleet. It helps to walk through a small example to verify your logic isn't counting individual cars instead of fleets.

Alternate Approach: Instead of an explicit stack, one can simply keep a counter of fleets and track the "slowest" (largest) time seen so far as you iterate from the car closest to target to farthest. Essentially, you iterate through sorted cars and if a car's time is greater than the `maxTimeSeen`, you increment fleet count and update `maxTimeSeen` to that time. If it's <= `maxTimeSeen`, it's part of an existing fleet and you ignore it [61] [62]. This is a slightly different formulation but yields the same result. It might even be simpler

to implement. However, conceptually both approaches are doing the same comparison of arrival times in descending order.

---

6. Largest Rectangle in Histogram (Stack for Span of Next Smaller Elements)

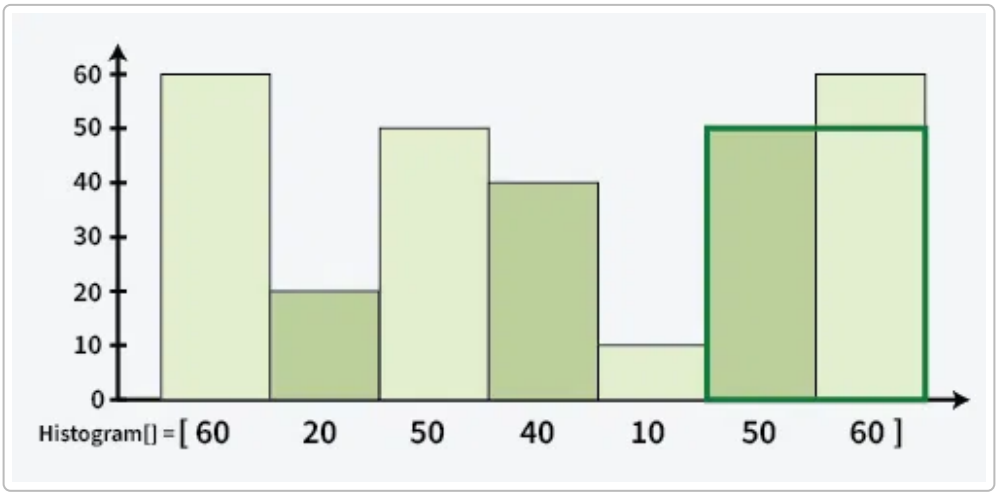

Figure: Visualizing the largest rectangle in a histogram. In this example, the largest rectangle (shaded in green) spans the bars of heights 7, 5, and 9 with a common height of 5, giving an area of 5 × 3 = 15 [63] [64] .

Problem: We are given an array of bar heights representing a histogram. We want to find the area of the largest rectangle that can be formed using contiguous bars. For example, in a histogram with heights `[3, 5, 1, 7, 5, 9]` (as illustrated above), the largest rectangular area is 15 (using the bars of heights 7, 5, and 9, with the limiting height being 5) [63] [64] .

Brute Force: A brute-force solution tries every possible rectangle. One way is: for each bar i, treat it as the potential lowest bar of a rectangle, then expand to the left and right while bars are >= height[i] [65] [66] . This finds the widest rectangle that has height = `heights[i]`. Do this for all i and keep track of the max area. While conceptually straightforward, this is O(n^2) in the worst case (e.g., heights are non-decreasing, so for each bar you extend through all bars) [67] .

Optimal Stack Approach – Monotonic Increasing Stack: The trick is to find the *next smaller element* on the left and right of each bar efficiently [68] . If you know the index of the first smaller bar to the left of i, and the first smaller bar to the right of i, then you know that bar i can extend between those two "bounds." Within those bounds, bar i is the shortest, so it determines the rectangle height. Thus area with bar i as the smallest bar = `heights[i] * (rightBound - leftBound - 1)`, where `leftBound` and `rightBound` are the indices of the next smaller bar to the left and right (or -1/`n` if none) [69] [70] .

A *monotonic increasing stack* can determine these bounds in one pass: - We keep a stack of indices of bars, maintaining the invariant that their heights are in increasing order on the stack (the top of the stack is the index of the tallest bar seen so far without a smaller break) [71] . - We iterate through the bars: - While the current bar's height is less than or equal to the height at the top index of the stack, it means the stack's top

bar has now encountered a bar that is shorter (or equal, which also effectively acts as a boundary) [72] [73]. So the bar at the top of the stack can't extend further right than the current index. We **pop** it and calculate the area of the rectangle with that bar as the smallest height. - When popping index `j`, after popping, the new top of stack (say index `k`) is the index of the next smaller bar to the left of `j`. The current index `i` is the first smaller bar to the right of `j`. Thus, the width of the rectangle using bar `j` is `i - k - 1` (if the stack is empty after popping, it means no smaller to the left, so width is `i` itself) [74] [75]. - Compute area = `heights[j] * width` and update max area. - Push the current index onto the stack. - To handle any bars that extend to the end, we can append a bar of height 0 at the end of the array (or equivalently, run the loop to `i = n` with `height=0`) to flush out remaining bars on the stack [76] [77]. These remaining bars will pop out, treating the end of the array as the first smaller element to their right.

This algorithm ensures each bar is pushed and popped at most once, giving linear time complexity.

Example Dry Concept: Imagine heights = `[2, 1, 5, 6, 2, 3]` (a classic example). The largest rectangle here is 10 (from the bars of height 5 and 6). Using the stack method: - We traverse, maintaining an increasing stack of indices. - When we reach the height 1 at index 1, it is smaller than height 2 (index 0 on stack), so we pop index 0 and calculate area = 2 * 1 (width 1, since no left smaller, right boundary at index1) = 2. - We continue and push index 1. Then push indices 2 and 3 (heights 5 and 6, stack now [1,2,3] with heights [1,5,6]). - At index 4, height 2 is smaller than height 6 (stack top, index 3), so pop index 3: area = height 6 * width (right boundary 4, left boundary index2) = 6 * (4-2-1) = 6 * 1 = 6. Now top of stack is index 2 (height 5), and 2 < 5, pop index 2: area = height 5 * width (right boundary 4, left boundary index1) = 5 * (4-1-1) = 5 * 2 = 10. Stack now [1], top is height 1 which is <= 2, so stop. Push index 4. - Continue to end, then pop remaining bars computing areas... (the max area found was 10). - The algorithm would find max = 10.

You don't need to follow all those details in an interview if you explain the general process with a smaller example. The core idea is each pop gives the area of the largest rectangle for the bar that was popped (with that bar as the limiting height) [68] [69].

JavaScript Implementation:

```
function largestRectangleArea(heights) {
  const n = heights.length;
  let maxArea = 0;
  const stack = [];  // stack will store indices of bars

  for (let i = 0; i <= n; i++) {
    // Treat index = n as a bar of height 0 (sentinel to pop all remaining)
    const h = (i === n ? 0 : heights[i]);
    // Ensure stack is increasing (strictly). Pop all taller or equal bars.
    while (stack.length > 0 && h <= heights[stack[stack.length - 1]]) {
      const height = heights[stack.pop()];       // height of bar to evaluate
      const leftIndex = stack.length > 0 ? stack[stack.length - 1] : -1;
      const width = i - leftIndex - 1;           // right boundary is i, left
boundary is leftIndex
      maxArea = Math.max(maxArea, height * width);
    }
```

```
    stack.push(i);
  }

  return maxArea;
}
```

In the code above, we append an iteration for `i == n` with `h = 0` to trigger popping of all remaining bars [77]. When popping: - `height` is the bar height. - After popping, `leftIndex` is the new top (the index of the previous smaller bar to the left, or -1 if none) [75]. - `i` is the index of the next smaller bar to the right. Thus the width = `i - leftIndex - 1` [74]. We update `maxArea` each time. By the end, `maxArea` is the largest area found.

Time Complexity: O(n). Each bar index is pushed once and popped at most once, so the operations are linear in total [78].
Space Complexity: O(n) for the stack of indices (in worst case, stack holds all indices if heights array is increasing).

Common Pitfalls: - *Off-by-One in Width:* When calculating the width of the rectangle after popping, remember that if the stack is empty it means the popped bar had no smaller element to the left, so it extends all the way to index 0. In that case width = `i - 0` (which we achieved by using -1 as a sentinel left index) [79]. If the stack is not empty, the current top is the index of a smaller bar on the left, so the rectangle extends from that index + 1 to `i - 1`, giving width = `i - stackTop - 1`. A common bug is forgetting the "-1" or misplacing it, leading to off-by-one errors in width [79]. - *Forgetting Final Cleanup:* It's crucial to pop all remaining bars at the end. If you don't handle the end (like the sentinel 0 height trick), some bars might never be popped and evaluated, and you'd miss rectangles that extend to the end of the histogram [80]. The dummy height at the end is a neat way to ensure everything comes off the stack. - *Strict vs Non-Strict Monotonicity:* In our solution, we used `<=` when comparing heights to decide popping. This ensures that if two bars have equal height, the later one will pop the earlier one as well (treating the earlier one's rectangle as ended at the later index). You could also handle equal heights by leaving them on stack, but you must be consistent in how you define the "next smaller" element (some implementations prefer `<` to maintain strictly increasing heights on stack). Just be careful: using `>` vs `>=` changes how equal heights are handled [81]. Our use of `<=` ensures that two equal-height bars will not both stay on stack (which could lead to incorrect width calculation for the left one). - *Empty Stack Handling:* As mentioned, when the stack is empty, use that to compute width properly (meaning the bar extends to the extreme left) rather than causing an error by accessing stack top. In code, we set `leftIndex = -1` when stack is empty, which nicely handles this. Forgetting to handle empty stack when computing width can cause index errors or wrong area [82]. - *Edge Cases:* Test simple cases like a single bar (the largest rectangle is just that bar itself), or all bars of equal height (the largest rectangle is the whole histogram). Ensure the algorithm handles these. Our approach naturally handles single-bar (width becomes i - (-1) - 1 = i, and with sentinel at end it will compute correctly) and equal heights (they will all be popped at the end or as soon as a smaller height or end is encountered). Always verify the output for these basic scenarios.

Alternate Approaches: There is a known divide-and-conquer solution that finds the minimum bar and computes the largest rectangle recursively in left and right partitions [83]. With segment trees or sparse tables to find minima, it can run in O(n log n) or even O(n) in the best case. However, this is more complex to implement and typically not needed given the elegance and efficiency of the stack solution. Interviewers

usually expect the monotonic stack method for this problem because it's a canonical example of that technique.

Memorization and Patterns

When it comes to stack problems, several common patterns emerge. It helps to remember these patterns and the key tricks associated with each:

- **Matching Brackets Pattern:** Use a stack to validate pairs of symbols (like the *Valid Parentheses* problem). The stack holds opening symbols, and you check that each closing symbol matches the most recent opening [4]. Remember to check for stack underflow (empty stack when expecting a match) and to ensure the stack is empty at the end. This pattern applies to various problems involving well-formed strings (HTML tags, XML parsing, etc., not just parentheses).

- **Monotonic Stack (Next Greater Element):** Many problems (like *Daily Temperatures*) ask for the next greater or smaller element. A monotonic stack – one that keeps elements in decreasing (for next greater) or increasing (for next smaller) order – allows you to solve these in O(n) time [39]. The idea is to traverse and use the stack to find the next greater element for each item by popping until the invariant is restored. This pattern is useful for *Next Greater Element* problems, stock span problems, and *Daily Temperatures* is a direct application. Key memory tip: **"Push indices, not values, when you need to compute spans or distances."**

- **Monotonic Stack (Spans of Greater Elements):** A variant of the above is maintaining an increasing stack to find spans of where an element is minimum (as in *Largest Rectangle in Histogram*). In these, you often add a sentinel value at the end to flush out the stack [77]. The largest rectangle problem and the *Trapping Rain Water* problem (using left/right bounds) are classic uses of this pattern. Remember: **use a dummy 0 at end to clear the stack** – a handy trick for histogram problems.

- **Evaluation Stack:** Some problems use a stack to evaluate expressions or parse data. The *Evaluate Reverse Polish Notation* problem is a prime example where the stack holds operands and operators trigger computations [29]. Similar scenarios include evaluating *infix* expressions (where you might use two stacks, one for values and one for operators following the Shunting-yard algorithm) or parsing nested structures (where a stack can help accumulate results for something like "decode string" problems). A good memory tip here is: **"Push numbers, pop on operators."** And always be cautious about operator precedence and order (for subtraction/division, pop order matters [32]).

- **Supporting Stack (Min Stack):** Some design problems require augmenting a stack with extra functionality (like getMin or getMax). The typical solution is to use an additional stack to track the desired property [17]. For a Min Stack, memorize that you should push the new min at each push, and pop from min-stack on each pop. A quick check: after each push, `minStack.top()` is the min of all values below. If you ever pop and break this sync, your results will be wrong. **Memorize the two-stack approach** since it's a common interview question and quite implementable under pressure.

- **General Stack Usage:** Stacks are also handy for any *last-in, first-out* context, like depth-first search (implicit recursion stack or explicit stack), undo operations, or backtracking algorithms. While those are broader, it's worth remembering that a stack can sometimes replace recursion if needed.

To reinforce these concepts, here are some flashcard-style Q&A and mnemonics:

1. **Q:** Why do we use a stack for the Valid Parentheses problem?
   A: Because it naturally handles the *last-in, first-out* requirement of matching brackets. The most recently opened bracket must be the first one closed, which is exactly how a stack operates [4]. Push opens, and on a closing bracket, check and pop the stack.

2. **Q:** How can a stack help in finding the next greater element for each item in a list?
   A: By maintaining a **monotonic decreasing stack** of indices. As you iterate, you pop from the stack until the stack's top is an index with a value greater than the current value, meaning the current value is the next greater for all popped indices [40]. Then push the current index. This way, each element gets processed in a single pass.

3. **Q:** In a Min Stack, how is the minimum tracked so that getMin() is O(1)?
   A: By using a second stack (min-stack) that stores the minimum value *at that stack depth*. Every push adds a min value (either the new value, if it's the new minimum, or the previous minimum again) [17]. The top of the min-stack is always the current minimum. Popping both stacks together maintains correct state.

4. **Q:** What is a **monotonic stack** in simple terms?
   A: It's a stack that keeps its elements in sorted order – either non-increasing or non-decreasing. You typically push elements and pop whenever the order would be violated. For example, in a decreasing stack, you pop until the top is greater than the new element before pushing it [71]. This structure is useful for span and boundary problems.

5. **Q:** In evaluating an RPN expression, what common mistake must you avoid with subtraction and division?
   A: Getting the operand order wrong. After popping two numbers from the stack, remember the second popped is the left operand and the first popped is the right operand. Do `left - right` or `left / right` (truncating toward zero) [32]. Also, ensure you convert the tokens to numbers and don't confuse a negative number token with an operator.

6. **Q:** How do we ensure all bars are processed in the Largest Rectangle in Histogram stack solution?
   A: Push a **sentinel value** (height 0) at the end of the heights array (or simulate it by looping to `i = n` and treating height as 0) [77]. This forces the stack to empty out completely by popping any remaining bars. It's a handy trick to avoid writing a separate cleanup loop.

7. **Q:** What's a quick way to identify that a problem might be solved with a monotonic stack?
   A: If the problem involves finding "next greater" or "next smaller" elements, or spans of elements until a smaller/larger break (as in temperatures, stock span, histogram rectangle, etc.), it's a strong sign to use a monotonic stack. Look for that pattern of *"for each element, compare it to some future*

element" – that often screams stack (since a brute force would be double loop, which you can optimize with a stack).

By memorizing these patterns and why they work, you'll be well-equipped to recognize when a stack is the right tool. It's also useful to remember template code snippets for these common stack scenarios (e.g., the loop structure for next greater element, or the push/pop logic for min stack) so that you can write them quickly and correctly during interviews. With practice, these stack techniques will become almost second nature. Good luck, and remember: **think in LIFO terms when order matters!**

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 20. Valid Parentheses - Solution & Explanation
https://neetcode.io/solutions/valid-parentheses

15 16 17 18 19 20 21 22 23 24 25 26 155. Min Stack - Solution & Explanation
https://neetcode.io/solutions/min-stack

27 28 29 30 31 32 33 34 35 150. Evaluate Reverse Polish Notation - Solution & Explanation
https://neetcode.io/solutions/evaluate-reverse-polish-notation

36 Daily Temperatures - LeetCode
https://leetcode.com/problems/daily-temperatures/

37 38 39 40 41 42 43 44 45 46 47 739. Daily Temperatures - Solution & Explanation
https://neetcode.io/solutions/daily-temperatures

48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 853. Car Fleet - Solution & Explanation
https://neetcode.io/solutions/car-fleet

63 64 Largest Rectangular Area in a Histogram - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/largest-rectangular-area-in-a-histogram-using-stack/

65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84. Largest Rectangle In Histogram - Solution & Explanation
https://neetcode.io/solutions/largest-rectangle-in-histogram