Dynamic Programming

70. Climbing Stairs

Here is how a senior engineer approaches the "Climbing Stairs" problem. At an L5 or L6 level, an interviewer isn't just looking for a working solution; they are looking for clear communication, an understanding of the underlying patterns (Dynamic Programming), the ability to optimize trade-offs (specifically space complexity), and production-quality code.

Here is a comprehensive breakdown.

---

1. Problem Explanation

The Goal: You are at the bottom of a staircase with `N` steps. You want to reach the top. **The Constraint:** You can only take either 1 step or 2 steps at a time. **The Question:** How many *distinct* ways can you combine 1-step and 2-step jumps to reach exactly step `N`?

Let's visualize a small example where N = 3:

```
Target: Step 3

Start at Ground (Step 0).
Ways to climb:
Path A: 1 step  -> 1 step  -> 1 step  (Arrive at 3)
Path B: 1 step  -> 2 steps            (Arrive at 3)
Path C: 2 steps -> 1 step             (Arrive at 3)

Total distinct ways for N=3 is 3.
```

If any part of this is tricky, the key is to stop thinking about the whole staircase at once and look at just the very last move. If you are standing on Step N, where could you possibly have come from? Because you can only jump 1 or 2 steps, you *must* have arrived at Step N from either Step (N - 1) or Step (N - 2).

Therefore, the total ways to get to Step N is simply the sum of:

1. All the ways to get to Step (N - 1)
2. All the ways to get to Step (N - 2)

This is a classic overlapping subproblems scenario, which screams **Dynamic Programming**.

---

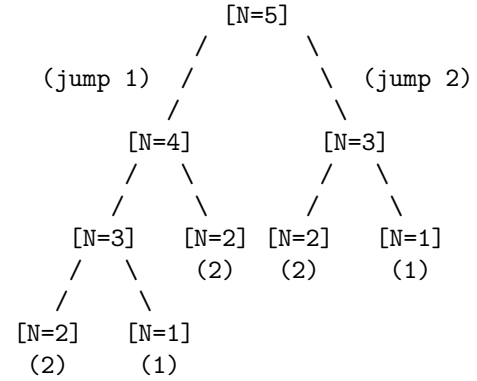
2. Solution Explanation

We will explore the most intuitive way first (Top-Down), and then show how a senior engineer optimizes it for production (Bottom-Up Space Optimized).

Approach A: Top-Down (Recursion + Memoization) - The Intuitive Start Top-down means we start at our goal (`N`) and break it down into smaller, identical problems until we hit base cases we already know the answer to.

- **Base Case 1:** If N = 1, there is only 1 way (take 1 step).
- **Base Case 2:** If N = 2, there are 2 ways (1+1, or 2).

Let's map out a pure recursive decision tree for N = 5.



The Non-Trivial Problem Here: Look at the tree above. We calculate `[N=3]` twice. We calculate `[N=2]` three times. As N grows (e.g., N = 45), this tree explodes exponentially, recalculating the same steps millions of times.

The Fix (Memoization): We introduce a "cache" (a dictionary or array). The first time we calculate the ways for `[N=3]`, we save it. The next time the tree asks for `[N=3]`, we return the saved answer immediately without drawing the rest of the branches.

```
CACHE STATE PROGRESSION (N=5):

Initially:  [ -, -, -, -, - ]
Calc N=2:   [ -, 2, -, -, - ] (Base case known)
Calc N=1:   [ 1, 2, -, -, - ] (Base case known)
Calc N=3:   [ 1, 2, 3, -, - ] (Because 1 + 2 = 3)
Calc N=4:   [ 1, 2, 3, 5, - ] (Because 2 + 3 = 5)
Calc N=5:   [ 1, 2, 3, 5, 8 ] (Because 3 + 5 = 8) -> Final Answer: 8
```

Approach B: Bottom-Up (Tabulation & Space Optimization) - The L5/L6 Standard Once you understand Top-Down, converting it to Bottom-Up is straightforward. Instead of starting at `N` and using recursion to dig down to the base cases, we just start at the base cases and build an array going up.

Step-by-step Tabulation visualization:

```
We want to reach N=5.
Let's build an array where array[i] = ways to reach step i.

Index:   0   1   2   3   4   5
Array: [ 1,  1,  2,  3,  5,  8 ]
                 ^   ^   ^   ^
                 |   |   |   |
      (1+1)------+   |   |   |
          (1+2)------+   |   |
              (2+3)------+   |
                  (3+5)------+
```

(Note: Index 0 is theoretically 1 way - doing nothing).

The Final Optimization (O(1) Space): A top-tier engineer will notice a critical detail in the Bottom-Up array: to calculate step 5, we *only* need step 4 and step 3. We absolutely do not care about step 1 or step 2 anymore. Therefore, keeping a whole array of size `N` is a waste of memory. We only need two variables to keep track of the "previous two" steps.

Space-Optimized ASCII Visualization:

```
Let's calculate N=5 using only 2 variables: prev_two and prev_one.

INITIAL STATE (At step 2):
Step 1    Step 2
[  1  ]   [  2  ]
  ^         ^
prev_two  prev_one

MOVE TO STEP 3:
current = prev_two + prev_one (1 + 2 = 3)
Shift variables forward...

Step 2    Step 3
[  2  ]   [  3  ]
  ^         ^
prev_two  prev_one

MOVE TO STEP 4:
current = prev_two + prev_one (2 + 3 = 5)
Shift variables forward...

Step 3    Step 4
[  3  ]   [  5  ]
  ^         ^
```

```
prev_two  prev_one

MOVE TO STEP 5:
current = prev_two + prev_one (3 + 5 = 8)
Shift variables forward...

Step 4    Step 5
[  5  ]   [  8  ]  <-- 8 is our final answer!
  ^         ^
prev_two  prev_one
```

---

3. Time and Space Complexity Analysis

Top-Down with Memoization:

```
Time Complexity: O(N)
Derivation:
Without cache: O(2^N) -> Tree branches 2 times per node.
With cache:    O(N)   -> Tree branches are pruned. We only calculate
                         each step exactly 1 time.
[Node N] -> [Node N-1] -> ... -> [Node 1] = N operations.

Space Complexity: O(N)
Derivation:
1. Recursion Stack: Max depth is N.  -> O(N) memory
2. Cache Array:     Stores N items.  -> O(N) memory
Total: O(N) + O(N) = O(N)
```

Bottom-Up Space Optimized:

```
Time Complexity: O(N)
Derivation:
We run a single simple loop from 3 up to N.
[Loop starts] ---> [Loop ends at N]
Total iterations = N - 2.
Constants drop out, leaving O(N).

Space Complexity: O(1) Constant Space
Derivation:
Variables allocated in memory:
1. prev_two (integer)
2. prev_one (integer)
3. current  (integer)
No matter if N = 5 or N = 50000, we only ever use exactly 3 integer
variables in memory. Therefore, space does not scale with N. -> O(1).
```

---

4. Solution Code

Here are the solutions in both Python and JavaScript.

Python Snippets

```python
# --- APPROACH 1: TOP-DOWN WITH MEMOIZATION ---
def climbStairs_topDown(n: int) -> int:
    """
    Solves Climbing Stairs using recursion and memoization.
    Time: O(N), Space: O(N)
    """
    # Dictionary to act as our cache.
    # Base cases are pre-loaded.
    memo = {1: 1, 2: 2}

    def solve(step):
        # If we have calculated this step before, return it immediately
        if step in memo:
            return memo[step]

        # Otherwise, calculate it by combining the previous two steps
        memo[step] = solve(step - 1) + solve(step - 2)
        return memo[step]

    return solve(n)


# --- APPROACH 2: BOTTOM-UP SPACE OPTIMIZED (L5/L6 EXPECTATION) ---
def climbStairs_bottomUp(n: int) -> int:
    """
    Solves Climbing Stairs using iterative tabulation with O(1) space.
    Time: O(N), Space: O(1)
    """
    # Handle base cases immediately to prevent out-of-bounds errors
    if n == 1:
        return 1
    if n == 2:
        return 2

    # Initialize the pointers for Step 1 and Step 2
    prev_two = 1
    prev_one = 2
```

```
    # Start iterating from Step 3 up to and including Step N
    for i in range(3, n + 1):
        # The core DP formula: current step is sum of previous two
        current = prev_two + prev_one

        # Shift our window forward for the next iteration
        prev_two = prev_one
        prev_one = current

    return prev_one
```

JavaScript Snippets

```
// --- APPROACH 1: TOP-DOWN WITH MEMOIZATION ---
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs_topDown = function(n) {
    // Array cache. Index matches the step number.
    const memo = new Array(n + 1).fill(-1);

    function solve(step) {
        // Base cases
        if (step === 1) return 1;
        if (step === 2) return 2;

        // Return cached result if it exists
        if (memo[step] !== -1) {
            return memo[step];
        }

        // Calculate, save to cache, and return
        memo[step] = solve(step - 1) + solve(step - 2);
        return memo[step];
    }

    return solve(n);
};

// --- APPROACH 2: BOTTOM-UP SPACE OPTIMIZED (L5/L6 EXPECTATION) ---
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs_bottomUp = function(n) {
```

```
    // Handle early returns for edge cases
    if (n === 1) return 1;
    if (n === 2) return 2;

    // Variables representing step (i-2) and step (i-1)
    let prevTwo = 1;
    let prevOne = 2;

    // Loop from step 3 to n
    for (let i = 3; i <= n; i++) {
        let current = prevTwo + prevOne;

        // Slide the variables up one step
        prevTwo = prevOne;
        prevOne = current;
    }

    // At the end of the loop, prevOne holds the value for step n
    return prevOne;
};
```

Note 1: Terms & Techniques Used

- **Dynamic Programming (DP):** An algorithmic paradigm that solves a complex problem by breaking it down into simpler subproblems, solving each of those just once, and storing their solutions. It applies here because the stairs problem is made of repeating smaller stair problems.
- **Memoization (Top-Down):** An optimization technique where you store the results of expensive recursive function calls and return the cached result when the same inputs occur again. It helps prevent the O(2^N) exponential time explosion.
- **Tabulation (Bottom-Up):** An approach where you start from the smallest subproblems, fill up a table (or variables), and build your way up to the final answer iteratively rather than recursively.
- **Space Optimization (Sliding Window):** A technique used in DP when the current state only depends on a fixed number of previous states (in this case, 2). It allows us to reduce memory usage from O(N) to O(1) by discarding older data we no longer need.

Note 2: Real-World & Interview Variants

Companies like Google, Meta, and Bloomberg rarely ask "Climbing Stairs" verbatim because it's too famous. Instead, they disguise it to test if you recognize the underlying `F(n) = F(n-1) + F(n-2)` pattern. Recent variations include:

1. **Decode Ways (Meta/Bloomberg):** You have a message mapped to numbers (A=1, B=2 … Z=26). Given a string of digits, how many ways can you decode it? (Very similar, but you have to check if combining 2 digits creates a valid number between 10 and 26).
2. **Min Cost Climbing Stairs (Google/Meta):** Each stair has an associated cost array. You can step 1 or 2 stairs, but you want to reach the top with the minimum total cost. (Changes the logic from `summing` branches to finding the `min()` of branches).
3. **Domino and Tromino Tiling (Google):** You have a `2 x N` board and want to fill it with dominoes (`2 x 1`). How many ways can you tile the board? (The mathematical breakdown is identical to climbing stairs for standard dominoes).
4. **Message Delivery / Ping Routing (Real World):** A packet needs to travel across nodes with varying packet-loss costs. Finding the most reliable route often uses these exact 1D DP concepts.

322. Coin Change

Here is how a senior engineer approaches the "Coin Change" problem. It is a classic optimization problem that tests your ability to break a large, complex task into smaller, manageable subproblems.

1. Problem Explanation

The Goal: You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. You need to return the fewest number of coins that make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Why the "Greedy" approach fails (Non-Trivial Concept): A common first instinct is a "Greedy" algorithm: always pick the largest coin possible first. *Example:* `coins = [1, 5, 10]`, `amount = 12`. Greedy picks: 10, then 1, then 1. Total: 3 coins. (This works).

Counter-Example where Greedy fails: `coins = [1, 3, 4]`, `amount = 6`. Greedy picks: 4, then 1, then 1. Total: 3 coins. Optimal solution: 3, then 3. Total: 2 coins.

Because the Greedy approach does not guarantee the optimal solution for arbitrary coin denominations, we must explore all valid combinations. However, exploring *every* combination is too slow. This is where Dynamic Programming (DP) comes in.

---

2. Solution Explanation

We will explore this using Dynamic Programming. The most intuitive way to grasp DP is usually starting Top-Down (Recursion + Memoization), and then converting it to Bottom-Up (Tabulation).

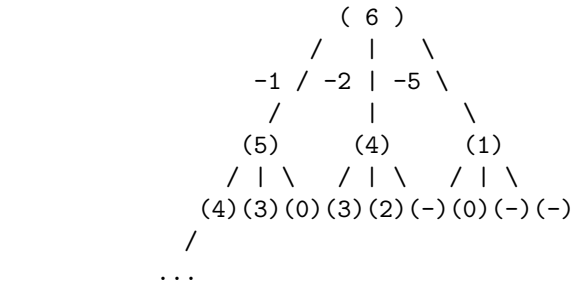
Approach A: Top-Down (Recursive with Memoization) Intuition: Imagine you want to make change for 11 cents using coins [1, 2, 5]. You can ask yourself: "If I use a 5-cent coin, how many coins do I need for the remaining 6 cents?" The answer for 11 cents is simply: 1 (for the coin I just picked) + the optimal answer for the remaining amount.

We try this for *every* coin and take the minimum:

- Use coin 1: Remaining is 10. Coins needed = 1 + optimal(10)
- Use coin 2: Remaining is 9. Coins needed = 1 + optimal(9)
- Use coin 5: Remaining is 6. Coins needed = 1 + optimal(6)
- Result = minimum of the above three.

The Problem: Overlapping Subproblems If we just use pure recursion, we will calculate the same amounts over and over.

`ASCII Recursion Tree for coins = [1, 2, 5], amount = 6`



Notice how `(4)` and `(3)` are evaluated multiple times? To fix this, we use a "memo" (a cache/dictionary) to store the answer for an amount once we compute it. Next time we see that amount, we return the cached answer immediately in O(1) time.

Approach B: Bottom-Up (Tabulation) - The Standard DP Approach Intuition: Instead of starting from the target amount and recursing down, we start from 0 and build our way up to the target amount. We use an array `dp` where `dp[i]` represents the minimum coins needed to make amount `i`.

Step-by-step Visualization: Let `coins = [1, 2, 5]` and `amount = 5`. We create an array `dp` of size `amount + 1` (size 6). We initialize it with a dummy maximum value (like Infinity, or simply `amount + 1` since we can never use more than `amount` coins of value 1). `dp[0] = 0` because it takes 0 coins to make amount 0.

```
Initial State: (Max value represented by 'X' for readability)
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  X,  X,  X,  X,  X ]
```

Now, we iterate through each amount from 1 to 5. For each amount, we check every coin.

Amount = 1:

- Coin 1: $1 <= 1$. `dp[1] = min(dp[1], dp[1 - 1] + 1) -> min(X, dp[0] + 1) -> 1`
- Coin 2: $2 > 1$. Skip.
- Coin 5: $5 > 1$. Skip.

```
State after Amount 1:
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  1,  X,  X,  X,  X ]
```

Amount = 2:

- Coin 1: $1 <= 2$. `dp[2] = min(dp[2], dp[2 - 1] + 1) -> min(X, dp[1] + 1) -> 2`
- Coin 2: $2 <= 2$. `dp[2] = min(dp[2], dp[2 - 2] + 1) -> min(2, dp[0] + 1) -> 1`
- Coin 5: $5 > 2$. Skip.

```
State after Amount 2:
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  1,  1,  X,  X,  X ]
```

Amount = 3:

- Coin 1: `dp[3] = min(X, dp[2] + 1) = 2`
- Coin 2: `dp[3] = min(2, dp[1] + 1) = 2`

```
State after Amount 3:
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  1,  1,  2,  X,  X ]
```

Amount = 4:

- Coin 1: `dp[4] = min(X, dp[3] + 1) = 3`
- Coin 2: `dp[4] = min(3, dp[2] + 1) = 2`

```
State after Amount 4:
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  1,  1,  2,  2,  X ]
```

Amount = 5:

- Coin 1: `dp[5] = min(X, dp[4] + 1) = 3`
- Coin 2: `dp[5] = min(3, dp[3] + 1) = 3`
- Coin 5: `dp[5] = min(3, dp[0] + 1) = 1`

```
Final State:
Amount (Index):  0   1   2   3   4   5
dp array:      [ 0,  1,  1,  2,  2,  1 ]
```

The answer is `dp[5]`, which is `1`.

---

3. Time and Space Complexity Analysis

Let **A** be the `amount` and **C** be the number of `coins`.

Time Complexity Derivation:

```
Amount (A)     : [ 0 ] [ 1 ] [ 2 ] ... [ A-1 ] [ A ]
                   |     |     |          |      |
For EACH cell,     v     v     v          v      v
we iterate      (C coins)  (C coins)  ...      (C coins)
through coins.

Total Operations = (Number of cells) * (Operations per cell)
Total Operations = A * C
```

- **Time Complexity:** O(A * C) for both Top-Down (due to memoization caching the states) and Bottom-Up approaches.

Space Complexity Derivation:

```
Array/Cache Size : [ 0, 1, 2, ..., A ]
Number of items  : A + 1 elements stored in memory
```

- **Space Complexity:** O(A)
- Bottom-Up: The DP array takes O(A) space.
- Top-Down: The recursion stack can go as deep as A (if we use only 1-cent coins), and the memoization hash map takes O(A) space.

---

4. Solution Code

JavaScript

```javascript
/**
 * Bottom-Up Approach (Tabulation)
 */
function coinChangeBottomUp(coins, amount) {
    // Initialize an array of size amount + 1.
    // Fill it with a value larger than any possible valid amount (amount + 1)
    const dp = new Array(amount + 1).fill(amount + 1);

    // Base case: 0 coins needed to make amount 0
```

```
    dp[0] = 0;

    // Iterate through every amount from 1 to the target amount
    for (let i = 1; i <= amount; i++) {
        // For each amount, try every coin
        for (let coin of coins) {
            // If the coin value is less than or equal to the current amount
            if (i - coin >= 0) {
                // Update the DP array with the minimum coins needed
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    // If dp[amount] is still amount + 1, it means we couldn't make the amount
    return dp[amount] === amount + 1 ? -1 : dp[amount];
}

/**
 * Top-Down Approach (Recursion + Memoization)
 */
function coinChangeTopDown(coins, amount) {
    const memo = new Map();

    // Helper function to perform DFS
    function dfs(rem) {
        if (rem === 0) return 0; // Base case: exact change found
        if (rem < 0) return -1;  // Base case: overshot the amount

        // If we already computed the answer for this remaining amount, return it
        if (memo.has(rem)) return memo.get(rem);

        let minCoins = Infinity;

        // Explore taking each coin
        for (let coin of coins) {
            const res = dfs(rem - coin);
            // If the recursive call returned a valid answer, update our minimum
            if (res >= 0 && res < minCoins) {
                minCoins = res + 1;
            }
        }

        // Cache the result. If minCoins is still Infinity, it's impossible.
        memo.set(rem, minCoins === Infinity ? -1 : minCoins);
        return memo.get(rem);
```

```
    }

    return dfs(amount);
}
```

Python

```python
# Bottom-Up Approach (Tabulation)
def coin_change_bottom_up(coins: list[int], amount: int) -> int:
    # Initialize DP array with amount + 1 (acts as our "infinity")
    dp = [amount + 1] * (amount + 1)
    dp[0] = 0

    # Iterate through all amounts from 1 up to 'amount'
    for i in range(1, amount + 1):
        for coin in coins:
            # Check if we can use this coin
            if i - coin >= 0:
                # State transition: min(current best, best for remaining amount + 1)
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != amount + 1 else -1


# Top-Down Approach (Recursion + Memoization)
def coin_change_top_down(coins: list[int], amount: int) -> int:
    memo = {}

    def dfs(rem: int) -> int:
        if rem == 0:
            return 0
        if rem < 0:
            return -1

        if rem in memo:
            return memo[rem]

        min_coins = float('inf')

        for coin in coins:
            res = dfs(rem - coin)
            if res >= 0 and res < min_coins:
                min_coins = res + 1

        # Store in memo dictionary
        memo[rem] = min_coins if min_coins != float('inf') else -1
```

```
        return memo[rem]

    return dfs(amount)
```

Glossary of Terms

- **Dynamic Programming (DP):** An algorithmic technique for solving complex problems by breaking them down into simpler, overlapping subproblems, and storing the results of these subproblems to avoid redundant computations. *Why it helps here:* It drastically reduces the time complexity from exponential O(coins^amount) to linear O(coins * amount).
- **Memoization (Top-Down):** An optimization technique where you execute a recursive function and "memoize" (cache/save) the return value for a specific set of parameters. *Applies here:* By storing the minimum coins needed for an intermediate amount, we prevent the recursion tree from re-evaluating branches we've already solved.
- **Tabulation (Bottom-Up):** A DP approach where you solve subproblems starting from the smallest possible base case and build a table (array) up to the final target state. *Applies here:* The `dp` array is our table, building from amount 0 to the target amount.
- **Unbounded Knapsack Problem:** A category of problems where you want to maximize or minimize a value given a capacity, and you can use an unlimited supply of each item. *Applies here:* The `amount` is our "capacity", the `coins` are our "items", and we have an unlimited supply of them to minimize the count.

Real-World / Interview Variations (Google, Meta, Bloomberg)

Companies often disguise this problem to test if you can recognize the underlying pattern.

- **Google:** "Given a list of available server capacities (e.g., 2GB, 4GB, 8GB), what is the minimum number of server instances needed to process exactly N GBs of data in parallel without leaving any capacity unused?"
- **Meta:** "You are mailing a package and need exactly 87 cents in postage. You have stamps in denominations of 5, 10, and 32 cents. Write a function to determine the fewest number of stamps required."
- **Bloomberg:** "A trader wants to buy exactly N shares of a stock. The exchange only allows trading in specific lot sizes (e.g., lots of 10, 50, and 100 shares). Find the minimum number of transaction lots needed to hit the exact target."

300. Longest Increasing Subsequence

Here is how a senior engineer approaches the Longest Increasing Subsequence (LIS) problem. A top-tier engineer does not just jump into writing loops; they clearly define the problem space, model the state, and evaluate trade-offs between different Dynamic Programming (DP) approaches.

1. Problem Explanation

We are given an integer array `nums`. We need to return the length of the longest strictly increasing subsequence.

- **Subarray vs. Subsequence:** A subarray is a contiguous part of an array. A subsequence is formed by deleting some (or no) elements without changing the order of the remaining elements.
- **Strictly Increasing:** Each chosen element must be strictly greater than the previously chosen element. Ties (equal values) are not allowed.

Example Walkthrough Input: `nums = [10, 9, 2, 5, 3, 7, 101, 18]`

- `[10, 101]` is a valid increasing subsequence (length 2).
- `[2, 5, 7, 101]` is a valid increasing subsequence (length 4).
- `[2, 3, 7, 18]` is another valid increasing subsequence (length 4).

The maximum length we can achieve is 4.

---

2. Solution Explanation

Dynamic Programming is the most intuitive way to solve this. DP is all about breaking a large problem into smaller, manageable subproblems, solving them, and reusing the results.

The Top-Down Approach (Recursion + Memoization)

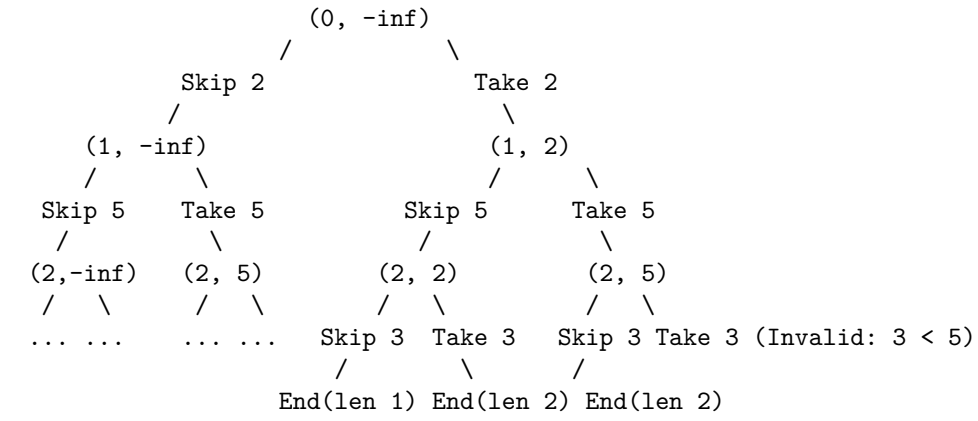
The most natural way to think about subsequences is to go element by element and make a decision: "Do I include this number in my sequence, or do I skip it?"

To make this decision, we need to know two things (our "state"):

1. **Current Index:** Which element are we looking at right now?
2. **Previous Index:** What was the last element we decided to include? (This dictates whether we *can* include the current element, because it must be strictly greater).

Let's visualize the decision tree for a smaller array: `nums = [2, 5, 3]`

```
State: (current_index, previous_value)
Start: (0, -infinity)
```



Notice how states like `(2, 5)` can be reached through multiple paths. A naive recursive tree recalculates these paths. By storing the results of `(current_index, previous_index)` in a dictionary or 2D array, we "memoize" the results and avoid redundant work.

The Bottom-Up Approach (Tabulation)

Top-down recursion can hit maximum recursion depth limits and carries call-stack overhead. A senior engineer will often convert this to a bottom-up approach.

Instead of asking "What elements come next?", we ask: "If my subsequence *ends* at the current element, what is the longest sequence I could have built up to this point?"

We use an array `dp` where `dp[i]` represents the length of the longest increasing subsequence that ends exactly at `nums[i]`.

Let's visualize building this DP array step-by-step for `nums = [2, 5, 3, 7]`.

Initialization: Every element is a valid subsequence of length 1 on its own.

```
nums:  [ 2,  5,  3,  7 ]
dp:    [ 1,  1,  1,  1 ]
```

Step 1 (i = 1, current = 5): Look at all previous numbers (j = 0). Is `nums[0]` (2) < `nums[1]` (5)? Yes. `dp[1] = max(dp[1], dp[0] + 1) = max(1, 1 + 1) = 2`

```
         j   i
nums:  [ 2,  5,  3,  7 ]
dp:    [ 1,  2,  1,  1 ]
```

Step 2 (i = 2, current = 3): Look at previous numbers (j = 0, 1). j=0: Is 2 < 3? Yes. `dp[2] = max(1, dp[0] + 1) = 2` j=1: Is 5 < 3? No. Skip.

```
             j   i
nums:  [ 2,  5,  3,  7 ]
dp:    [ 1,  2,  2,  1 ]
```

Step 3 (i = 3, current = 7): Look at previous numbers (j = 0, 1, 2). j=0: Is 2 < 7? Yes. `dp[3] = max(1, dp[0] + 1) = 2` j=1: Is 5 < 7? Yes. `dp[3] = max(2, dp[1] + 1) = 3` j=2: Is 3 < 7? Yes. `dp[3] = max(3, dp[2] + 1) = 3`

```
                 j   i
nums:  [ 2,  5,  3,  7 ]
dp:    [ 1,  2,  2,  3 ]
```

The answer is the maximum value in the `dp` array, which is 3.

---

3. Time and Space Complexity Analysis

Here is the visual derivation for the Bottom-Up approach complexities.

Time Complexity Derivation:

```
Array Size = N
Outer Loop (i) runs N times.
Inner Loop (j) runs 'i' times for each 'i'.

i = 0 -> j runs 0 times
i = 1 -> j runs 1 time
i = 2 -> j runs 2 times
...
i = N-1 -> j runs N-1 times

Total Operations = 0 + 1 + 2 + ... + (N-1)
                 = (N * (N - 1)) / 2

Drop the constants and lower order terms -> Time Complexity = O(N^2)
```

Space Complexity Derivation:

```
We created a single 1D array 'dp' of the exact same length as 'nums'.
nums: [ x, x, x, ..., x ] -> Length N
dp:   [ y, y, y, ..., y ] -> Length N

Additional variables: i, j, max_val -> O(1)

Total Space = Array of size N -> Space Complexity = O(N)
```

(Note: The Top-Down approach takes O(N^2) time and O(N^2) space because the memoization table must store results for N current indices multiplied by N

previous indices).

---

4. Solution Code

Python Code

```python
# ------------------------------------------------------------
# Approach 1: Top-Down DP (Memoization)
# ------------------------------------------------------------
def lengthOfLISTopDown(nums):
    # Dictionary to store solved subproblems.
    # Keys: (current_index, previous_index)
    memo = {}

    def dfs(curr_idx, prev_idx):
        # Base case: We reached the end of the array
        if curr_idx == len(nums):
            return 0

        # Check if we have already solved this exact state
        if (curr_idx, prev_idx) in memo:
            return memo[(curr_idx, prev_idx)]

        # Option 1: Skip the current element
        skip = dfs(curr_idx + 1, prev_idx)

        # Option 2: Take the current element
        # (only if it's the first element or strictly greater than previous)
        take = 0
        if prev_idx == -1 or nums[curr_idx] > nums[prev_idx]:
            take = 1 + dfs(curr_idx + 1, curr_idx)

        # Store the maximum of taking or skipping
        memo[(curr_idx, prev_idx)] = max(skip, take)
        return memo[(curr_idx, prev_idx)]

    return dfs(0, -1)


# ------------------------------------------------------------
# Approach 2: Bottom-Up DP (Tabulation)
# ------------------------------------------------------------
def lengthOfLISBottomUp(nums):
    if not nums:
        return 0
```

```
    # Initialize DP array with 1s.
    # Every element is a subsequence of length 1 by itself.
    dp = [1] * len(nums)

    # Iterate through the array
    for i in range(1, len(nums)):
        # Look back at all previous elements
        for j in range(i):
            # If the current element is greater than the previous element,
            # we can extend the subsequence ending at j
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    # The longest subsequence could end anywhere, so return the max of the DP array
    return max(dp)
```

JavaScript Code

```
// ------------------------------------------------------------
// Approach 1: Top-Down DP (Memoization)
// ------------------------------------------------------------
function lengthOfLISTopDown(nums) {
    // 2D Array initialized with undefined
    // Rows = curr_idx, Cols = prev_idx + 1 (to handle -1)
    const memo = Array.from({ length: nums.length }, () => Array(nums.length + 1).fill(undef

    function dfs(currIdx, prevIdx) {
        if (currIdx === nums.length) return 0;

        // Use prevIdx + 1 to avoid negative index mapping
        if (memo[currIdx][prevIdx + 1] !== undefined) {
            return memo[currIdx][prevIdx + 1];
        }

        let skip = dfs(currIdx + 1, prevIdx);
        let take = 0;

        if (prevIdx === -1 || nums[currIdx] > nums[prevIdx]) {
            take = 1 + dfs(currIdx + 1, currIdx);
        }

        memo[currIdx][prevIdx + 1] = Math.max(skip, take);
        return memo[currIdx][prevIdx + 1];
    }
```

```
    return dfs(0, -1);
}

// ------------------------------------------------------------
// Approach 2: Bottom-Up DP (Tabulation)
// ------------------------------------------------------------
function lengthOfLISBottomUp(nums) {
    if (nums.length === 0) return 0;

    const dp = new Array(nums.length).fill(1);

    for (let i = 1; i < nums.length; i++) {
        for (let j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }

    return Math.max(...dp);
}
```

---

5. Key Terms & Advanced L5/L6 Concepts

- **Memoization:** A top-down optimization technique where you execute a recursive function and cache the return values of expensive function calls. If the function is called again with the same inputs, you return the cached result instead of recalculating it. It helps convert exponential time recursive trees into polynomial time.
- **Tabulation:** A bottom-up optimization technique where you systematically fill a table (like our `dp` array) starting from the smallest, simplest subproblems up to the main problem. It removes the memory overhead of the recursive call stack.
- **Patience Sorting / Binary Search (The True Optimal Solution):** While O(N^2) DP is standard, an L5/L6 engineer must mention that LIS can actually be solved in O(N * log N) time. This is done by maintaining an active array of the smallest tail elements of all increasing subsequences seen so far, and using Binary Search to find the insertion point for each new element.

6. Real-World / Interview Variations

Top tech companies rarely ask LIS directly anymore. They mask it behind business logic:

- **Russian Doll Envelopes (Google/Meta):** You are given 2D envelopes `[width, height]`. You can only fit one inside another if both width and height are strictly smaller. By sorting widths ascending and heights descending, the problem reduces to finding the 1D LIS on the heights.
- **Box Stacking (Bloomberg):** Similar to envelopes, but you can rotate boxes in 3D space to maximize height. It requires generating all rotations and applying LIS.
- **Building Bridges:** You are given pairs of coordinates representing cities on two sides of a river. You must build straight bridges without them crossing. Sorting one side and running LIS on the other side yields the answer.

1143. Longest Common Subsequence

Here is how a senior engineer would approach, break down, and communicate the solution to the Longest Common Subsequence (LCS) problem.

1. Problem Explanation

At its core, the problem asks us to find the length of the longest "subsequence" shared between two strings, `text1` and `text2`.

The Non-Trivial Part: Subsequence vs. Substring Before writing any code, we must clearly define what a subsequence is, as it's the most common pitfall.

- A **substring** must be contiguous (characters next to each other). For example, in "abcde", "bcd" is a substring.
- A **subsequence** does *not* need to be contiguous, but it *must* maintain the relative order. For example, in "abcde", "ace" is a valid subsequence. We just "deleted" 'b' and 'd'.

Example:

- `text1` = "abcde"
- `text2` = "ace"
- **Result**: 3. The longest common subsequence is "ace".

If there is no common subsequence (e.g., "abc" and "def"), the answer is 0.

---

2. Solution Explanation

This is a classic optimization problem where we are looking for the "longest" of something. Whenever you see a problem asking for a maximum/minimum/longest property involving strings or arrays, **Dynamic Programming (DP)** should be your first instinct.

Let's explore the most intuitive way first: **Top-Down (Recursion + Memoization)**, and then we'll flip it to the industry standard **Bottom-Up (Tabulation)**.

Approach A: Top-Down (Recursive Intuition) Think about comparing the two strings character by character from the beginning (or end). Let's say we are comparing `text1` at index `i` and `text2` at index `j`.

We have exactly two scenarios:

1. **Match! (`text1[i] == text2[j]`)**: We found a common character! We add 1 to our total length, and we move *both* pointers forward to see what else matches.
2. **Mismatch (`text1[i] != text2[j]`)**: The current characters don't match. We don't know which string has the character we need next. So, we branch into two parallel universes:

- Universe A: Skip the current character in `text1` and keep `text2` as is.
- Universe B: Skip the current character in `text2` and keep `text1` as is.
- We take the `max` of these two universes.

Let's visualize this with an ASCII decision tree for `text1 = "abc"`, `text2 = "ac"`.

```
Evaluating: LCS("abc", "ac")
Pointers: i=0, j=0 ('a' vs 'a')

                    [ "abc", "ac" ]
                           |
                    'a' == 'a' (MATCH!)
                    Result = 1 + ...
                           |
                     [ "bc", "c" ]  <-- Moved both pointers
                    i=1, j=1 ('b' vs 'c')
                           |
                    'b' != 'c' (MISMATCH)
                    Branching into 2 paths!
                   /                       \
     Skip 'b' in text1                 Skip 'c' in text2
     [ "c", "c" ]                        [ "bc", "" ]
     i=2, j=1                            i=1, j=2
         |                                   |
   'c' == 'c' (MATCH!)                 Hit end of text2!
    Result = 1 + ...                   (Base Case: returns 0)
         |
    [ "", "" ]
    i=3, j=2
```

```
Hit end of both!
 (Base Case: 0)
```

Notice how if the strings were longer, we would eventually evaluate the same sub-strings over and over. To prevent this `O(2^(m+n))` explosion, we "memoize" or cache the results of `LCS(i, j)`.

Approach B: Bottom-Up (Tabulation) While Top-Down is intuitive, it relies on the call stack, which can cause stack overflow for massive strings. A senior engineer will often write out the Bottom-Up approach for production code.

Instead of starting at the whole string and breaking it down, we start at the base cases (empty strings) and build up to the whole string using a 2D grid.

Let's build a grid where `grid[row][col]` represents the LCS of the prefixes of `text1` up to `row` and `text2` up to `col`. Let's use `text1 = "abc"`, `text2 = "ac"`.

Step 1: The Base Cases (Initialization) If either string is empty (""), the LCS is 0. We add an extra row and column of 0s to represent this.

```
       ""   a    c   (text2)
      ----------------
 ""  | 0  | 0  | 0  |
      ----------------
 a   | 0  | .  | .  |
      ----------------
 b   | 0  | .  | .  |
      ----------------
 c   | 0  | .  | .  |
      ----------------
(text1)
```

Step 2: Filling the Grid The rules for filling the grid are the exact same as our recursive universes:

- If `text1[row-1] == text2[col-1]`, we take the diagonal value (which represents the string before this match) and add 1. `grid[row][col] = 1 + grid[row-1][col-1]`
- If they don't match, we take the max of the cell directly above or directly to the left. `grid[row][col] = max(grid[row-1][col], grid[row][col-1])`

Let's evaluate row 'a' (row 1):

- 'a' vs 'a' (Match!) -> 1 + diagonal(0) = 1.
- 'a' vs 'c' (Mismatch) -> max(top(0), left(1)) = 1.

```
      ""   a    c
     ---------------
""   | 0  | 0  | 0  |
     ---------------
a    | 0  | 1  | 1  |
     ---------------
b    | 0  | .  | .  |
     ---------------
c    | 0  | .  | .  |
     ---------------
```

Let's evaluate row 'b' (row 2):

- 'b' vs 'a' (Mismatch) -> max(top(1), left(0)) = 1.
- 'b' vs 'c' (Mismatch) -> max(top(1), left(1)) = 1.

```
      ""   a    c
     ---------------
""   | 0  | 0  | 0  |
     ---------------
a    | 0  | 1  | 1  |
     ---------------
b    | 0  | 1  | 1  |
     ---------------
c    | 0  | .  | .  |
     ---------------
```

Let's evaluate row 'c' (row 3):

- 'c' vs 'a' (Mismatch) -> max(top(1), left(0)) = 1.
- 'c' vs 'c' (Match!) -> 1 + diagonal(1) = 2.

```
      ""   a    c
     ---------------
""   | 0  | 0  | 0  |
     ---------------
a    | 0  | 1  | 1  |
     ---------------
b    | 0  | 1  | 1  |
     ---------------
c    | 0  | 1  | 2  |  <-- Final Answer!
     ---------------
```

The bottom-right cell contains the length of the Longest Common Subsequence.

---

3. Time and Space Complexity Analysis

Let `m` be the length of `text1` and `n` be the length of `text2`.

```
Time Complexity Derivation:
We are iterating through a 2D grid.

    +-------------------+
    | O(1) | O(1) | O(1)|
    +-------------------+
    | O(1) | O(1) | O(1)|   (m rows tall)
    +-------------------+
    | O(1) | O(1) | O(1)|
    +-------------------+
       (n columns wide)

Total Time = (m rows) * (n columns) * O(1) work per cell
Time Complexity = O(m * n)

Space Complexity Derivation:
Our 2D array stores an integer for every combination of prefixes.

Memory Allocated = Grid of size (m + 1) * (n + 1)
Space Complexity = O(m * n)

[L6 Optimization Note]:
Because we only ever look at the CURRENT row and the PREVIOUS row
(the cell above, the cell to the left, or the cell diagonal),
we don't need to keep the whole m * n grid in memory!
We can reduce the Space Complexity to O(min(m, n)) by only keeping
two 1D arrays (a "previous row" and a "current row").
```

---

4. Solution Code

Here are both approaches. As an L5/L6, I would provide the bottom-up approach first as the primary solution, but be fully prepared to code the top-down if the interviewer prefers it.

Python Code

```python
class Solution:
    # --- BOTTOM UP (Tabulation) ---
    def longestCommonSubsequence_BottomUp(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)

        # Create a 2D grid filled with 0s.
        # Dimensions: (m+1) x (n+1) to handle the empty string base cases.
        dp = [[0] * (n + 1) for _ in range(m + 1)]
```

```
        # Iterate through every character of both strings
        for row in range(1, m + 1):
            for col in range(1, n + 1):

                # DP Transition logic:
                # If characters match, add 1 to the diagonal (LCS of prefixes without these
                if text1[row - 1] == text2[col - 1]:
                    dp[row][col] = 1 + dp[row - 1][col - 1]

                # If characters don't match, take the best we've seen so far
                # either by skipping a char in text1 (look UP) or text2 (look LEFT)
                else:
                    dp[row][col] = max(dp[row - 1][col], dp[row][col - 1])

        return dp[m][n]

    # --- TOP DOWN (Memoization) ---
    def longestCommonSubsequence_TopDown(self, text1: str, text2: str) -> int:
        # Cache to store previously computed subproblems to avoid O(2^(m+n)) runtime
        memo = {}

        def solve(i, j):
            # Base case: if either pointer goes out of bounds, no more common chars
            if i == len(text1) or j == len(text2):
                return 0

            if (i, j) in memo:
                return memo[(i, j)]

            # Match scenario
            if text1[i] == text2[j]:
                memo[(i, j)] = 1 + solve(i + 1, j + 1)
            # Mismatch scenario
            else:
                memo[(i, j)] = max(solve(i + 1, j), solve(i, j + 1))

            return memo[(i, j)]

        return solve(0, 0)
```

JavaScript Code

```
// --- BOTTOM UP (Tabulation) ---
var longestCommonSubsequence_BottomUp = function(text1, text2) {
    const m = text1.length;
    const n = text2.length;
```

```
    // Create an (m+1) by (n+1) grid initialized with 0s
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    for (let row = 1; row <= m; row++) {
        for (let col = 1; col <= n; col++) {

            // DP Transition logic:
            // Match: take diagonal + 1
            if (text1[row - 1] === text2[col - 1]) {
                dp[row][col] = 1 + dp[row - 1][col - 1];
            }
            // Mismatch: take max of Top or Left
            else {
                dp[row][col] = Math.max(dp[row - 1][col], dp[row][col - 1]);
            }
        }
    }

    return dp[m][n];
};

// --- TOP DOWN (Memoization) ---
var longestCommonSubsequence_TopDown = function(text1, text2) {
    // 2D Array for memoization initialized with -1
    const memo = Array.from({ length: text1.length }, () => Array(text2.length).fill(-1));

    function solve(i, j) {
        // Base case: end of string
        if (i === text1.length || j === text2.length) return 0;

        // Return cached result if already computed
        if (memo[i][j] !== -1) return memo[i][j];

        if (text1[i] === text2[j]) {
            memo[i][j] = 1 + solve(i + 1, j + 1);
        } else {
            memo[i][j] = Math.max(solve(i + 1, j), solve(i, j + 1));
        }

        return memo[i][j];
    }

    return solve(0, 0);
};
```

Glossary of Terms & Techniques Used

- **Dynamic Programming (DP):** An algorithmic technique for solving optimization problems by breaking them down into simpler overlapping subproblems. Instead of solving the same subproblem repeatedly, DP solves each one once and stores the result.
- **Top-Down (Memoization):** A DP approach starting from the main problem, recursively breaking it down, and "caching" (memoizing) the answers to subproblems in a hash map or array so they are never computed twice. It feels like natural problem-solving.
- **Bottom-Up (Tabulation):** A DP approach where you solve the smallest possible subproblems first (usually filling out a table or grid) and use their answers to iteratively build up to the main problem. It avoids recursive call stack overhead.
- **Space Optimization (1D DP Array):** A technique applied to Bottom-Up DP. If your grid transition only requires looking at the current row and the immediate row above it, you don't need a full `O(m * n)` matrix. You can simply maintain two `O(n)` arrays, swapping them as you iterate downwards, saving massive amounts of memory.

Real-World / Interview Variations (Google, Meta, Bloomberg)

In top-tier interviews, you rarely get asked plain "Longest Common Subsequence." Instead, it is disguised as real-world applications:

1. **"Implement a simplified Git Diff tool":** This is a classic Google question. Given two text files (arrays of strings), print out what was deleted and what was added to get from File A to File B. The core engine of this is finding the LCS to know what *didn't* change.
2. **"Edit Distance" (Levenshtein Distance):** Given two strings, find the minimum number of operations (insert, delete, replace) required to convert one word into another. Very similar DP grid setup, just different transition logic. Highly favored by Meta.
3. **"Longest Palindromic Subsequence":** Given a single string, find the longest subsequence that is a palindrome. The "trick" here is that this is literally just LCS in disguise! You just run LCS on `(string, reverse_of_string)`. Bloomberg asks this frequently.
4. **Interleaving Strings:** Given three strings `s1`, `s2`, and `s3`, find whether `s3` is formed by an interleaving of `s1` and `s2`. This tests the same "two-pointer with parallel universe branching" intuition.

139. Word Break

Here is how a senior engineer approaches the "Word Break" problem. At an L5/L6 level, the focus isn't just on getting a working solution, but on clearly defining the state space, analyzing the tradeoffs between different dynamic programming approaches, and writing production-ready, highly readable code.

---

1. Problem Explanation

The Goal: You are given a main string (`s`) and a list of valid dictionary words (`wordDict`). You need to determine if you can completely divide (or "segment") the main string into a sequence of one or more words that exist in the dictionary.

The Rules:

- You can use the same word from the dictionary multiple times.
- You cannot have any characters "left over" in the main string. Everything must map to a dictionary word.
- You cannot rearrange the characters; the order must remain intact.

Example Walkthrough:

- **Input:** `s = "catsand"`, `wordDict = ["cat", "cats", "and", "sand"]`
- **Output:** `True`
- **Why?** Because we can segment `"catsand"` into `"cats" + "and"` (or `"cat" + "sand"`). Both are valid partitions using exactly the words from our dictionary.

---

2. Solution Explanation

This problem screams **Dynamic Programming (DP)**. Why? Because it features overlapping subproblems. If we want to know if `"catsand"` can be broken down, we can check if `"cat"` is a word, and then ask the exact same fundamental question for the remainder of the string: "Can `"sand"` be broken down?"

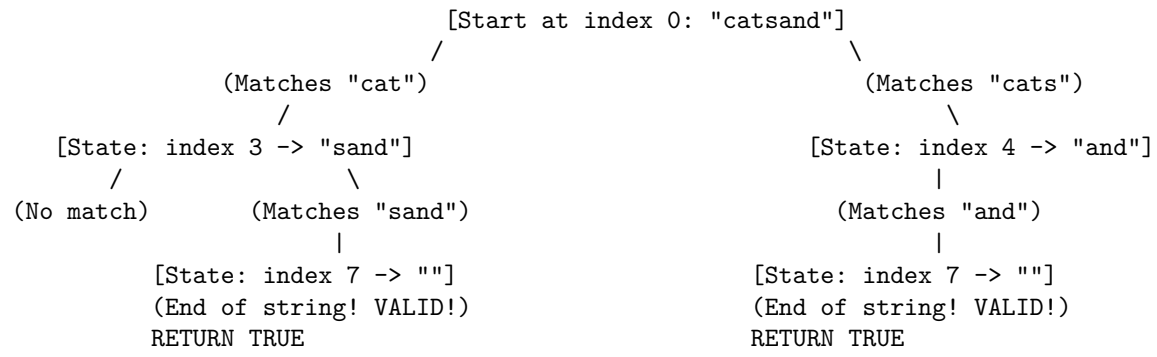
An L5 engineer will always start by defining the **state**.

- **State `i**`: Represents the starting index of the substring we are currently evaluating.
- **Question at State `i**`: Can the substring from index `i` to the end of the string, `s[i...n-1]`, be segmented into dictionary words?

Approach 1: Top-Down Dynamic Programming (Memoization)

This is the most intuitive way for human brains to process the problem. We start at the beginning of the string and recursively explore paths. To prevent recalculating the same substrings, we save (memoize) the results of indices we've already visited.

ASCII Visualization: Top-Down Execution Tree Let's use `s = "catsand"`, `wordDict = ["cat", "cats", "and", "sand"]`



If we ever encountered a state (an index) that we had evaluated before and found to be `False`, our cache (memoization map) would immediately return `False` rather than expanding that entire branch again.

Approach 2: Bottom-Up Dynamic Programming (Tabulation)

Instead of recursion, we build an array from the ground up. We create a boolean array `dp` of size `N + 1` (where N is the length of `s`).

- `dp[i]` represents: "Can the prefix of the string up to index `i` be formed by dictionary words?"
- We seed the start: `dp[0] = True` (an empty string requires 0 words, which is trivially true).

We iterate through the string. If `dp[i]` is `True` (meaning we've successfully built a valid sequence of words up to this point), we check if any dictionary words match the string starting at index `i`. If they do, we mark the ending index of that word as `True`.

ASCII Visualization: Bottom-Up Array State Let's use `s = "leetcode"`, `wordDict = ["leet", "code"]`

```
INITIAL STATE:
String:   l   e   e   t   c   o   d   e
Index:  0   1   2   3   4   5   6   7   8
dp:    [T] [F] [F] [F] [F] [F] [F] [F] [F]
        ^

      dp[0] is True. Empty string is valid.
```

```
STEP 1 (i = 0):
Since dp[0] == True, we check our dictionary words starting at index 0.
Word "leet" matches s[0...3]! The word "leet" is 4 characters long.
We set dp[0 + 4] to True.

String:   l   e   e   t   c   o   d   e
Index:  0   1   2   3   4   5   6   7   8
dp:    [T] [F] [F] [F] [T] [F] [F] [F] [F]
                        ^
                dp[4] becomes True

STEPS 2, 3, 4 (i = 1, 2, 3):
dp[1], dp[2], and dp[3] are False. We cannot start new words from these
indices because no valid sequence of words brought us to these points.
We skip them.

STEP 5 (i = 4):
dp[4] == True. We check our dictionary words starting at index 4 ("code").
Word "code" matches s[4...7]! The word "code" is 4 characters long.
We set dp[4 + 4] to True.

String:   l   e   e   t   c   o   d   e
Index:  0   1   2   3   4   5   6   7   8
dp:    [T] [F] [F] [F] [T] [F] [F] [F] [T]
                                        ^
                                dp[8] becomes True

CONCLUSION:
We check dp[8] (length of string). It is True!
Return True.
```

---

3. Time and Space Complexity Analysis

Here is the visual derivation for the optimal implementation (iterating through the dictionary at each state). Let `N` be the length of string `s`, `M` be the number of words in `wordDict`, and `L` be the maximum length of a word in `wordDict`.

```
+-------------------------------------------------------------+
| TIME COMPLEXITY DERIVATION                                  |
+-------------------------------------------------------------+
| [1] Maximum number of unique states (indices in string 's') |
|     Total states = N                                        |
|                                                             |
| [2] Work done at each state                                 |
```

```
|     Iterate through every word in 'wordDict'                |
|     Total words checked per state = M                       |
|                                                             |
| [3] Cost of string comparison for each word                 |
|     Maximum length of a word = L                            |
|     Substring comparison takes O(L) time                    |
|                                                             |
| DERIVATION PATH:                                            |
| (Total States) * (Work per State) * (Cost of string match)  |
|       (N)      * (M)        * (L)            |
|                                                             |
| FINAL TIME COMPLEXITY: O(N * M * L)                         |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
| SPACE COMPLEXITY DERIVATION                                 |
+-------------------------------------------------------------+
| [1] Call Stack (Top-Down) OR Array Size (Bottom-Up)         |
|     Maximum recursion depth or array size is N + 1          |
|     Space = O(N)                                            |
|                                                             |
| [2] Memoization Cache (Top-Down only)                       |
|     Stores a boolean for up to N unique indices             |
|     Space = O(N)                                            |
|                                                             |
| FINAL SPACE COMPLEXITY: O(N)                                |
+-------------------------------------------------------------+
```

(Note: An alternative approach involves iterating `j` from `i` to `N` and checking a Hash Set for the substring. That yields `O(N^3)` time complexity due to substring hashing and extraction. The `O(N * M * L)` approach provided below is often preferred by senior engineers because `M * L` is usually much smaller than `N^2` in real-world constraints).

---

4. Solution Code

Python 3 Solutions

```python
# --- APPROACH 1: TOP-DOWN (Memoization) ---
def wordBreakTopDown(s: str, wordDict: list[str]) -> bool:
    # Memoization dictionary to store the result for each starting index 'i'
    memo = {}

    # Inner recursive function to keep the namespace clean
    def dfs(i: int) -> bool:
```

```
        # Base Case: We've successfully reached the end of the string
        if i == len(s):
            return True

        # If we've calculated this state before, return the cached result
        if i in memo:
            return memo[i]

        # Try matching every dictionary word starting from index 'i'
        for word in wordDict:
            # Check if 's' starting at 'i' matches the word
            if s[i:].startswith(word):
                # Move the pointer forward by the length of the word matched
                if dfs(i + len(word)):
                    memo[i] = True
                    return True

        # If no words can lead to a valid segmentation from this index, cache and return Fa
        memo[i] = False
        return False

    return dfs(0)


# --- APPROACH 2: BOTTOM-UP (Tabulation) ---
def wordBreakBottomUp(s: str, wordDict: list[str]) -> bool:
    # dp[i] represents whether s[0...i-1] can be segmented
    dp = [False] * (len(s) + 1)

    # Base case: Empty string is always a valid segmentation (requires 0 words)
    dp[0] = True

    for i in range(len(s)):
        # Only process if we have a valid word sequence ending at index 'i'
        if dp[i]:
            for word in wordDict:
                word_len = len(word)
                # Check if the word fits within remaining bounds AND matches the substring
                if i + word_len <= len(s) and s[i : i + word_len] == word:
                    # Mark the end index of the newly matched word as True
                    dp[i + word_len] = True

    # Return whether we could form a sequence that reaches the very end of the string
    return dp[len(s)]
```

JavaScript Solutions

```javascript
// --- APPROACH 1: TOP-DOWN (Memoization) ---
function wordBreakTopDown(s, wordDict) {
    // Cache to store results. Size is s.length to map each index to a boolean
    const memo = new Map();

    function dfs(i) {
        // Base Case: successfully reached the end of the string
        if (i === s.length) {
            return true;
        }

        // Return cached result if we've processed this index before
        if (memo.has(i)) {
            return memo.get(i);
        }

        // Try every word in the dictionary at the current index 'i'
        for (const word of wordDict) {
            // Check if the substring of 's' starting at 'i' begins with 'word'
            if (s.startsWith(word, i)) {
                // If the recursive call for the rest of the string returns true, cache and
                if (dfs(i + word.length)) {
                    memo.set(i, true);
                    return true;
                }
            }
        }

        // No path led to a valid segmentation
        memo.set(i, false);
        return false;
    }

    return dfs(0);
}


// --- APPROACH 2: BOTTOM-UP (Tabulation) ---
function wordBreakBottomUp(s, wordDict) {
    // Initialize DP array of size n + 1 with false
    const dp = new Array(s.length + 1).fill(false);

    // Base case: empty string
    dp[0] = true;
```

```
    for (let i = 0; i < s.length; i++) {
        // Only branch out if 'i' is reachable via valid dictionary words
        if (dp[i]) {
            for (const word of wordDict) {
                // Check if the substring starting at 'i' matches the current word
                // s.startsWith(searchString, position) is highly efficient
                if (s.startsWith(word, i)) {
                    // Mark the index immediately after the matched word as reachable
                    dp[i + word.length] = true;
                }
            }
        }
    }

    // The final index holds the truth value for the entire string
    return dp[s.length];
}
```

Technical Glossary

- **Dynamic Programming (DP):** An algorithmic paradigm that solves a complex problem by breaking it down into smaller, overlapping subproblems. It solves each subproblem once and stores the result to avoid redundant work. It applies here because verifying the second half of a string is the same logic as verifying the whole string.
- **Memoization:** The "Top-Down" DP technique. It involves writing a standard recursive function but adding a cache (like a map or array) to save the answers to specific function inputs. If the function is called again with the same input, it returns the cached answer instead of recalculating.
- **Tabulation:** The "Bottom-Up" DP technique. It avoids recursion entirely. Instead, it systematically fills a table (usually an array) starting from the smallest, simplest base cases (like an empty string) and builds up to the final answer iteratively.

Real-World / Interview Variations

Senior engineers know that FAANG companies rarely ask the raw LeetCode question. They dress it up. Here is how you will likely see this concept disguised:

1. **Google - "Search Query Segmentation":** You are building a search engine feature. Users often type queries without spaces on mobile (e.g., `cheapflightsnewyork`). Given a dictionary of valid English words and known entities, write an algorithm to determine if the query can be perfectly split into legitimate words. *(Direct 1:1 mapping to Word Break).*

2. **Meta - "Hashtag Parser":** On Instagram, users use massive combined hashtags (e.g., `#alwaysbelearning`). To power the recommendation algorithm, we need to extract the root concepts. Can this hashtag string be broken down into a valid list of dictionary words?
3. **Bloomberg - "Ticker Decoder":** A stream of financial data arrived corrupted, stripping all spaces between stock tickers (e.g., `AAPLMSFTGOOG`). Given a valid list of active exchange tickers, verify if the incoming string is a completely valid sequence of known tickers or if it contains garbage data.

377. Combination Sum IV

To solve "Combination Sum IV," a Google L5/L6 engineer doesn't just look for "a" solution; they look for the **optimal** solution by identifying the core pattern. This is a classic **Dynamic Programming (DP)** problem, specifically a variation of the "Unbounded Knapsack" or "Coin Change" problem where **order matters**.

---

1. Problem Explanation

Given an array of **distinct** integers `nums` and a `target` integer, return the number of possible combinations that add up to `target`.

The Key Detail: Different sequences are counted as unique combinations. For example, if `nums = [1, 2, 3]` and `target = 4`:

- `(1, 1, 2)` is one way.
- `(1, 2, 1)` is another way.
- `(2, 1, 1)` is a third way.

Because $(1, 2, 1)$ and $(1, 1, 2)$ are distinct, we are actually looking for **permutations** that sum to the target, not just combinations in the mathematical sense.

---

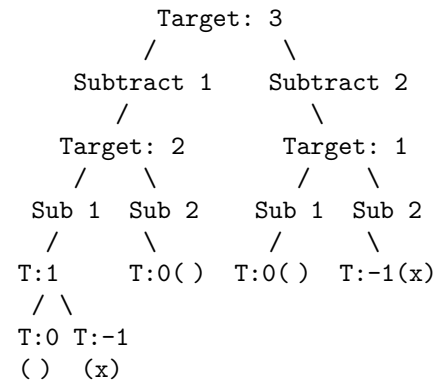
2. Solution Explanation

The Intuitive Start: Top-Down (Recursive with Memoization)

Imagine you are at the `target` value and you want to get to `0`. You can take any number from `nums` and subtract it from your current value.

Example: `nums = [1, 2]`, `target = 3`

Step 1: The Decision Tree



Non-trivial point: Notice how `Target: 1` appears multiple times? Without **Memoization**, we would recalculate the result for `1` every single time. An L5 engineer knows that caching the result of `solve(1)` turns an exponential time complexity into a linear one relative to the target.

Transitioning to Bottom-Up (Iterative DP)

While Top-Down is intuitive, Bottom-Up is often preferred in production for avoiding stack overflow on large targets. We build a table `dp` where `dp[i]` is the number of ways to reach the sum `i`.

The Logic: To find `dp[4]`, if our numbers are `[1, 2, 3]`:

- Ways ending in 1: `dp[4 - 1]` (ways to make 3)
- Ways ending in 2: `dp[4 - 2]` (ways to make 2)
- Ways ending in 3: `dp[4 - 3]` (ways to make 1)
- **Total:** `dp[4] = dp[3] + dp[2] + dp[1]`

Step-by-Step Visualization (`nums = [1, 2]`, `target = 3`):

```
Initialize DP table of size Target + 1 (size 4):
Index:  0   1   2   3
Value: [1,  0,  0,  0]  <-- dp[0] = 1 (There's 1 way to make 0: use nothing)

Iteration i = 1:
  Can we use nums[0]=1? Yes (1 >= 1). dp[1] += dp[1-1] -> dp[1] = 1
  Can we use nums[1]=2? No (1 < 2).
  Table: [1, 1, 0, 0]

Iteration i = 2:
  Can we use nums[0]=1? Yes. dp[2] += dp[2-1] -> dp[2] = 0 + 1 = 1
  Can we use nums[1]=2? Yes. dp[2] += dp[2-2] -> dp[2] = 1 + 1 = 2
  Table: [1, 1, 2, 0]

Iteration i = 3:
```

```
  Can we use nums[0]=1? Yes. dp[3] += dp[3-1] -> dp[3] = 0 + 2 = 2
  Can we use nums[1]=2? Yes. dp[3] += dp[3-2] -> dp[3] = 2 + 1 = 3
  Table: [1, 1, 2, 3]

Result: dp[3] = 3.
```

---

3. Time and Space Complexity Analysis

We define `T` as the target value and `N` as the number of elements in `nums`.

Time Complexity (TC)

```
For every value from 1 to T:
   Iterate through every number in nums (N elements)
   Perform a constant time addition: dp[i] += dp[i - num]

Visual Derivation:
[ 1 ] -> loops N times
[ 2 ] -> loops N times
...
[ T ] -> loops N times

Total Operations = T * N
TC = O(T * N)
```

Space Complexity (SC)

```
We create a DP array (or a Memo hashmap) to store results
for every integer from 0 up to T.

Visual Derivation:
Memory: [ 0, 1, 2, ... , T ]
Size of Array: T + 1

SC = O(T)
```

---

4. Solution Code

Python (Top-Down & Bottom-Up)

```python
class Solution:
    # TOP-DOWN APPROACH (Recursive + Memoization)
    # This uses a cache to store results of subproblems to avoid
    # redundant work in the recursion tree.
```

```python
    def combinationSum4_TopDown(self, nums: list[int], target: int) -> int:
        memo = {}

        def dfs(remain):
            if remain == 0: return 1
            if remain < 0: return 0
            if remain in memo: return memo[remain]

            count = 0
            for num in nums:
                count += dfs(remain - num)

            memo[remain] = count
            return count

        return dfs(target)

    # BOTTOM-UP APPROACH (Iterative)
    # This fills a table from 0 up to the target, ensuring every
    # dependency is calculated before it's needed.
    def combinationSum4_BottomUp(self, nums: list[int], target: int) -> int:
        dp = [0] * (target + 1)
        dp[0] = 1 # Base case: 1 way to make sum 0 (empty set)

        for i in range(1, target + 1):
            for num in nums:
                if i - num >= 0:
                    dp[i] += dp[i - num]

        return dp[target]
```

JavaScript (Top-Down & Bottom-Up)

```javascript
/**
 * TOP-DOWN APPROACH
 * Uses a Map for memoization. Effective for large, sparse target ranges.
 */
function combinationSum4_TopDown(nums, target) {
    const memo = new Map();

    function dp(remain) {
        if (remain === 0) return 1;
        if (remain < 0) return 0;
        if (memo.has(remain)) return memo.get(remain);

        let count = 0;
```

```
        for (let num of nums) {
            count += dp(remain - num);
        }

        memo.set(remain, count);
        return count;
    }

    return dp(target);
}

/**
 * BOTTOM-UP APPROACH
 * Uses a TypedArray for performance. This is the standard "L6" production
 * style for simple DP problems due to cache locality.
 */
function combinationSum4_BottomUp(nums, target) {
    const dp = new Array(target + 1).fill(0);
    dp[0] = 1;

    for (let i = 1; i <= target; i++) {
        for (let num of nums) {
            if (i - num >= 0) {
                dp[i] += dp[i - num];
            }
        }
    }

    return dp[target];
}
```

Terminology & Real-World Context

Dynamic Programming (DP): A method for solving complex problems by breaking them down into simpler subproblems. It applies here because the number of ways to make `10` depends on the number of ways to make `9`, `8`, etc.

Real-World / Interview Variations: Google and Meta often disguise this problem in real-world scenarios:

1. **Step Climbing:** "You can climb 1, 2, or 3 steps at a time. How many ways to reach the top?" (This is exactly Combination Sum IV with `nums = [1, 2, 3]`).
2. **API Rate Limiting:** "Given various request 'weights', how many sequences of requests can fill a buffer of size X?"

3. **Keyboard Layouts:** Bloomberg has asked variations regarding the number of ways to type a string given certain key-combination rules.
4. **E-commerce:** "How many ways can a customer combine different discount vouchers to reach exactly $100 off?"

198. House Robber

A Google L5 or L6 engineer doesn't just look for a solution; they look for the **optimal** solution while considering trade-offs, edge cases, and code maintainability. For "House Robber," they would immediately recognize this as a **Dynamic Programming (DP)** problem because it involves making a sequence of dependent choices to maximize a value.

---

1. Problem Explanation

Imagine you are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed.

The Catch: All houses are connected in a security system. If two adjacent houses are broken into on the same night, the system will automatically contact the police.

The Goal: Maximize the total money you can rob without alerting the police.

Example:

`Houses = [2, 7, 9, 3, 1]`

- If you rob house 0 (2) and house 2 (9) and house 4 (1), total = 2 + 9 + 1 = **12**.
- If you rob house 1 (7) and house 3 (3), total = 7 + 3 = **10**.
- The maximum is **12**.

---

2. Solution Explanation

The Intuition: The "To Rob or Not To Rob" Decision

At every house `i`, you have two choices:

1. **Rob House `i`**: **You get the money in `House[i]`, but you** must** have skipped `House[i-1]`. Therefore, your total is `House[i] + Max Profit up to House[i-2]`.
2. **Skip House `i`**: You don't get the money in `House[i]`. Your total is simply the `Max Profit up to House[i-1]`.

The formula is: `Max_at_i = Max(Rob_i, Skip_i)` `Max_at_i = Max(House[i] + Max_at_i-2, Max_at_i-1)`

Step-by-Step Visualization (Bottom-Up)

Let's use the example: `[1, 2, 3, 1]`

Initial State:

```
Houses:    [ 1,  2,  3,  1 ]
Indices:     0   1   2   3
DP Table:  [ 0,  0,  0,  0 ]  <-- To store max profit at each step
```

Step 1: House 0 (Value 1) You rob it. Max profit is 1.

`DP: [ 1, 0, 0, 0 ]`

Step 2: House 1 (Value 2) Option A: Rob House 1 (2) + Max at House -1 (0) = 2 Option B: Skip House 1 (Max at House 0) = 1 Max(2, 1) = 2.

`DP: [ 1, 2, 0, 0 ]`

Step 3: House 2 (Value 3) Option A: Rob House 2 (3) + Max at House 0 (1) = 4 Option B: Skip House 2 (Max at House 1) = 2 Max(4, 2) = 4.

`DP: [ 1, 2, 4, 0 ]`

Step 4: House 3 (Value 1) Option A: Rob House 3 (1) + Max at House 1 (2) = 3 Option B: Skip House 3 (Max at House 2) = 4 Max(3, 4) = 4.

`DP: [ 1, 2, 4, 4 ]`

Final Result: 4

Converting to Space-Optimized Approach

Notice that to calculate `DP[i]`, we only ever need `DP[i-1]` and `DP[i-2]`. We don't need the whole array! An L6 engineer would optimize the space from O(N) to O(1).

```
Iterating through [1, 2, 3, 1]:

Start:   rob1 = 0, rob2 = 0
House 1: temp = max(1 + 0, 0) = 1.  Update: rob1 = 0, rob2 = 1
House 2: temp = max(2 + 0, 1) = 2.  Update: rob1 = 1, rob2 = 2
House 3: temp = max(3 + 1, 2) = 4.  Update: rob1 = 2, rob2 = 4
House 1: temp = max(1 + 2, 4) = 4.  Update: rob1 = 4, rob2 = 4

Result: rob2 (which is 4)
```

---

3. Complexity Analysis

Time Complexity (TC)

We iterate through the list of houses exactly once. Every operation inside the loop (addition and max comparison) is constant time.

```
House 1 -> House 2 -> ... -> House N
|___________________________________|
              N operations

TC = O(N)
```

Space Complexity (SC)

- **Top-Down/Bottom-Up with Array:** We store a result for every house. `SC = O(N)`
- **Optimized Bottom-Up:** We only store two variables (`rob1`, `rob2`). `SC = O(1)`

```
Space used:
[ rob1 ] [ rob2 ]  <-- Only 2 memory slots regardless of N
SC = O(1)
```

---

4. Solution Code

JavaScript

```javascript
/**
 * BOTTOM-UP (Space Optimized)
 * This is the gold standard for an interview.
 * We use two variables to track the previous two maximums.
 */
function robBottomUp(nums) {
    let rob1 = 0; // Max profit if we stop 2 houses ago
    let rob2 = 0; // Max profit if we stop 1 house ago

    for (let n of nums) {
        // temp represents the max profit if we consider the current house 'n'
        let temp = Math.max(n + rob1, rob2);
        rob1 = rob2;
        rob2 = temp;
    }
    return rob2;
}
```

```javascript
/**
 * TOP-DOWN (Memoization)
 * We use a recursive approach but save results in a 'memo' object
 * to avoid recalculating the same subproblems.
 */
function robTopDown(nums) {
    let memo = {};

    function studyHouse(i) {
        if (i < 0) return 0;
        if (i in memo) return memo[i];

        // Choice: Rob current house + skip previous OR Skip current house
        memo[i] = Math.max(studyHouse(i - 2) + nums[i], studyHouse(i - 1));
        return memo[i];
    }

    return studyHouse(nums.length - 1);
}
```

Python

```python
class Solution:
    def rob_bottom_up(self, nums: list[int]) -> int:
        # rob1, rob2 = [rob1, rob2, n, n+1, ...]
        rob1, rob2 = 0, 0

        for n in nums:
            # We decide: include current house + rob1 OR just keep rob2
            temp = max(n + rob1, rob2)
            rob1 = rob2
            rob2 = temp
        return rob2

    def rob_top_down(self, nums: list[int]) -> int:
        memo = {}

        def dfs(i):
            # Base case: no more houses to rob
            if i >= len(nums):
                return 0
            if i in memo:
                return memo[i]

            # Decision: Rob this house (move to i+2) or skip (move to i+1)
```

```
            res = max(dfs(i + 1), nums[i] + dfs(i + 2))
            memo[i] = res
            return res

        return dfs(0)
```

Note 1: Terms and Techniques

- **Dynamic Programming (DP):** An algorithmic technique that breaks a complex problem into smaller overlapping subproblems and stores the results to avoid redundant work.
- **Memoization (Top-Down):** Storing results of expensive function calls (usually in a Hash Map or Array) to return them when the same inputs occur again.
- **Tabulation (Bottom-Up):** Filling a table (array) iteratively starting from the simplest case up to the final answer.

Note 2: Real-World Interview Variations

Google and Meta often use "House Robber" as a base and add constraints to test an L5/L6's ability to adapt:

1. **House Robber II (Circular):** The houses are in a circle (the first and last are neighbors). *Strategy: Run the algorithm twice (once skipping the first house, once skipping the last).*
2. **House Robber III (Tree):** The houses are arranged in a Binary Tree (cannot rob parent and child). *Strategy: Use Depth First Search (DFS).*
3. **Maintenance Scheduling (Meta):** Scheduling server maintenance windows where you cannot pick consecutive time slots without causing downtime.
4. **Resource Allocation (Bloomberg):** Picking projects to fund where certain high-value projects are "mutually exclusive" if they share a specific timeline.

213. House Robber II

For a top-of-the-band L5 or L6 engineer at Google, the goal isn't just to "solve" the problem, but to identify the core mathematical pattern, reduce it to a known primitive, and handle edge cases with clean, modular code.

1. Problem Explanation

The problem asks us to find the maximum amount of money we can rob from a **circular** row of houses.

The Constraints:

- Each house has a certain amount of money: `nums[i]`.
- **Constraint 1:** You cannot rob two adjacent houses (the alarm will go off).
- **Constraint 2 (The Twist):** The houses are arranged in a circle. This means the **first** house and the **last** house are neighbors.

The "Aha!" Moment: In the original "House Robber I" (linear), the choice at the last house didn't affect the first. Here, if you rob House 0, you **cannot** rob House N-1.

Visualizing the Conflict:

```
Linear (Easy): [ H0 ] - [ H1 ] - [ H2 ] - [ H3 ]
Circle (Hard):
      [ H0 ] -- [ H1 ]
        |         |
      [ H3 ] -- [ H2 ]
```

If you pick H0, H1 and H3 are blocked.

---

2. Solution Explanation

An L6 engineer would immediately see this as a **Case Decomposition** problem. We can break the circular constraint into two linear sub-problems:

- **Scenario A:** You consider robbing houses from the **first to the second-to-last**. (Ignore the last house).
- **Scenario B:** You consider robbing houses from the **second to the last**. (Ignore the first house).

The answer is simply `max(Scenario A, Scenario B)`.

The Dynamic Programming Intuition (Bottom-Up) Let's look at the linear version first. For any house `i`, you have two choices:

1. **Rob it:** You get `nums[i]` + whatever you robbed up to `i-2`.
2. **Skip it:** You keep whatever you robbed up to `i-1`.

State Transition: `dp[i] = max(dp[i-1], nums[i] + dp[i-2])`

ASCII Step-by-Step Visualization Example: `nums = [2, 3, 2]`

Step 1: Break into two linear arrays

```
Sub-problem 1 (Exclude last): [2, 3]
Sub-problem 2 (Exclude first): [3, 2]
```

Step 2: Solve Sub-problem 1 `[2, 3]`

```
Index:      0      1
Value:    [ 2 ]  [ 3 ]

DP State:
i=0: rob 2. Max = 2
i=1: max(skip 3 [prev=2], rob 3 [prev-prev=0]). Max = 3
Result 1 = 3
```

Step 3: Solve Sub-problem 2 `[3, 2]`

```
Index:      0      1
Value:    [ 3 ]  [ 2 ]

DP State:
i=0: rob 3. Max = 3
i=1: max(skip 2 [prev=3], rob 2 [prev-prev=0]). Max = 3
Result 2 = 3
```

Step 4: Take the Global Max `max(3, 3) = 3`

Space Optimization (The L5/L6 way) We don't need an entire `dp` array. We only ever need the last two values.

```
Before iteration:  [prev2] -> [prev1] -> [current]
After iteration:             [prev2] -> [prev1]
```

---

3. Complexity Analysis

We iterate through the array twice (once for each scenario).

Time Complexity (TC):

```
Scenario A: O(N)
Scenario B: O(N)
Total: O(N) + O(N) = O(N)
```

Space Complexity (SC):

```
Bottom-Up (Optimized): O(1) -- only two variables used.
Top-Down (Memoization): O(N) -- due to recursion stack and hash map.
```

---

4. Solution Code

Top-Down (Recursive + Memoization) This is often more intuitive for complex DP, but uses more memory due to the recursion stack.

Python

```python
class Solution:
    def rob(self, nums: list[int]) -> int:
        if len(nums) == 1: return nums[0]

        # Helper to solve linear house robber using memoization
        def solve_linear(arr):
            memo = {}
            def dp(i):
                if i < 0: return 0
                if i in memo: return memo[i]
                # Option 1: Skip current, Option 2: Rob current + jump 1
                memo[i] = max(dp(i - 1), arr[i] + dp(i - 2))
                return memo[i]
            return dp(len(arr) - 1)

        return max(solve_linear(nums[:-1]), solve_linear(nums[1:]))
```

JavaScript

```javascript
var rob = function(nums) {
    if (nums.length === 1) return nums[0];

    /**
     * Helper function: Solves the linear version of the problem
     * using a memoization object to store sub-problem results.
     */
    const solveLinear = (arr) => {
        const memo = new Map();
        const dp = (i) => {
            if (i < 0) return 0;
            if (memo.has(i)) return memo.get(i);

            const res = Math.max(dp(i - 1), arr[i] + dp(i - 2));
            memo.set(i, res);
            return res;
        };
        return dp(arr.length - 1);
    };

    return Math.max(solveLinear(nums.slice(0, -1)), solveLinear(nums.slice(1)));
};
```

Bottom-Up (Iterative + Space Optimized) This is the production-grade approach favored in high-level interviews.

Python

```python
class Solution:
    def rob(self, nums: list[int]) -> int:
        if len(nums) == 1: return nums[0]

        # Optimized linear robber: Only tracks the last two maximums
        def solve_linear(houses):
            prev2, prev1 = 0, 0
            for amount in houses:
                # current_max = max(skip_current, rob_current)
                new_rob = max(prev1, amount + prev2)
                prev2 = prev1
                prev1 = new_rob
            return prev1

        # Case 1: Exclude the last house
        # Case 2: Exclude the first house
        return max(solve_linear(nums[:-1]), solve_linear(nums[1:]))
```

JavaScript

```javascript
var rob = function(nums) {
    if (nums.length === 1) return nums[0];

    /**
     * Iterative approach with O(1) space.
     * prev2 represents dp[i-2], prev1 represents dp[i-1].
     */
    const solveLinear = (houses) => {
        let prev2 = 0;
        let prev1 = 0;
        for (const amount of houses) {
            let temp = prev1;
            prev1 = Math.max(prev1, amount + prev2);
            prev2 = temp;
        }
        return prev1;
    };

    return Math.max(
        solveLinear(nums.slice(0, nums.length - 1)),
        solveLinear(nums.slice(1))
    );
};
```

Note 1: Terms and Techniques

- **Case Decomposition:** Breaking a problem with a complex constraint (the circle) into two simpler problems (linear rows) that cover all possible optimal solutions. This helps because "Circular DP" is often just "Linear DP" run twice with different boundaries.
- **State Compression:** Moving from an `O(N)` space array to `O(1)` variables. Since `dp[i]` only depends on `dp[i-1]` and `dp[i-2]`, we discard older data to save memory.

Note 2: Real-World Interview Variations

Companies like Google and Bloomberg rarely ask the "House Robber" story directly anymore. Instead, they use these patterns:

1. **Google (Infrastructure):** "You have a ring of data centers. You need to perform maintenance. You cannot shut down two adjacent centers. What is the maximum capacity you can maintain?"
2. **Meta (Product):** "Users are arranged in a circular social graph. You want to show ads but don't want to show them to neighbors to avoid fatigue. Each user has a 'receptivity' score. Maximize total score."
3. **Bloomberg (Finance):** "A circular set of trading windows. Each has a profit. You can't trade in consecutive windows due to cooling-off regulations. Maximize profit."

91. Decode Ways

An L5/L6 engineer doesn't just look for a solution; they look for **edge cases**, **state definitions**, and **memory efficiency**. For "Decode Ways," the challenge isn't just the logic—it's handling the "0" character, which acts as a "blocker" in the decoding stream.

---

1. Problem Explanation

You are given a string of digits. Each number maps to a letter (1 = A, 2 = B … 26 = Z). You need to find the total number of ways to decode the string.

The Crucial Rules:

- **Single Digit:** Any digit from '1' to '9' can be a letter.
- **Double Digit:** Any two digits from "10" to "26" can be a letter.
- **The Zero Trap:** '0' cannot stand alone (there is no 0th letter). It must be preceded by a '1' or '2' (to make 10 or 20). If it appears as '05' or '30', that path is invalid.

---

2. Solution Explanation

The Intuition (Top-Down)

Think of this like a decision tree. At any index `i`, you have two choices:

1. Take **one** digit (if it's not '0').
2. Take **two** digits (if they form a number between 10 and 26).

Visualizing the Decision Tree (Input: "226")

```
[Index 0: "226"]
       /        \
   "2"           "22"
  (B)            (V)
   |              |
[Index 1: "26"]  [Index 2: "6"]
 /      \         |
"2"     "26"     "6"
(B)     (Z)      (F)
 |       |        |
[Idx 2] [Done]   [Done]  <-- 3 Ways Found
 |
"6"
(F)
 |
[Done]
```

The State Transition (Bottom-Up)

Top-down is intuitive, but a Senior Engineer often prefers Bottom-Up (Iterative DP) to avoid stack overflow and clearly see the "space optimization" potential.

We define `dp[i]` as: **"How many ways can we decode the string starting from index i to the end?"**

Step-by-Step Visualization (Input: "226")

We initialize a DP array of size `n + 1`. `dp[3] = 1` (Base case: an empty string has 1 way to be "finished").

Step 1: Process '6' (Index 2)

- '6' is not '0', so it can stand alone.
- `dp[2] = dp[3] = 1`.

```
String:  2  2  6  _
DP:     [?][?][1][1]
               ^  ^-- Base case
               |-- Ways to decode "6"
```

Step 2: Process '2' (Index 1)

- Single digit '2' is valid: add `dp[2]`.
- Double digit '26' is valid (<= 26): add `dp[3]`.
- `dp[1] = dp[2] + dp[3] = 1 + 1 = 2`.

```
String:  2  2  6  _
DP:     [?][2][1][1]
            ^
            |-- Ways to decode "26" (either "2,6" or "26")
```

Step 3: Process '2' (Index 0)

- Single digit '2' is valid: add `dp[1]`.
- Double digit '22' is valid: add `dp[2]`.
- `dp[0] = dp[1] + dp[2] = 2 + 1 = 3`.

```
String:  2  2  6  _
DP:     [3][2][1][1]
         ^
         |-- Final Answer
```

---

3. Time and Space Complexity Analysis

As requested, here is the derivation in ASCII format:

```
TIME COMPLEXITY:
We iterate through the string of length N exactly once.
Inside each iteration, we perform constant time O(1) checks
(checking if a character is '0' or if two digits are <= 26).

Total Time = N * O(1) = O(N)

SPACE COMPLEXITY (Standard DP):
We use an array (DP table) of size N + 1 to store subproblem results.

Total Space = O(N)

SPACE COMPLEXITY (Optimized):
Notice that dp[i] only relies on dp[i+1] and dp[i+2].
We can just use two variables (prev1, prev2) to store these.

Total Space = O(1)
```

---

4. Solution Code

Python (Top-Down & Bottom-Up)

```python
class Solution:
    # TOP-DOWN (Memoization)
    # This approach mimics the recursive decision tree, storing results
    # in a 'memo' dictionary to avoid redundant calculations.
    def numDecodingsTopDown(self, s: str) -> int:
        memo = {}

        def recursive_helper(index):
            if index == len(s):
                return 1
            if s[index] == '0':
                return 0
            if index in memo:
                return memo[index]

            # Option 1: Take one digit
            res = recursive_helper(index + 1)

            # Option 2: Take two digits (check bounds and range 10-26)
            if index + 1 < len(s) and (s[index] == '1' or (s[index] == '2' and s[index+1] in
                res += recursive_helper(index + 2)

            memo[index] = res
            return res

        return recursive_helper(0)

    # BOTTOM-UP (Tabulation - Space Optimized)
    # We only need the last two calculated values to find the current one.
    def numDecodingsBottomUp(self, s: str) -> int:
        if not s or s[0] == '0':
            return 0

        n = len(s)
        # prev1 represents dp[i-1], prev2 represents dp[i-2]
        prev2 = 1 # Empty string
        prev1 = 1 # First character (already checked for '0')

        for i in range(2, n + 1):
            current = 0
            # Check single digit
            if s[i-1] != '0':
```

```
                current += prev1

            # Check double digit
            two_digit = int(s[i-2:i])
            if 10 <= two_digit <= 26:
                current += prev2

            prev2 = prev1
            prev1 = current

        return prev1
```

JavaScript (Top-Down & Bottom-Up)

```
/**
 * TOP-DOWN Approach
 * Uses a 'memo' array to store the number of ways to decode from index 'i'.
 */
var numDecodingsTopDown = function(s) {
    const memo = new Array(s.length).fill(-1);

    function solve(idx) {
        if (idx === s.length) return 1;
        if (s[idx] === '0') return 0;
        if (memo[idx] !== -1) return memo[idx];

        // One digit
        let ways = solve(idx + 1);

        // Two digits
        if (idx + 1 < s.length) {
            let twoDigit = parseInt(s.substring(idx, idx + 2));
            if (twoDigit >= 10 && twoDigit <= 26) {
                ways += solve(idx + 2);
            }
        }

        return memo[idx] = ways;
    }
    return solve(0);
};

/**
 * BOTTOM-UP Approach
 * Iteratively builds the solution. Space is O(1) as we only track
 * the two previous states.
```

```
 */
var numDecodingsBottomUp = function(s) {
    if (s[0] === '0') return 0;

    let n = s.length;
    let prev2 = 1; // dp[i-2]
    let prev1 = 1; // dp[i-1]

    for (let i = 2; i <= n; i++) {
        let current = 0;

        // One digit check
        if (s[i-1] !== '0') {
            current += prev1;
        }

        // Two digit check
        let twoDigit = parseInt(s.substring(i-2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            current += prev2;
        }

        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
};
```

Terminology Note: Dynamic Programming (DP)

DP is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable here because the number of ways to decode a long string depends directly on the ways we decoded the shorter substrings before it.

- **Top-Down** starts at the goal and looks backward (using recursion).
- **Bottom-Up** starts at the smallest pieces and builds up to the goal (using loops).

Real-World Interview Variants

Google and Meta often wrap this logic in "System Design" or "String Processing" contexts:

1. **Bloomberg:** "Stock Ticker Decoder" – Similar logic where characters represent specific stock codes.
2. **Google:** "Valid IP Address Combinations" – Instead of 1-26, the range is 0-255.
3. **Meta:** "Identify all possible English words in a concatenated string" (Word Break variation).

62. Unique Paths

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just "solve" this problem; they decompose it. They look for the recursive structure, identify the overlapping subproblems, and then optimize for both performance and memory.

This is a classic **Dynamic Programming (DP)** problem. I will break it down the way it's handled in high-level technical interviews.

---

1. Problem Explanation

Imagine a robot on a 2D grid of size `m` (rows) by `n` (cols).

- **Start:** Top-left corner `(0, 0)`
- **Goal:** Bottom-right corner `(m-1, n-1)`
- **Movement:** The robot can **only** move **Down** or **Right**.

We need to find the total number of unique paths to reach the goal.

The Insight: If you are at a specific cell `(r, c)`, how did you get there? Since you can only move Right and Down, you must have come from either:

1. The cell directly above it: `(r-1, c)`
2. The cell directly to the left: `(r, c-1)`

Therefore: `Paths to(r, c) = Paths to(r-1, c) + Paths to(r, c-1)`

---

2. Solution Explanation

The Intuitive Start: Top-Down (Recursive with Memoization)

For most engineers, the most intuitive way to think is "Backwards from the Goal." To know how many ways there are to reach the finish line, I need to know the ways to reach the two cells that lead into it.

ASCII Visualization: The Recursive Tree (3x3 Grid) To go from (0,0) to (2,2):

```
Target: (2,2)
                                (2,2)
                               /     \
                          (1,2)           (2,1)  <-- Sub-targets
                         /     \         /     \
                    (0,2)     (1,1)    (1,1)    (2,0)
                    /  \      /   \    /   \     /  \
                Base (0,1) (0,1)(1,0)(0,1)(1,0)(1,0) Base
```

The Problem: Notice `(1,1)` and `(0,1)` are calculated multiple times. This is "Overlapping Subproblems." We use a **Memo** (a dictionary or 2D array) to store results we've already found.

The Efficient Shift: Bottom-Up (Tabulation)

An L6 engineer often prefers Bottom-Up because it avoids stack overflow risks and makes "Space Optimization" obvious. We fill a table from the start to the finish.

Step-by-Step ASCII Construction (3x7 Grid) Step 1: Initialize the grid. The top row and the first column must all be `1`. Why? Because there is only one way to reach any cell in the first row (keep going right) and only one way to reach any cell in the first column (keep going down).

```
    c0  c1  c2  c3  c4  c5  c6
r0 [1] [1] [1] [1] [1] [1] [1]  <- Only one way to stay on the edge
r1 [1] [ ] [ ] [ ] [ ] [ ] [ ]
r2 [1] [ ] [ ] [ ] [ ] [ ] [ ]
```

Step 2: Fill cell (1,1). `Ways to (1,1) = Left (1,0) + Above (0,1) -> 1 + 1 = 2`

```
    c0  c1  c2  ...
r0 [1] [1] [1]
r1 [1] [2] [ ]  <-- (1+1)
r2 [1] [ ] [ ]
```

Step 3: Fill the rest of Row 1. Each cell is the sum of its left and top neighbors.

```
    c0  c1  c2  c3  c4  c5  c6
r0 [1] [1] [1] [1] [1] [1] [1]
r1 [1] [2] [3] [4] [5] [6] [7]  <-- 1+2=3, 1+3=4, etc.
r2 [1] [ ] [ ] [ ] [ ] [ ] [ ]
```

Step 4: Fill Row 2. `Cell (2,1) = Left (2,0) + Above (1,1) -> 1 + 2 = 3 Cell (2,2) = Left (2,1) + Above (1,2) -> 3 + 3 = 6`

```
    c0  c1  c2  c3  c4  c5  c6
r0 [1] [1] [1] [1] [1] [1] [1]
r1 [1] [2] [3] [4] [5] [6] [7]
r2 [1] [3] [6] [10][15][21][28] <-- Final Answer is 28
```

---

3. Time and Space Complexity Analysis

Time Complexity (TC)

We visit every cell in the `m x n` grid exactly once. Inside each cell, we perform a simple addition (O(1)).

```
TC Derivation:
Rows (m) * Columns (n) * Constant Work (1)
= O(m * n)
```

Space Complexity (SC)

In the standard Bottom-Up approach, we create a 2D grid.

```
SC Derivation (Standard):
Rows (m) * Columns (n) space for the DP table
= O(m * n)
```

L6 Optimization (Space): Notice that to calculate a new row, we only ever need the **current row's previous value** and the **previous row's value** at that same column. We can reduce the grid to just one single array of size `n`.

```
SC Derivation (Optimized):
Only store one row of size n
= O(n)
```

---

4. Solution Code

Python

```python
# TOP-DOWN APPROACH (Recursive with Memoization)
def uniquePaths_TopDown(m, n):
    memo = {}

    # This function explores paths from (r, c) to (0, 0)
    def solve(r, c):
        # Base Case: If we are on the first row or first column, only 1 path exists
        if r == 0 or c == 0:
            return 1
```

```
        # Check memo to see if we solved this sub-grid already
        if (r, c) in memo:
            return memo[(r, c)]

        # Recursive Step: Sum of paths from Top and Left
        memo[(r, c)] = solve(r - 1, c) + solve(r, c - 1)
        return memo[(r, c)]

    return solve(m - 1, n - 1)


# BOTTOM-UP APPROACH (Tabulation with Space Optimization)
def uniquePaths_BottomUp(m, n):
    # We only need the previous row to calculate the current row
    # Initialize a row of size n with all 1s (representing the top row)
    row = [1] * n

    # Iterate through each row (starting from the second row)
    for i in range(m - 1):
        new_row = [1] * n
        # Iterate through each column (starting from the second column)
        for j in range(1, n):
            # The value is the sum of the value to the left (new_row[j-1])
            # and the value from the row above (row[j])
            new_row[j] = new_row[j - 1] + row[j]
        row = new_row

    return row[n - 1]
```

Javascript

```
/**
 * TOP-DOWN APPROACH
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
function uniquePaths_TopDown(m, n) {
    const memo = Array.from({ length: m }, () => Array(n).fill(-1));

    /**
     * Recursive helper that uses memoization to store path counts
     */
    function solve(r, c) {
        if (r === 0 || c === 0) return 1;
        if (memo[r][c] !== -1) return memo[r][c];
```

```
        memo[r][c] = solve(r - 1, c) + solve(r, c - 1);
        return memo[r][c];
    }

    return solve(m - 1, n - 1);
}

/**
 * BOTTOM-UP APPROACH (Space Optimized)
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
function uniquePaths_BottomUp(m, n) {
    // Current state of the "previous" row
    let row = new Array(n).fill(1);

    for (let i = 1; i < m; i++) {
        // We can actually do this in-place in a single array
        for (let j = 1; j < n; j++) {
            // current cell = itself (above) + previous index (left)
            row[j] = row[j] + row[j - 1];
        }
    }

    return row[n - 1];
}
```

Note 1: Techniques Used

- **Dynamic Programming (DP):** A method for solving complex problems by breaking them down into simpler subproblems. It is applicable here because the path to a cell depends on the paths to its neighbors.
- **Memoization:** "Remembering" results of expensive function calls so you don't repeat the work.
- **Space Optimization:** Recognizing that a 2D DP table is often redundant if you only need the previous "layer" of data to compute the next.

Note 2: Real-World / Indirect Interview Variations

1. **Obstacles (Leetcode 63):** "There's a rock at (2,2). How many paths now?" (Logic: if cell has obstacle, paths = 0).
2. **Minimum Path Sum (Leetcode 64):** "Each cell has a cost. Find the cheapest path to the end." (Logic: instead of sum, use `min(top, left)`

`+ current_cost`).
3. **Dungeon Game:** A Meta favorite. You have health and move through a grid. What's the minimum health needed to reach the end?
4. **Bloomberg Variation:** "A trader can move through a price grid. Find the path with the max profit, but you can only move Right or Down-Right."

55. Jump Game

A top-tier engineer at a company like Google doesn't just look for *a* solution; they look for the *most efficient* one while understanding the trade-offs of every approach. For "Jump Game," an L5/L6 would quickly identify that while this looks like a Dynamic Programming (DP) problem, it can actually be solved in linear time using a **Greedy** approach.

Here is the breakdown of the problem and the evolution of the solution.

---

1. Problem Explanation

You are given an array of non-negative integers. Each element represents your **maximum** jump length at that position. You start at the first index, and your goal is to determine if you can reach the last index.

Key Insight: You don't have to jump the *exact* number at your current index. If `nums[i] = 3`, you can jump 1, 2, or 3 steps.

Example: `nums = [2, 3, 1, 1, 4]`

- At index 0, you can jump up to 2 steps (to index 1 or 2).
- If you jump to index 1, you can then jump up to 3 steps, easily reaching the end.
- **Result:** True.

---

2. Solution Explanation

The Intuitive Start: Top-Down DP (Backtracking + Memoization)

The most "natural" way to think about this is: "From where I am, can I reach the end by trying every possible jump?"

1. Start at index 0.
2. Try jumping 1 step, 2 steps, ..., up to `nums[0]` steps.
3. For each jump, recursively ask the same question: "Can I reach the end from this new index?"

4. **Optimization:** Use a "memo" table to remember if an index is `GOOD` (can reach end) or `BAD` (cannot), so we don't recalculate.

The Pivot: Greedy Strategy

While DP works, an L6 engineer would notice that we only care about the **farthest point** we can currently reach. If our "max reach" ever meets or exceeds the last index, we win.

ASCII Visualization: The "Farthest Reach" Method Let's trace `nums = [2, 3, 1, 0, 4]`

```
Step 0: Start at index 0. nums[0] = 2.
Current Max Reach = 2.
Index: [ 0 ]  1   2   3   4
Nums:  [ 2 ]  3   1   0   4
Range:  |-------| (Can reach up to index 2)

Step 1: Move to index 1. nums[1] = 3.
Potential reach from here: index 1 + 3 = 4.
New Max Reach = max(2, 4) = 4.
Index:   0  [ 1 ]  2   3   4
Nums:    2  [ 3 ]  1   0   4
Range:        |-----------| (Can reach up to index 4)

Step 2: Max Reach (4) >= Last Index (4).
RESULT: SUCCESS
```

ASCII Visualization: The "Backward Goal" Method Alternatively, imagine the goal is a wall. If a position can jump to or past the wall, that position *becomes* the new wall (the new goal).

```
Array: [ 2,  3,  1,  1,  4 ]
Indices: 0   1   2   3   4

Initial Goal: Index 4

Check Index 3: nums[3] is 1. index 3 + 1 = 4.
Can reach goal? Yes.
New Goal: Index 3
[ 2,  3,  1, (1),  4 ]
              ^Goal

Check Index 2: nums[2] is 1. index 2 + 1 = 3.
Can reach goal (3)? Yes.
New Goal: Index 2
[ 2,  3, (1),  1,  4 ]
```

```
          ^Goal

Check Index 1: nums[1] is 3. index 1 + 3 = 4.
Can reach goal (2)? Yes (4 is > 2).
New Goal: Index 1
[ 2, (3),  1,  1,  4 ]
      ^Goal

Final Check: Can index 0 reach Goal (1)?
nums[0] is 2. index 0 + 2 = 2.
Yes, 2 > 1.
New Goal: Index 0.

If Goal == 0 at the end, return True.
```

---

3. Complexity Analysis

Time Complexity (TC)

We iterate through the array exactly once.

```
Total Operations = Number of elements in array (n)
TC = O(n)

ASCII Derivation:
[ x ] -> [ x ] -> [ x ] ... -> [ x ]
1   +    1    +    1   ...  +   1    = n operations
```

Space Complexity (SC)

We only store one or two variables (the `farthest_reach` or the `goal` index).

```
Variables stored: goal (integer)
SC = O(1)

ASCII Derivation:
Memory Used: | Goal (4 bytes) |
(Constant size regardless of input array size)
```

---

4. Solution Code

Greedy Approach (Recommended by L5/L6 for Efficiency)

Python (Greedy)

```python
def canJump(nums):
    # We start the goal at the very last index
    goal = len(nums) - 1

    # Iterate backwards from the second-to-last element to the first
    for i in range(len(nums) - 1, -1, -1):
        # If the current position plus its max jump can reach the goal
        if i + nums[i] >= goal:
            # Move the goalpost to the current position
            goal = i

    # If the goalpost reached the start, we have a path
    return goal == 0
```

JavaScript (Greedy)

```javascript
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var canJump = function(nums) {
    let goal = nums.length - 1;

    for (let i = nums.length - 1; i >= 0; i--) {
        // If current index + jump capacity reaches or passes goal
        if (i + nums[i] >= goal) {
            goal = i;
        }
    }

    return goal === 0;
};
```

Dynamic Programming (Bottom-Up)

Python (DP)

```python
def canJump(nums):
    n = len(nums)
    # dp[i] stores whether index i can reach the end
    dp = [False] * n
    dp[n - 1] = True  # The last index is already at the end

    for i in range(n - 2, -1, -1):
        furthest_jump = min(i + nums[i], n - 1)
```

```
        for j in range(i + 1, furthest_jump + 1):
            if dp[j]:
                dp[i] = True
                break
    return dp[0]
```

JavaScript (DP)

```
var canJump = function(nums) {
    const n = nums.length;
    const dp = new Array(n).fill(false);
    dp[n - 1] = true;

    for (let i = n - 2; i >= 0; i--) {
        let furthest = Math.min(i + nums[i], n - 1);
        for (let j = i + 1; j <= furthest; j++) {
            if (dp[j] === true) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[0];
};
```

Terminology & Real-World Context

Greedy Algorithm: An algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

- **Why it helps:** In this problem, it converts an O(n^2) DP problem into an O(n) solution by realizing we don't need to know *how* we reached a point, only the *farthest* point reachable.

Indirect Interview Variants

Companies rarely ask "Jump Game" directly anymore; they use these variations:

1. **Google (Resource Management):** "Given a set of refueling stations with different fuel capacities, can you reach the destination?" (Exactly Jump Game 1).
2. **Meta (UI/UX flow):** "Given a series of app screens and the number of permitted 'back' or 'forward' actions from each, can a user reach the checkout page from the home screen?"

3. **Bloomberg (Liquidity/Trading):** "Given a sequence of trades and the maximum 'slippage' or move allowed from each trade, can you reach a specific target price?"

Jump Game II

This is a classic problem, and you are right to ask how a senior engineer would approach it. An entry-level engineer might just try to find *any* solution that works, but a top-of-band L5 or L6 engineer at Google or Meta will immediately look for the optimal pattern.

In an interview, a senior engineer will recognize that while this *can* be solved with Dynamic Programming (DP), the DP approach is sub-optimal. They will briefly explain the DP intuition to show thoroughness, but quickly pivot to the most efficient approach: a **Greedy Algorithm** modeled as an **Implicit Breadth-First Search (BFS)**.

Let's break this down exactly how you would discuss it in a high-level technical interview.

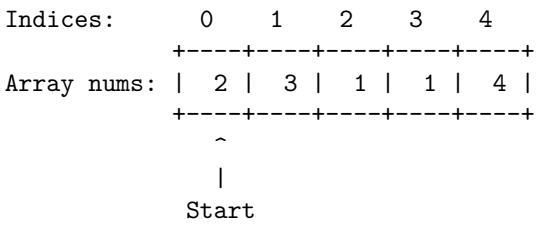
---

1. Problem Explanation

You are given an array of non-negative integers.

- Each element in the array represents your **maximum** jump length from that specific position.
- You start at the first index (index 0).
- Your goal is to reach the last index using the **minimum number of jumps**.
- The problem guarantees that it is always possible to reach the last index.

Let's look at a basic example: Input array: `[2, 3, 1, 1, 4]`



From index 0, the value is `2`. This means you can jump a maximum of 2 steps. You can choose to land on index `1` or index `2`. We need to figure out which choices lead to the end in the fewest total jumps.

---

2. Solution Explanation

Before jumping to the optimal solution, let's look at the DP approach, as it's the most common initial thought process.

The Intuitive DP Approach (Sub-optimal) If you view this as a DP problem, you want to break it down into smaller subproblems.

- **Top-Down (Memoization):** You start at index 0 and say, "I can jump to index 1 or 2. Let me recursively calculate the minimum jumps to reach the end from index 1, and the minimum jumps from index 2. I'll take the minimum of those two answers and add 1." You save these answers in an array to avoid recalculating them.
- **Bottom-Up (Tabulation):** You create an array `dp` of the same length as `nums`, filled with infinity. You set `dp[0] = 0`. Then, for every index `i`, you look at all the indices you can reach from `i`, and update their `dp` values: `dp[reachable_index] = min(dp[reachable_index], dp[i] + 1)`.

Why an L5/L6 engineer abandons DP here: The DP approach requires a nested loop. For every element, you might iterate through up to N subsequent elements. This gives a Time Complexity of O(N^2). In the real world (and in Google/Meta interviews), O(N^2) for this specific problem is too slow.

The Optimal L5/L6 Approach: Greedy / Implicit BFS A senior engineer will realize that this problem can be solved in a single pass (O(N) time) by treating it like a Breadth-First Search (BFS) on a graph, without actually building the graph.

Instead of looking at individual jumps, we look at **windows** (or "levels" in BFS terminology).

- **Window 0:** The starting index.
- **Window 1:** All indices you can reach from Window 0.
- **Window 2:** All indices you can reach from Window 1.

Every time we move to a new window, we increment our jump count. We just need to keep track of the boundaries of our current window and the farthest point we can reach for the *next* window.

Let's visualize the step-by-step Greedy execution: Input: `[2, 3, 1, 1, 4]`

Variables we will track:

- `jumps`: Total jumps taken so far.
- `current_window_end`: The farthest index we can reach with our *current* number of jumps.
- `farthest`: The absolute farthest index we have discovered we can reach so far.

Initialization:

```
jumps = 0
current_window_end = 0
farthest = 0
```

```
            +----+----+----+----+----+
Array nums: |  2 |  3 |  1 |  1 |  4 |
            +----+----+----+----+----+
Indices:      0    1    2    3    4
              ^
              |
      Current Window (Ends at 0)
```

Iteration Step 1 (Index 0): We are at index 0. The value is 2. Farthest we can reach is: index 0 + value 2 = index 2. `farthest` becomes max(0, 2) = 2.

```
            +----+----+----+----+----+
Array nums: |  2 |  3 |  1 |  1 |  4 |
            +----+----+----+----+----+
Indices:      0    1    2    3    4
              ^         ^
              |         |
              i      farthest (2)
```

We have reached the `current_window_end` (which is 0). This means we MUST make a jump to move forward.

- Increment `jumps` to 1.
- Update `current_window_end` to our `farthest` value (2). This defines our next window.

Iteration Step 2 (Index 1): We are at index 1. The value is 3. Farthest we can reach is: index 1 + value 3 = index 4. `farthest` becomes max(2, 4) = 4.

```
            +----+----+----+----+----+
Array nums: |  2 |  3 |  1 |  1 |  4 |
            +----+----+----+----+----+
Indices:      0    1    2    3    4
                   ^              ^
                   |              |
                   i           farthest (4)
```

`Current Window is indices [1, 2]. We are inside it.`

Iteration Step 3 (Index 2): We are at index 2. The value is 1. Farthest we can reach is: index 2 + value 1 = index 3. `farthest` remains max(4, 3) = 4.

```
            +----+----+----+----+----+
Array nums: |  2 |  3 |  1 |  1 |  4 |
```

```
            +----+----+----+----+----+
Indices:      0    1    2    3    4
                        ^         ^
                        |         |
                        i      farthest (4)
```

Notice that index 2 is the `current_window_end`! We have explored all options in our current window. We MUST make another jump to enter the next window.

- Increment `jumps` to 2.
- Update `current_window_end` to `farthest` (4).

Iteration Step 4 (Index 3): We are at index 3. Wait, our `current_window_end` is now 4, which is the last index. We don't even need to process the rest of the array because we know our current window already covers the end of the array. The loop can safely ignore the last element because we are guaranteed to reach it.

Total `jumps` = 2.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) Where N is the length of the array. We iterate through the array exactly one time.

```
Derivation Visualization (Time Complexity):

Array: [2, 3, 1, 1, 4]
Pass:   -> -> -> -> (Stop before last element)

Operations per element:
- Read value
- Calculate max (O(1))
- Check if i == current_window_end (O(1))

Total time = N elements * O(1) operations = O(N)
```

Space Complexity: O(1) We only use three variables (`jumps`, `current_window_end`, `farthest`) regardless of the size of the input array. We are not allocating any new arrays or call stacks.

```
Derivation Visualization (Space Complexity):

Memory Allocation:
+----------------------+
| jumps: Integer       | -> O(1) memory
| window_end: Integer  | -> O(1) memory
| farthest: Integer    | -> O(1) memory
```

```
+----------------------+
Total Auxiliary Space = O(1)
```

---

4. Solution Code

Here are the optimal Greedy solutions in JavaScript and Python.

JavaScript Snippet:

```javascript
/**
 * Calculates the minimum number of jumps to reach the last index.
 * We use an implicit BFS (Greedy) approach to track reachable windows.
 */
function jump(nums) {
    let jumps = 0;
    let currentWindowEnd = 0;
    let farthestReachable = 0;

    // We loop up to nums.length - 1 because we don't need to jump from the last index.
    for (let i = 0; i < nums.length - 1; i++) {
        // Update the farthest we can reach from the current window
        farthestReachable = Math.max(farthestReachable, i + nums[i]);

        // If we have reached the end of the current window, we must jump
        if (i === currentWindowEnd) {
            jumps++;
            currentWindowEnd = farthestReachable;

            // Minor optimization: If our window covers the end, we can stop early
            if (currentWindowEnd >= nums.length - 1) {
                break;
            }
        }
    }

    return jumps;
}
```

Python Snippet:

```python
def jump(nums):
    """
    Calculates the minimum number of jumps to reach the last index.
    Uses a Greedy approach tracking the maximum reach per "jump level".
    """
    jumps = 0
```

```
    current_window_end = 0
    farthest_reachable = 0

    # Iterate through the array, stopping right before the last element
    for i in range(len(nums) - 1):
        # Continually track the furthest point we can reach
        farthest_reachable = max(farthest_reachable, i + nums[i])

        # When we hit the boundary of our current jump's reach...
        if i == current_window_end:
            jumps += 1                          # ...we make a jump
            current_window_end = farthest_reachable  # ...and set the new boundary

            # Optimization: break early if the new boundary reaches or passes the end
            if current_window_end >= len(nums) - 1:
                break

    return jumps
```

Note 1: Terminology Explained

- **Greedy Algorithm:** An algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. In this problem, it applies because maximizing our `farthest_reachable` at every step guarantees we cover the most ground, leading to the fewest jumps. We never have to "undo" a jump choice.
- **Implicit BFS (Breadth-First Search):** BFS is usually used on explicit graphs (nodes and edges). "Implicit BFS" means the data isn't structured as a graph in memory, but we traverse it conceptually exactly like a graph. The "windows" we tracked perfectly map to the "levels" of a BFS tree. This helps us find the shortest path (minimum jumps) optimally.

Note 2: Real-World Interview Variations

Companies like Google, Meta, and Bloomberg rarely ask "Jump Game II" verbatim anymore. They wrap the exact same logic in real-world scenarios:

1. **Video Streaming / Buffering (Meta/Google):** "You are given a list of video chunks in a buffer. Each chunk at index `i` has a length `L`. You want to skip through the video to the end using the minimum number of clicks, where one click fast-forwards you anywhere within your currently buffered chunk. Find the minimum clicks."
2. **Watering a Garden / Cell Towers (Bloomberg/Google):** *LeetCode 1326: Minimum Number of Taps to Open to Water a Garden.* "You have a 1D garden and an array of taps. Each tap covers a specific radius left

and right. Find the minimum number of taps needed to cover the entire garden." (This requires an initial sorting/interval mapping step, but the core logic is identical to Jump Game II).

3. **Network Packet Routing:** "A packet must travel through a series of routers. Each router has a maximum transmission range (number of nodes it can skip). What is the minimum number of hops for the packet to reach the destination server?"

1526. Minimum Number of Increments on Subarrays to Form a Target Array

An L5/L6 engineer at a company like Google doesn't just look for a "working" solution; they look for the **optimal mathematical insight** that reduces a complex-looking problem into a single pass (O(n)).

This problem, at first glance, looks like it requires complex Segment Trees or Divide and Conquer, but the "Senior Engineer" insight is realizing this is about **accumulated differences**.

---

1. Problem Explanation

You are given an array `target`. You start with an array `initial` of the same length, filled with zeros. In one operation, you can choose any **subarray** and increment every element in it by 1. What is the minimum number of operations to make `initial` equal to `target`?

The Core Challenge

If we have `target = [1, 2, 3, 2, 1]`, we want to build this using the fewest "blocks" possible.

Imagine the target array as a mountain range. Every time the height increases, you are starting "new" increment operations that weren't covered by the previous steps. When the height decreases, you are simply "ending" operations you already started.

---

2. Solution Explanation

The Intuition: "The Wall Builder"

Think of this like building a wall with bricks.

- If the next section of the wall is **taller** than the current one, you *must* start new operations to fill that extra height.

- If the next section is **shorter** or the **same**, you can just "carry over" the operations you were already doing.

Step-by-Step Visualization

Let's use `target = [3, 1, 5, 4, 2]`

Step 1: Look at index 0 (Value: 3) We start at 0. To get to 3, we need 3 operations. Total Ops: 3

```
Value: 3
Visual:
[#]
[#]
[#]
Current Total: 3
```

Step 2: Look at index 1 (Value: 1) We are at height 3 and moving to height 1. Since $1 < 3$, we don't need any *new* operations. We just "stop" two of the operations we were already doing. Total Ops: 3 (No change)

```
Values: [3, 1]
Visual:
[#]
[#]
[#] [#]  <-- The bottom row "carried over"
Current Total: 3
```

Step 3: Look at index 2 (Value: 5) We are at height 1 and moving to height 5. We need to add 4 *new* levels (5 - 1 = 4). Total Ops: 3 + 4 = 7

```
Values: [3, 1, 5]
Visual:
            [#] <-- New
            [#] <-- New
            [#] <-- New
[#]         [#] <-- New
[#]         [#]
[#] [#]     [#]
Current Total: 7
```

Step 4: Look at index 3 (Value: 4) 4 is less than 5. No new operations needed. Total Ops: 7

Step 5: Look at index 4 (Value: 2) 2 is less than 4. No new operations needed. Total Ops: 7

Final Answer: 7

The Logic Rule

```
Operations = target[0] + Sum of all (target[i] - target[i-1])
where target[i] > target[i-1]
```

---

3. Time and Space Complexity Analysis

The L5 approach is to solve this in a single pass without extra storage.

```
TIME COMPLEXITY: O(N)
-----------------------------------------------------------
| Phase              | Operation            | Complexity   |
|--------------------|----------------------|--------------|
| Iteration          | One pass through N   | O(N)         |
| Comparison/Math    | Constant time per i  | O(1)         |
| Total              | N * 1                | O(N)         |
-----------------------------------------------------------

SPACE COMPLEXITY: O(1) or O(N)
-----------------------------------------------------------
| Type               | Reason               | Complexity   |
|--------------------|----------------------|--------------|
| Iterative (Greedy)| Only 1-2 variables   | O(1)         |
| Recursive (DP)     | Call stack depth N   | O(N)         |
-----------------------------------------------------------
```

---

4. Solution Code

Note on DP Approaches

While this problem is most efficiently solved with the **Greedy** approach (O(1) space) shown above, it can be framed as DP.

- **Bottom-Up (Iterative):** We build the solution index by index based on the previous height.
- **Top-Down (Recursive):** We ask "How many ops to build up to index `i`?" which depends on `i-1`.

Python Snippet

```python
# BOTTOM-UP ITERATIVE (Highly Recommended/Optimal)
def minNumberOperations_BottomUp(target):
    # The first element always requires target[0] operations
    res = target[0]
```

```python
    # We iterate from the second element to the end
    # If the current height is greater than the previous,
    # the difference represents 'new' subarrays starting.
    for i in range(1, len(target)):
        if target[i] > target[i-1]:
            res += (target[i] - target[i-1])

    return res

# TOP-DOWN RECURSIVE (With Memoization)
# Note: Python's recursion limit might hit for large N,
# but this demonstrates the logic.
import sys
sys.setrecursionlimit(200000)

def minNumberOperations_TopDown(target):
    memo = {}

    def solve(idx):
        if idx == 0:
            return target[0]
        if idx in memo:
            return memo[idx]

        # Operations needed for the previous part of the array
        prev_ops = solve(idx - 1)

        # New operations needed only if current is taller than previous
        added_ops = max(0, target[idx] - target[idx-1])

        memo[idx] = prev_ops + added_ops
        return memo[idx]

    return solve(len(target) - 1)
```

JavaScript Snippet

```javascript
/**
 * BOTTOM-UP ITERATIVE
 * This is the standard "production-grade" solution.
 */
function minNumberOperations_BottomUp(target) {
    let result = target[0];

    for (let i = 1; i < target.length; i++) {
        // Only add the positive "delta" (increase in height)
```

```
        if (target[i] > target[i - 1]) {
            result += (target[i] - target[i - 1]);
        }
    }

    return result;
}

/**
 * TOP-DOWN RECURSIVE
 * Using an array for memoization to store results of sub-problems.
 */
function minNumberOperations_TopDown(target) {
    const memo = new Array(target.length).fill(-1);

    function dp(idx) {
        // Base case: first element
        if (idx === 0) return target[0];
        if (memo[idx] !== -1) return memo[idx];

        // Cost is cost of previous + any increase at current step
        const diff = Math.max(0, target[idx] - target[idx - 1]);
        memo[idx] = dp(idx - 1) + diff;

        return memo[idx];
    }

    return dp(target.length - 1);
}
```

Terminology: "The Difference Array" Technique

This solution relies on the concept of a **Difference Array**. In many range-update problems, instead of tracking the values themselves, we track the change between adjacent values. Here, any "increase" in the difference array signifies a new set of operations starting, while a "decrease" signifies operations ending.

Real-World / Interview Variants

Google and Meta often wrap this logic in "System Design Lite" or "Physical Simulation" contexts:

1. **The "Skyline Building" Problem (Google):** You are given a silhouette of a city. You have a crane that can place a row of bricks across any continuous segment. What is the minimum number of crane drops?

2. **The "Water Flow / Dam" Problem (Bloomberg):** Given a 2D side-view of a terrain, how many "layers" of concrete must be poured to reach a certain elevation profile?
3. **The "Stock Trading Signal" (Meta):** You are given a target number of shares to hold each day. You can only buy/sell in "blocks" over consecutive days. How many buy transactions are needed? (This is essentially the same math).

42. Trapping Rain Water

This is a classic problem that separates those who can "brute force" from those who can optimize for space and time. At an L5/L6 level (Senior/Staff), Google interviewers look for more than just the code—they want to see you recognize the **bottleneck** and move from a 3-pass solution to a single-pass 2-pointer approach.

---

1. Problem Explanation

Imagine you have a series of bars of varying heights. When it rains, water gets trapped between these bars. The goal is to calculate the total units of water trapped.

The Golden Rule of Trapping Water: For any single bar at index `i`, the amount of water it can hold is determined by the **shorter** of the two tallest walls to its left and its right, minus its own height.

```
Water at i = max(0, min(max_left, max_right) - height[i])
```

---

2. Solution Explanation

The Intuition: "The Three-Pass Approach"

Before jumping to the 2-pointer "pro" solution, you must understand the logic of the walls.

For every index, we need to know:

1. What is the tallest bar to the left?
2. What is the tallest bar to the right?

Step-by-Step ASCII Visualization Consider the input: `[0, 1, 0, 2, 1, 0, 1, 3]`

The Input Landscape:

```
      #
  #   #
_##_###
01234567 (indices)
[0,1,0,2,1,0,1,3] (heights)
```

Step 1: Compute Max Left for each position Scanning from left to right, we keep track of the highest bar seen so far.

```
Index:      0  1  2  3  4  5  6  7
Height:     0  1  0  2  1  0  1  3
Max Left:   0  1  1  2  2  2  2  3
```

Step 2: Compute Max Right for each position Scanning from right to left, we keep track of the highest bar seen so far.

```
Index:      0  1  2  3  4  5  6  7
Height:     0  1  0  2  1  0  1  3
Max Right:  3  3  3  3  3  3  3  3
```

Step 3: Calculate Water For each index, take `min(Left, Right) - Height`.

```
Index:      0  1  2  3  4  5  6  7
Min(L,R):   0  1  1  2  2  2  2  3
Height:     0  1  0  2  1  0  1  3
---------------------------------
Water:      0  0  1  0  1  2  1  0  => Total: 5 units
```

The "Senior Engineer" Optimization: Two Pointers

The L5/L6 approach realizes we don't need two full arrays. We can use two pointers (`left` and `right`) and two variables (`left_max` and `right_max`).

How it works: If `left_max < right_max`, we know that for the `left` pointer, the "limiting factor" is the `left_max`. We don't care how high the right side is, as long as it's higher than `left_max`.

Visualizing the 2-Pointer Logic:

```
L = Left Pointer, R = Right Pointer
LM = Left Max, RM = Right Max

Initial: [0, 1, 0, 2, 1, 0, 1, 3]
          L                   R
          LM=0                RM=3

1. LM(0) < RM(3) -> Process L. Water += LM - height[L] (0). Move L.
2. LM(1) < RM(3) -> Process L. Water += LM - height[L] (0). Move L.
```

```
3. LM(1) < RM(3) -> Process L. Water += 1 - 0 = 1 unit. Move L.
... and so on.
```

---

3. Time and Space Complexity Analysis

The complexity is derived by looking at how many times we touch each element in the input array.

Time Complexity:

```
Operation          | Complexity | Reason
-------------------|------------|-------------------------------------
Initialize Pointers| O(1)       | Constant time setup
Single Pass Scan   | O(n)       | We move L or R once per element
-------------------|------------|-------------------------------------
Total Time          | O(n)       | Linear time relative to input size
```

Space Complexity:

```
Approach           | Complexity | Reason
-------------------|------------|-------------------------------------
Dynamic Programming| O(n)       | Requires LeftMax and RightMax arrays
Two Pointers       | O(1)       | Only uses 4 integer variables
-------------------|------------|-------------------------------------
Total Space        | O(1)       | Constant space (for Two Pointers)
```

---

4. Solution Code

Approach A: Two Pointers (The most efficient)

Python

```python
def trap(height):
    if not height:
        return 0

    # Initialize pointers and boundary trackers
    left, right = 0, len(height) - 1
    left_max, right_max = height[left], height[right]
    water = 0

    # Process the array from both ends meeting in the middle
    while left < right:
        # We always process the side with the smaller max boundary
        # because the smaller side is the bottleneck for water height.
        if left_max < right_max:
```

```
            left += 1
            left_max = max(left_max, height[left])
            water += left_max - height[left]
        else:
            right -= 1
            right_max = max(right_max, height[right])
            water += right_max - height[right]

    return water
```

JavaScript

```
/**
 * Calculates trapped water using two-pointer strategy.
 * This is the O(1) space complexity approach favored in L5+ interviews.
 */
function trap(height) {
    let left = 0;
    let right = height.length - 1;
    let leftMax = 0;
    let rightMax = 0;
    let totalWater = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            // If the current left height is a new max, update it.
            // Otherwise, it can hold water because we know height[right] is taller.
            height[left] >= leftMax ? (leftMax = height[left]) : (totalWater += leftMax - he
            left++;
        } else {
            // Same logic for the right side.
            height[right] >= rightMax ? (rightMax = height[right]) : (totalWater += rightMax
            right--;
        }
    }
    return totalWater;
}
```

Approach B: Dynamic Programming (Bottom-Up)

Python

```
def trap_dp(height):
    n = len(height)
    if n == 0: return 0
```

```
    left_max = [0] * n
    right_max = [0] * n

    # Fill left_max array: tallest bar to the left of i
    left_max[0] = height[0]
    for i in range(1, n):
        left_max[i] = max(height[i], left_max[i-1])

    # Fill right_max array: tallest bar to the right of i
    right_max[n-1] = height[n-1]
    for i in range(n-2, -1, -1):
        right_max[i] = max(height[i], right_max[i+1])

    # Total water is the min of the two boundaries minus current height
    water = 0
    for i in range(n):
        water += min(left_max[i], right_max[i]) - height[i]

    return water
```

JavaScript

```
/**
 * DP approach: Pre-calculating the boundaries for every index.
 * Helpful for explaining the core logic before optimizing space.
 */
function trapDP(height) {
    const n = height.length;
    if (n === 0) return 0;

    const leftMax = new Array(n).fill(0);
    const rightMax = new Array(n).fill(0);
    let totalWater = 0;

    leftMax[0] = height[0];
    for (let i = 1; i < n; i++) {
        leftMax[i] = Math.max(height[i], leftMax[i - 1]);
    }

    rightMax[n - 1] = height[n - 1];
    for (let i = n - 2; i >= 0; i--) {
        rightMax[i] = Math.max(height[i], rightMax[i + 1]);
    }

    for (let i = 0; i < n; i++) {
```

```
        totalWater += Math.min(leftMax[i], rightMax[i]) - height[i];
    }

    return totalWater;
}
```

Key Terms & Techniques

- **Monotonicity (Two Pointers):** We use the fact that the water level at any point is strictly limited by the minimum of the maximum heights on both sides. This "monotone" property allows us to move pointers inward without ever needing to look back.
- **Preprocessing (DP):** The DP approach uses "memoization" (storing results of subproblems) to avoid re-scanning the array for every single index.

Real-World Interview Variations

1. **Google (Infrastructure):** "Given a 2D grid representing elevation, how much water is trapped after a storm?" (This is LeetCode 407: Trapping Rain Water II).
2. **Meta (Product):** "You are building a histogram UI component. Given user data points, calculate the maximum amount of 'filler' color needed to level out troughs between peaks."
3. **Bloomberg (Finance):** "Stock price peaks. If you buy at a peak and sell at a peak, what is the maximum 'volume' of price movement you can capture between them?"

17. Letter Combinations of a Phone Number

This is a classic "Combinatorial Search" problem. A top-tier engineer at Google (L5/L6) doesn't just look for a way to get the right output; they look for the most memory-efficient way to traverse the **State Space Tree**.

---

1. Problem Explanation

Imagine an old-school T9 phone keypad. Each digit (2-9) is mapped to a set of letters.

- 2: "abc"
- 3: "def"
- 4: "ghi" … and so on.

Given a string of digits (e.g., “23”), we need to return all possible letter combinations that those numbers could represent.

The “Non-Trivial” Insight: This isn’t just a loop. If you have “234”, the first digit ‘2’ has 3 choices, the second ‘3’ has 3 choices, and the third ‘4’ has 3 choices. This creates a **Cartesian Product**. We are essentially building a path from the first digit to the last, picking one letter at each step.

---

2. Solution Explanation (The Backtracking Approach)

A Senior Engineer would use **Backtracking** (Top-Down) or a **Queue-based BFS** (Bottom-Up). Let’s visualize the Top-Down approach as it’s the most intuitive for recursive branching.

The State Space Tree (Visualization)

Input: “23” Mapping: 2 -> “abc”, 3 -> “def”

```
Level 0: Empty String "" (Start)
                  |
         -------------------------
         |           |           |
Level 1: (Picking from '2')
        "a"         "b"         "c"
         |           |           |
     ---------   ---------   ---------
     |   |   |   |   |   |   |   |   |
Level 2: (Picking from '3')
   "ad" "ae" "af" "bd" "be" "bf" "cd" "ce" "cf"
```

Step-by-Step Execution Diagram

Let’s trace the recursion for digits = “23”:

Step 1: Start at Index 0 (Digit ‘2’)

```
Current Path: []
Digit: '2' -> Choices: [a, b, c]
Action: Pick 'a', move to Index 1
```

Step 2: At Index 1 (Digit ‘3’)

```
Current Path: [a]
Digit: '3' -> Choices: [d, e, f]
Action: Pick 'd', move to Index 2 (End)
Result Found: "ad"
```

Step 3: Backtrack (Pop ‘d’)

```
Current Path: [a]
Action: Pick 'e', move to Index 2
Result Found: "ae"
... and so on.
```

---

3. Time and Space Complexity Analysis

We avoid complex math symbols and use basic logic.

Time Complexity (TC)

```
Total Operations = (Number of choices per digit) ^ (Number of digits)

If N = length of digits string:
- Most digits have 3 letters (abc, def, etc.)
- Some digits have 4 letters (pqrs, wxyz)

Worst Case TC: 4^N * N
(The extra * N comes from the time it takes to build the final string
and join the characters at each leaf node).
```

Space Complexity (SC)

```
Recursion Stack Depth:
The depth of the tree is equal to the length of the input string (N).
SC = N

Output Space:
The number of strings in the result list is 4^N.
(Usually, output space is not counted in algorithmic SC unless specified).
```

---

4. Solution Code

Top-Down (Recursive Backtracking)

This is usually the "cleanest" way to write it in an interview.

Python

```python
def letterCombinations(digits):
    if not digits:
        return []

    mapping = {
```

```
        "2": "abc", "3": "def", "4": "ghi", "5": "jkl",
        "6": "mno", "7": "pqrs", "8": "tuv", "9": "wxyz"
    }

    result = []

    # backtrack function explores all paths in the decision tree
    # index: current digit we are looking at
    # path: the current string we are building
    def backtrack(index, path):
        # Base Case: if path length matches digits length, we found a combination
        if len(path) == len(digits):
            result.append("".join(path))
            return

        # Get the letters for the current digit
        current_digit = digits[index]
        for letter in mapping[current_digit]:
            path.append(letter)      # Choose
            backtrack(index + 1, path) # Explore
            path.pop()               # Un-choose (Backtrack)

    backtrack(0, [])
    return result
```

JavaScript

```
var letterCombinations = function(digits) {
    if (!digits.length) return [];

    const mapping = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    };

    const result = [];

    /**
     * Recursive helper to build combinations.
     * @param {number} i - current digit index
     * @param {string} str - current accumulated string
     */
    const backtrack = (i, str) => {
        if (i === digits.length) {
            result.push(str);
            return;
```

```
        }

        for (const char of mapping[digits[i]]) {
            backtrack(i + 1, str + char);
        }
    };

    backtrack(0, "");
    return result;
};
```

Bottom-Up (Iterative / BFS)

This approach treats the problem like a "rolling" list of combinations.

Python

```python
def letterCombinations(digits):
    if not digits: return []

    mapping = {"2":"abc", "3":"def", "4":"ghi", "5":"jkl",
               "6":"mno", "7":"pqrs", "8":"tuv", "9":"wxyz"}

    # Initialize queue with an empty string
    queue = [""]

    for digit in digits:
        temp_queue = []
        # For every combination we have so far, add every letter of the new digit
        for combination in queue:
            for letter in mapping[digit]:
                temp_queue.append(combination + letter)
        queue = temp_queue # Update the queue for the next digit

    return queue
```

JavaScript

```javascript
var letterCombinations = function(digits) {
    if (!digits.length) return [];

    const mapping = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    };
```

```
    let combinations = [""];

    for (const digit of digits) {
        const nextLevel = [];
        for (const prev of combinations) {
            for (const letter of mapping[digit]) {
                nextLevel.push(prev + letter);
            }
        }
        combinations = nextLevel;
    }

    return combinations;
};
```

Note 1: Terminology

Backtracking: This is a refined version of "Brute Force." It explores all possible candidates, but it builds the solution incrementally and abandons a candidate ("backtracks") as soon as it determines the candidate cannot lead to a valid solution. In this problem, it helps us explore every branch of the phone keypad without writing nested loops for every digit.

Note 2: Real World / Interview Variations

Companies like Google and Meta rarely ask this exact problem anymore; they ask variations to see if you can adapt:

1. **The Dictionary Filter (Google):** "Given a string of digits, return only the letter combinations that are **actual words** found in a provided dictionary." (Solution: Use a Trie to prune the backtracking tree).
2. **The "Invalid Keys" Variant (Bloomberg):** "What if some keys are broken or certain combinations are banned? How do you modify your search?"
3. **Keyboard Proximity (Meta):** "Instead of T9, imagine a QWERTY keyboard. If a user types 'G', they might have meant 'F' or 'H'. Generate all 'likely' words for a typed string."

410. Split Array Largest Sum (Binary Search) - Most optimal

Solving "Split Array Largest Sum" with **Binary Search** is a brilliant shift in perspective. Instead of trying to find where to "cut" the array (as we do in DP), we guess a "Max Sum" and check if it's possible to fit the array into `k` subarrays without exceeding that sum.

---

1. Category of Algorithm

This is **Binary Search on the Answer Space**. Instead of searching for an element in an array, we are searching for the smallest valid "Threshold Value" within a range of possible sums.

2. Problem It Solves

It minimizes the maximum subarray sum when splitting an array into exactly `k` continuous pieces. This "Minimize the Maximum" pattern is a massive hint for Binary Search.

---

3. Solution (What, Why, How)

The "What" (The Search Range)

The answer must fall between two values:

- **Low (Left):** The largest single element in `nums`. Any valid split must be at least this large.
- **High (Right):** The sum of all elements in `nums`. This is the case where $k = 1$.

The "Why" (Monotonicity)

If we can split the array with a Max Sum of **15**, we can definitely do it with **20**. If we can't do it with **10**, we definitely can't do it with **5**. This "Yes/No" property allows us to discard half the search space in every step.

The "How" (The Greedy Check)

For a "Guess" (Mid), we walk through the array and keep adding numbers to a current subarray. If the sum exceeds "Mid," we must start a new subarray (increment our counter). If our counter exceeds `k`, the "Mid" was too small.

Step-by-Step Visualization

Example: `nums = [7, 2, 5, 10, 8]`, `k = 2`

Step 1: Define Range

- **Left:** Max(7, 2, 5, 10, 8) = **10**
- **Right:** Sum(7+2+5+10+8) = **32**

Step 2: First Guess (Mid = (10 + 32) / 2 = 21) Check if we can split using Max Sum **21**:

```
[7, 2, 5] (Sum: 14) -> Add 10? (Sum: 24 > 21) -> NO.
Subarray 1: [7, 2, 5]
Subarray 2: [10, 8]
Total Subarrays used: 2.
Since 2 <= k, 21 is POSSIBLE. Try smaller (Right = 20).
```

Step 3: Second Guess (Mid = (10 + 20) / 2 = 15) Check if we can split using Max Sum **15**:

```
[7, 2, 5] (Sum: 14) -> Add 10? (Sum: 24 > 15) -> NO.
Subarray 1: [7, 2, 5]
[10] -> Add 8? (Sum: 18 > 15) -> NO.
Subarray 2: [10]
Subarray 3: [8]
Total Subarrays used: 3.
Since 3 > k, 15 is IMPOSSIBLE. Try larger (Left = 16).
```

Step 4: Third Guess (Mid = (16 + 20) / 2 = 18) Check if we can split using Max Sum **18**:

```
[7, 2, 5] (Sum: 14) -> Add 10? (Sum: 24 > 18) -> NO.
Subarray 1: [7, 2, 5]
[10, 8] (Sum: 18) -> Perfect.
Subarray 2: [10, 8]
Total Subarrays used: 2.
Since 2 <= k, 18 is POSSIBLE. Try smaller (Right = 17).
```

Result: After more narrowing, we find **18** is the smallest possible maximum.

---

4. Time and Space Complexity

Time Complexity: O(n * log(S))

- `n` is the length of the array.
- `S` is the sum of all elements in the array.
- We perform a Binary Search over the range `(Sum - Max)`. The number of steps is `log(S)`.
- Inside each step, we traverse the array once `O(n)`.

```
Binary Search Steps: log(Sum)
        |
        v
[ Array Traversal: O(n) ]
-------------------------
Total: O(n * log(Sum))
```

Space Complexity: O(1)

We only store a few variables (`left`, `right`, `mid`, `current_sum`, `count`). We do not use recursion or extra arrays.

---

6. Implementations

Python Implementation

```python
def split_array(nums, k):
    # The smallest possible max sum is the largest single number
    # The largest possible max sum is the total sum of the array
    left, right = max(nums), sum(nums)
    answer = right

    def can_split(max_sum_threshold):
        subarray_count = 1
        current_sum = 0

        for num in nums:
            if current_sum + num <= max_sum_threshold:
                current_sum += num
            else:
                # Need a new subarray
                subarray_count += 1
                current_sum = num

        return subarray_count <= k

    while left <= right:
        mid = (left + right) // 2

        if can_split(mid):
            # If possible, this could be our answer, but try even smaller
            answer = mid
            right = mid - 1
        else:
            # Too small, must increase the allowed sum
```

```
            left = mid + 1

    return answer

# Usage
print(split_array([7, 2, 5, 10, 8], 2)) # 18
```

JavaScript Implementation

```
var splitArray = function(nums, k) {
    let left = Math.max(...nums);
    let right = nums.reduce((a, b) => a + b, 0);
    let answer = right;

    const canSplit = (maxSumThreshold) => {
        let subarrayCount = 1;
        let currentSum = 0;

        for (const num of nums) {
            if (currentSum + num <= maxSumThreshold) {
                currentSum += num;
            } else {
                // Current subarray full, start next one
                subarrayCount++;
                currentSum = num;
            }
        }
        return subarrayCount <= k;
    };

    while (left <= right) {
        let mid = Math.floor((left + right) / 2);

        if (canSplit(mid)) {
            answer = mid;
            right = mid - 1; // Look for smaller valid maximum
        } else {
            left = mid + 1; // Increase threshold
        }
    }

    return answer;
};
```

---

7. Comparison with DP Approach

Feature	DP (Top-Down)	Binary Search
Time Complexity	O(k * n^2)	O(n * log(Sum))
Space Complexity	O(n * k)	O(1)
Intuitiveness	Breaking down choices	Guessing the answer
Performance	Slower for large N	Extremely fast

---

8. Relevant LeetCode Problems

This "Binary Search on Answer" technique is the exact solution for:

1. **Koko Eating Bananas (LC 875):** Find the minimum speed to eat all bananas.
2. **Capacity To Ship Packages Within D Days (LC 1011):** Find min ship capacity.
3. **Minimum Difficulty of a Job Schedule (LC 1335):** Can be solved this way if tasks are continuous.
4. **Path With Minimum Effort (LC 1631):** Finding min maximum weight on a path.

410. Split Array Largest Sum (Dynamic Programming)

The **Split Array Largest Sum** problem is a classic optimization challenge. While it is often solved more efficiently using Binary Search, the **Dynamic Programming (DP)** approach is more intuitive for understanding how to explore all possible ways to partition data.

---

1. Category of Algorithm

This approach falls under **Top-Down Dynamic Programming with Memoization**. It uses a recursive strategy to break the problem into sub-problems and stores results to avoid redundant calculations.

2. Problem It Solves

Given an array `nums` and an integer `k`, you must split the array into `k` non-empty continuous subarrays. The goal is to minimize the **largest sum** among these `k` subarrays.

---

3. Solution (What, Why, How)

The Logic

We need to make `k-1` "cuts" in the array.

- **What:** At any given index, we decide where to put the next cut.
- **Why:** By trying every possible cut position for the first subarray, and then recursively asking the same question for the remaining `k-1` subarrays, we can find the optimal solution.
- **How:** We keep track of our current `index` in the array and how many `subarraysLeft` we need to create.

Step-by-Step Visualization

Example: `nums = [7, 2, 5]`, `k = 2`

```
Decision Tree for solve(index=0, subarraysLeft=2):

Option 1: First subarray is [7] (sum = 7)
   -> Remaining: solve(index=1, subarraysLeft=1)
   -> Subarray is [2, 5] (sum = 7)
   -> Max of this path: max(7, 7) = 7

Option 2: First subarray is [7, 2] (sum = 9)
   -> Remaining: solve(index=2, subarraysLeft=1)
   -> Subarray is [5] (sum = 5)
   -> Max of this path: max(9, 5) = 9

Result: min(Option 1, Option 2) = 7
```

---

4. Time and Space Complexity

Time Complexity: O(k * n * n)

- `n` is the length of the array.
- There are `n * k` unique states in our DP table (index * subarraysLeft).
- For each state, we run a loop that can go up to `n` times to find the next split point.

```
States: [ 0...n ] x [ 1...k ]  --> (n * k)
Work per state: Loop through n elements
Total Time: n * k * n = O(k * n^2)
```

Space Complexity: O(n * k)

- We store results in a memoization table/hashmap of size `n * k`.
- The recursion stack depth is at most `n`.

6. Implementations

Python Implementation

```python
def split_array(nums, k):
    memo = {}

    # Precompute prefix sums for fast subarray sum calculation
    # sum(nums[i:j]) = prefix_sums[j] - prefix_sums[i]
    n = len(nums)
    prefix_sums = [0] * (n + 1)
    for i in range(n):
        prefix_sums[i+1] = prefix_sums[i] + nums[i]

    def solve(index, subarrays_left):
        # Base Case: If we only need 1 subarray, it must take all remaining elements
        if subarrays_left == 1:
            return prefix_sums[n] - prefix_sums[index]

        state = (index, subarrays_left)
        if state in memo:
            return memo[state]

        res = float('inf')
        # Try splitting the first subarray at every possible position
        # We must leave enough elements for the remaining subarrays_left
        for i in range(index, n - subarrays_left + 1):
            # Sum of the current first subarray
            current_sum = prefix_sums[i + 1] - prefix_sums[index]

            # Recurse for the remaining part
            largest_sum_remaining = solve(i + 1, subarrays_left - 1)

            # We want the 'largest sum' of this specific split
            current_max = max(current_sum, largest_sum_remaining)

            # We want to 'minimize' that largest sum across all split options
            res = min(res, current_max)

            # Optimization: If current_sum exceeds our best result,
            # further increasing current_sum won't help.
            if current_sum >= res:
                break
```

```
        memo[state] = res
        return res

    return solve(0, k)

# Example
print(split_array([7, 2, 5, 10, 8], 2)) # Output: 18
```

JavaScript Implementation

```
var splitArray = function(nums, k) {
    const n = nums.length;
    const memo = new Map();

    // Prefix sums to calculate subarray sums in O(1)
    const prefixSums = new Array(n + 1).fill(0);
    for (let i = 0; i < n; i++) {
        prefixSums[i + 1] = prefixSums[i] + nums[i];
    }

    function solve(index, subarraysLeft) {
        if (subarraysLeft === 1) {
            return prefixSums[n] - prefixSums[index];
        }

        const key = `${index}-${subarraysLeft}`;
        if (memo.has(key)) return memo.get(key);

        let minLargestSum = Infinity;

        for (let i = index; i <= n - subarraysLeft; i++) {
            const currentSum = prefixSums[i + 1] - prefixSums[index];
            const remainingMax = solve(i + 1, subarraysLeft - 1);

            const currentMax = Math.max(currentSum, remainingMax);
            minLargestSum = Math.min(minLargestSum, currentMax);

            // Optimization: currentSum only grows, so if it's already
            // worse than minLargestSum, stop.
            if (currentSum >= minLargestSum) break;
        }

        memo.set(key, minLargestSum);
        return minLargestSum;
    }
```

```
    return solve(0, k);
};
```

---

7. Comparison with Other Approaches

- **vs Binary Search (Optimal):** Binary Search on the answer takes **O(n * log(sum of array))**. This is significantly faster than DP's **O(k * n^2)**. However, DP is easier to adapt if the problem constraints change (e.g., if there were specific "costs" associated with each split).
- **vs Brute Force:** Brute force explores every combination of splits, which is exponential **O(2^n)**. DP reduces this by reusing results for shared `(index, k)` states.

---

8. Relevant LeetCode Problems

The "Split array/partition into K groups" pattern using Top-Down DP is common in:

1. **Palindrome Partitioning II (LC 132):** Minimize cuts to make every part a palindrome.
2. **Allocate Books (InterviewBit):** Exactly the same problem logic.
3. **Capacity To Ship Packages Within D Days (LC 1011):** Similar partition logic.
4. **Minimum Difficulty of a Job Schedule (LC 1335):** Very similar "split into k days" logic.

221. Maximal Square

To solve "Maximal Square" like a top-tier Google L5/L6, you don't just jump into the code. You analyze the **substructure** of the problem. A senior engineer looks for the "bottleneck" and the "redundant calculation."

---

1. Problem Explanation

You are given an `m x n` binary matrix filled with `'0'`s and `'1'`s. You need to find the largest **square** containing only `'1'`s and return its **area**.

The "Aha!" Moment

Finding a rectangle is hard, but a **square** is restricted: its height must equal its width. If you find a square of size 3x3, it **must** be composed of smaller

overlapping squares of size 2x2. This "nesting" property is the classic signal for **Dynamic Programming**.

---

2. Solution Explanation (The DP Way)

The most intuitive way to solve this is **Bottom-Up DP**. We want to know: "What is the largest square whose **bottom-right corner** ends at this cell (i, j)?"

The Logic

To form a square of side length K at (i, j), you need the cells to its left, top, and top-left to already be the bottom-right corners of squares of side length $K - 1$.

The Formula: `DP[i][j] = min(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + 1` (if matrix[i][j] == '1')

Step-by-Step Visualization

Imagine this input matrix:

```
1 1 1
1 1 1
1 1 1
```

Step 1: Initialize DP table with zeros

```
[ 0, 0, 0 ]
[ 0, 0, 0 ]
[ 0, 0, 0 ]
```

Step 2: Fill Row 0 and Column 0 (A '1' on the edge can only ever be a 1x1 square)

```
[ 1, 1, 1 ]
[ 1, 0, 0 ]
[ 1, 0, 0 ]
```

Step 3: Process cell (1, 1) Look at its neighbors: Top(1), Left(1), Top-Left(1). Min is 1. So `1 + 1 = 2`.

```
[ 1, 1, 1 ]
[ 1, 2, 0 ]
[ 1, 0, 0 ]
```

Step 4: Process cell (1, 2) Look at neighbors: Top(1), Left(2), Top-Left(1). Min is 1. So `1 + 1 = 2`.

```
[ 1, 1, 1 ]
[ 1, 2, 2 ]
[ 1, 0, 0 ]
```

Step 5: Process cell (2, 2) - The Finale Look at neighbors: Top(2), Left(2), Top-Left(2). Min is 2. So `2 + 1 = 3`.

```
[ 1, 1, 1 ]
[ 1, 2, 2 ]
[ 1, 2, 3 ] <-- Max side found is 3. Area = 3 * 3 = 9.
```

---

3. Time and Space Complexity Analysis

An L6 engineer would immediately point out that we don't need a full 2D DP table. We only ever look at the "previous row" and the "current row." This allows us to optimize Space Complexity from $O(M * N)$ to $O(N)$.

Time Complexity (TC)

```
Total Cells = Rows (M) * Cols (N)
Work per cell = O(1) (Constant time comparison)

TC = O(M * N)
```

Space Complexity (SC)

```
Standard DP = O(M * N) (Full 2D grid)
Optimized DP = O(N) (Only storing one row)

SC = O(N)
```

---

4. Solution Code

Bottom-Up Approach (Most Efficient)

This approach fills the table iteratively. It's usually preferred in interviews for this specific problem because it's easier to optimize for space.

Python (Bottom-Up)

```python
def maximalSquare(matrix):
    if not matrix or not matrix[0]:
        return 0

    rows, cols = len(matrix), len(matrix[0])
    # dp[i][j] represents the side length of the maximum square
    # ending at cell (i, j)
    dp = [[0] * (cols + 1) for _ in range(rows + 1)]
    max_side = 0
```

```
    for r in range(rows):
        for c in range(cols):
            if matrix[r][c] == '1':
                # The core logic: check neighbors to determine the
                # possible square size at the current cell
                dp[r+1][c+1] = min(dp[r][c+1], dp[r+1][c], dp[r][c]) + 1
                max_side = max(max_side, dp[r+1][c+1])

    return max_side * max_side
```

JavaScript (Bottom-Up)

```
/**
 * Bottom-up DP to find the largest square of 1s.
 */
var maximalSquare = function(matrix) {
    let rows = matrix.length;
    let cols = matrix[0].length;
    let dp = Array.from({ length: rows + 1 }, () => Array(cols + 1).fill(0));
    let maxSide = 0;

    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (matrix[r][c] === '1') {
                // dp[r+1][c+1] maps to matrix[r][c] to handle
                // boundary conditions without extra 'if' checks
                dp[r + 1][c + 1] = Math.min(dp[r][c + 1], dp[r + 1][c], dp[r][c]) + 1;
                maxSide = Math.max(maxSide, dp[r + 1][c + 1]);
            }
        }
    }
    return maxSide * maxSide;
};
```

Top-Down Approach (Recursive with Memoization)

This follows the natural thought process: "To solve for this cell, give me the answers for the three neighbors."

Python (Top-Down)

```
def maximalSquare(matrix):
    rows, cols = len(matrix), len(matrix[0])
    memo = {}
```

```
    def solve(r, c):
        # Base Case: Out of bounds
        if r >= rows or c >= cols:
            return 0

        if (r, c) in memo:
            return memo[(r, c)]

        # We must call the recursion for neighbors regardless of current cell
        # to ensure the whole grid is explored
        down = solve(r + 1, c)
        right = solve(r, c + 1)
        diag = solve(r + 1, c + 1)

        if matrix[r][c] == "1":
            memo[(r, c)] = 1 + min(down, right, diag)
        else:
            memo[(r, c)] = 0

        return memo[(r, c)]

    solve(0, 0)
    # The max value could be anywhere in the memo, so we find it
    return max(memo.values())**2 if memo else 0
```

JavaScript (Top-Down)

```
/**
 * Recursive approach with a memoization map to store
 * previously calculated square sizes.
 */
var maximalSquare = function(matrix) {
    const rows = matrix.length;
    const cols = matrix[0].length;
    const memo = new Map();
    let maxSide = 0;

    function solve(r, c) {
        if (r >= rows || c >= cols) return 0;
        const key = `${r},${c}`;
        if (memo.has(key)) return memo.get(key);

        const down = solve(r + 1, c);
        const right = solve(r, c + 1);
        const diag = solve(r + 1, c + 1);
```

```
        let result = 0;
        if (matrix[r][c] === '1') {
            result = 1 + Math.min(down, right, diag);
        }

        maxSide = Math.max(maxSide, result);
        memo.set(key, result);
        return result;
    }

    solve(0, 0);
    return maxSide * maxSide;
};
```

Terminology Table

Term	Meaning	Application to this problem
Overlapping Subproblems	Solving the same small problem multiple times.	Calculating the square size for cell (2,2) requires knowing (1,1), which was already used for (1,2).
Optimal Substructure	The global solution is built from local optimal solutions.	A 3x3 square is only possible if three 2x2 squares overlap perfectly.
Space Optimization	Reducing memory by only keeping necessary states.	Realizing we only need the "previous row" of DP values, not the whole history.

Real World / Interview Variations

1. **Google (Infrastructure):** "Given a 2D map of a data center where some racks are occupied ('1') and some are free ('0'), find the largest square area of free space to install a new high-density cooling unit."
2. **Meta (Ads/UI):** "Find the largest square region in a pixel grid that is entirely one color for a compression algorithm."

3. **Bloomberg (FinTech):** "Given a grid representing price movements, find the largest square period where all stocks remained above a certain threshold (a 'stable' square)."
4. **Advanced Variation:** "Maximal Rectangle" (Leetcode 85). Instead of a square, it can be any rectangle. This is significantly harder and uses a monotonic stack.

152. Maximum Product Subarray

This is a classic "dynamic programming" problem that often trips people up because, unlike the Maximum Sum Subarray (Kadane's Algorithm), a very small negative number can suddenly become a very large positive number when multiplied by another negative.

An L5/L6 engineer at Google would approach this by identifying the **optimal substructure** and the "edge case" of signs.

---

1. Problem Explanation

We are given an integer array `nums`. We need to find a contiguous non-empty subarray that has the largest product and return that product.

The Catch:

- **Positives:** Easy. `2 * 3 = 6`. Products grow.
- **Zeros:** They "reset" the product to 0.
- **Negatives:** This is the tricky part. A single negative number makes the product negative, but a *second* negative number can flip a huge negative back into a huge positive.

Example: `[2, 3, -2, 4, -1]`

- `[2, 3]` product is 6.
- `[2, 3, -2]` product is -12.
- `[2, 3, -2, 4, -1]` product is 48.

The "state" we need to track isn't just the maximum product so far, but also the **minimum** product so far (because a minimum could be a large negative number waiting to become the maximum).

---

2. Solution Explanation

We use a variation of **Kadane's Algorithm**. At each index `i`, we want to know the maximum product ending at that position.

The Core Logic

At any element `nums[i]`, the new maximum/minimum can come from three sources:

1. The number itself (`nums[i]`).
2. The current number times the previous maximum (`nums[i] * prevMax`).
3. The current number times the previous minimum (`nums[i] * prevMin`) — *This handles the "negative * negative = positive" case.*

Step-by-Step Visualization

Let's trace `nums = [2, 3, -2, 4, -1]`

```
Initial State:
res = 2 (first element)
currMax = 2
currMin = 2

Step 1: num = 3
------------------------------------------------------------
Choices for new Max/Min:
A: 3
B: 3 * 2 (prevMax) = 6
C: 3 * 2 (prevMin) = 6

currMax = max(3, 6, 6) = 6
currMin = min(3, 6, 6) = 3
res = max(2, 6) = 6

Visualization:
[ 2,  3, -2,  4, -1 ]
  ^   ^
  |   +-- Current Max Subarray: [2, 3] (Product: 6)
------------------------------------------------------------

Step 2: num = -2
------------------------------------------------------------
Choices:
A: -2
B: -2 * 6 (prevMax) = -12
C: -2 * 3 (prevMin) = -6

currMax = max(-2, -12, -6) = -2
currMin = min(-2, -12, -6) = -12
res = max(6, -2) = 6
```

```
Visualization:
[ 2,  3, -2,  4, -1 ]
          ^
          +-- Current Min Subarray: [2, 3, -2] (Product: -12)
          +-- Current Max Subarray: [-2] (Product: -2)
------------------------------------------------------------

Step 3: num = 4
------------------------------------------------------------
Choices:
A: 4
B: 4 * -2 (prevMax) = -8
C: 4 * -12 (prevMin) = -48

currMax = max(4, -8, -48) = 4
currMin = min(4, -8, -48) = -48
res = max(6, 4) = 6
------------------------------------------------------------

Step 4: num = -1
------------------------------------------------------------
Choices:
A: -1
B: -1 * 4 (prevMax) = -4
C: -1 * -48 (prevMin) = 48  <-- THE FLIP!

currMax = max(-1, -4, 48) = 48
currMin = min(-1, -4, 48) = -4
res = max(6, 48) = 48

Final Result: 48
------------------------------------------------------------
```

Top-Down vs Bottom-Up

- **Bottom-Up (Iterative):** This is the most intuitive for this problem. You process the array once, keeping track of the running "local" max and min. It uses constant space if you only keep the previous variables.
- **Top-Down (Recursive):** We define a function `dp(i)` that returns both the max and min product ending at index `i`. To solve `dp(i)`, we must first solve `dp(i-1)`.

---

3. Time and Space Complexity Analysis

The complexity is derived by looking at how many times we visit each element and how much extra memory we use.

```
TIME COMPLEXITY DERIVATION
-----------------------------------------------------------
Operation                 | Frequency      | Cost
--------------------------|----------------|-------------
Iterate through array     | n times        | O(1)
Max/Min comparisons       | 1 per element  | O(1)
--------------------------|----------------|-------------
Total Time                | n * O(1)       | O(n)

Conclusion: O(n) where n is the length of the array.
-----------------------------------------------------------

SPACE COMPLEXITY DERIVATION
-----------------------------------------------------------
Approach      | Storage Used                    | Complexity
--------------|---------------------------------|---------
Bottom-Up     | 3 variables (max, min, res)     | O(1)
Top-Down      | Recursion stack + Memoization   | O(n)
-----------------------------------------------------------
```

---

4. Solution Code

Python

```python
def maxProduct_bottom_up(nums):
    """
    Iterative approach using constant space.
    We track curr_max and curr_min to handle negative flips.
    """
    if not nums: return 0

    res = nums[0]
    curr_max, curr_min = 1, 1

    for n in nums:
        # If n is 0, it resets curr_max and curr_min to 0 in the next step
        # because n, n*max, and n*min will all be 0.

        # We store curr_max in a temp variable because it's updated
        # before we calculate curr_min.
        temp = curr_max * n
```

```
        curr_max = max(n, temp, curr_min * n)
        curr_min = min(n, temp, curr_min * n)

        res = max(res, curr_max)

    return res

def maxProduct_top_down(nums):
    """
    Recursive approach with Memoization.
    dp(i) returns (max_ending_at_i, min_ending_at_i)
    """
    memo = {}

    def dp(i):
        if i == 0:
            return nums[0], nums[0]
        if i in memo:
            return memo[i]

        prev_max, prev_min = dp(i - 1)

        # Possible values for max/min at current index
        vals = (nums[i], nums[i] * prev_max, nums[i] * prev_min)

        memo[i] = (max(vals), min(vals))
        return memo[i]

    overall_max = nums[0]
    for i in range(len(nums)):
        curr_max, _ = dp(i)
        overall_max = max(overall_max, curr_max)

    return overall_max
```

JavaScript

```
/**
 * Bottom-Up Approach (Iterative)
 * Space: O(1), Time: O(n)
 */
function maxProductBottomUp(nums) {
    if (nums.length === 0) return 0;

    let res = nums[0];
    let currMax = 1;
```

```
    let currMin = 1;

    for (let n of nums) {
        let prevMax = currMax;
        // Calculate max and min by comparing current num,
        // num * prevMax, and num * prevMin
        currMax = Math.max(n, n * prevMax, n * currMin);
        currMin = Math.min(n, n * prevMax, n * currMin);

        res = Math.max(res, currMax);
    }

    return res;
}

/**
 * Top-Down Approach (Recursive with Memoization)
 * Space: O(n), Time: O(n)
 */
function maxProductTopDown(nums) {
    const memo = new Map();

    function solve(i) {
        if (i === 0) return [nums[0], nums[0]];
        if (memo.has(i)) return memo.get(i);

        const [prevMax, prevMin] = solve(i - 1);

        const options = [nums[i], nums[i] * prevMax, nums[i] * prevMin];
        const res = [Math.max(...options), Math.min(...options)];

        memo.set(i, res);
        return res;
    }

    let globalMax = nums[0];
    for (let i = 0; i < nums.length; i++) {
        globalMax = Math.max(globalMax, solve(i)[0]);
    }
    return globalMax;
}
```

Terminology & Real-World Context

Kadane's Algorithm: A greedy/DP approach used to find the maximum sum subarray in linear time. We adapted it here by adding a `min` tracker to account for negative multiplications.

Why this matters in interviews: Google and Meta often use this to test if a candidate thinks about **state**. Most people remember Kadane's for sums, but this tests if you can handle "discontinuous" state changes (like the sign flip).

Indirect / Real-World variants:

1. **Stock Market Volatility (Bloomberg):** Find the period with the highest "cumulative growth" (multiplicative) vs. simple gain (additive).
2. **Exchange Rate Arbitrage (Meta/Google):** If you have a series of exchange rates (e.g., 1.05, 0.98, 1.10), what is the maximum possible conversion ratio you can achieve over a continuous period?
3. **Maximum Product of 3 Numbers:** A variation where the window isn't contiguous, requiring you to track the two smallest negatives and the largest positives.

746. Min Cost Climbing Stairs

A Google L5 (Senior) or L6 (Staff) engineer doesn't just look for the "answer." They look for the **optimal pattern**, **edge cases**, and **memory efficiency**. For this problem, they would immediately recognize it as a "Shortest Path in a DAG" (Directed Acyclic Graph) problem, which is the foundation of Dynamic Programming (DP).

---

1. Problem Explanation

Imagine a staircase. Each step has a "toll" or a cost you must pay to step on it.

- You can start at either **Index 0** or **Index 1**.
- From your current step, you can leap **1 step** forward or **2 steps** forward.
- Your goal: Reach the "Top of the floor" (which is one index *past* the last element of the array) with the **minimum total cost**.

The "Mental Trap"

The "top" isn't the last index of the array; it is the space beyond it. If the array is `[10, 15, 20]`, the indices are 0, 1, and 2. The "top" is **Index 3**.

---

2. Solution Explanation

An L5+ engineer would likely start with the **Bottom-Up** approach because it is easier to optimize for space.

The Intuition: Working Backwards

To stand on the "Top," you must have come from either the **last step** or the **second-to-last step**. Therefore, the cost to reach the top is: `Min(Cost to reach last step, Cost to reach second-to-last step)`

Step-by-Step Visualization (Bottom-Up)

Let's use the example: `cost = [10, 15, 20]`

Step 1: The Setup We create a `min_cost` array of size `n + 1`.

```
Indices:      0      1      2      3 (Top)
Cost:       [10]   [15]   [20]    [0]
Min_Cost:   [ ?]   [ ?]   [ ?]    [ ?]
```

Step 2: Base Cases The cost to stay at index 0 is just 10. The cost to stay at index 1 is 15.

```
Min_Cost:   [10]   [15]   [ ?]    [ ?]
```

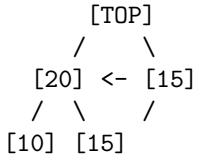
Step 3: Calculating Index 2 To get to Index 2, I could come from Index 0 (10) or Index 1 (15). 10 is cheaper. So, `10 + 20 (current step cost) = 30`.

```
Calculation: min(Min_Cost[0], Min_Cost[1]) + cost[2]
             min(10, 15) + 20 = 30

Min_Cost:   [10]   [15]   [30]    [ ?]
```

Step 4: Reaching the Top (Index 3) To reach the top, you don't pay a "top" cost; you just need to land there from Index 1 or Index 2.

```
Calculation: min(Min_Cost[1], Min_Cost[2])
             min(15, 30) = 15

Min_Cost:   [10]   [15]   [30]    [15]
                                   ^-- Final Answer
```

Visualization of the State Transitions



The "Staff Engineer" Optimization (Space)

We only ever look at the **last two values**. We don't need the whole array. We can use two variables: `down_one` and `down_two`.

```
Initial:
[10, 15, 20]
  ^   ^
 d2  d1

Step 1 (Looking at index 2):
current = cost[2] + min(d1, d2) -> 20 + min(15, 10) = 30
Update: d2 = d1 (15), d1 = current (30)

Final Step (Top):
return min(d1, d2) -> min(30, 15) = 15
```

---

3. Complexity Analysis

Time Complexity (TC)

We iterate through the array exactly once.

```
Derivation:
Array length = N
Operations per step = Constant (O(1) for addition and min)
Total = N * O(1)
TC = O(N)
```

Space Complexity (SC)

For the optimized Bottom-Up approach:

```
Derivation:
Variable 1 (down_one)  = O(1)
Variable 2 (down_two)  = O(1)
Variable 3 (temp/curr) = O(1)
Total = O(1)
SC = O(1)
```

---

4. Solution Code

Top-Down (Recursive with Memoization)

This approach mimics how a human thinks: "To solve the top, solve the sub-problems first."

Python

```python
def minCostClimbingStairs(cost):
    memo = {}

    # Recursive function to find the minimum cost to reach step 'i'
    def solve(i):
        # Base cases: If we are at the starting steps, cost is just the step cost
        if i <= 1:
            return cost[i]
        if i in memo:
            return memo[i]

        # Recursive step: current cost + min of previous two steps
        memo[i] = cost[i] + min(solve(i - 1), solve(i - 2))
        return memo[i]

    n = len(cost)
    # The top can be reached from the last or second-to-last step
    return min(solve(n - 1), solve(n - 2))
```

Javascript

```javascript
function minCostClimbingStairs(cost) {
    const memo = new Map();

    /**
     * Recursively calculates the cost to reach and step on index i
     */
    function solve(i) {
        if (i <= 1) return cost[i];
        if (memo.has(i)) return memo.get(i);

        const result = cost[i] + Math.min(solve(i - 1), solve(i - 2));
        memo.set(i, result);
        return result;
    }

    const n = cost.length;
    return Math.min(solve(n - 1), solve(n - 2));
}
```

Bottom-Up (Iterative - Space Optimized)

This is the “Gold Standard” for a coding interview.

Python

```python
def minCostClimbingStairs(cost):
    # Two variables to track the min cost of the two steps below us
    down_two = cost[0]
    down_one = cost[1]

    # Start from index 2 up to the last index
    for i in range(2, len(cost)):
        # Minimum cost to reach and step on current stair i
        current = cost[i] + min(down_one, down_two)
        down_two = down_one
        down_one = current

    # The answer is the minimum of the last two calculated costs
    return min(down_one, down_two)
```

Javascript

```javascript
function minCostClimbingStairs(cost) {
    let downTwo = cost[0];
    let downOne = cost[1];

    /**
     * Iteratively build the solution from the ground up.
     * We only store the last two results to save memory.
     */
    for (let i = 2; i < cost.length; i++) {
        let current = cost[i] + Math.min(downOne, downTwo);
        downTwo = downOne;
        downOne = current;
    }

    return Math.min(downOne, downTwo);
}
```

Terminology Note: Dynamic Programming (DP)

DP is a method for solving complex problems by breaking them down into simpler sub-problems. It is applicable here because the problem has **Overlapping Sub-problems** (to calculate step 4, you need step 3 and 2; to calculate step 5, you need step 4 and 3—step 3 is reused). Using DP avoids re-calculating the same values, which would otherwise lead to an exponential time complexity.

Real-World / Indirect Interview Variants

Companies like Google and Bloomberg rarely ask the "staircase" directly. They disguise it as:

1. **Server Hopper:** You have a list of servers with different latency costs. You can skip one server or hop to the next. Find the path with minimum total latency.
2. **Video Game Health:** A character moves across a map where each tile costs health. The character can jump 1 or 2 tiles. What's the minimum health lost to reach the exit?
3. **Bloomberg Finance Variant:** Calculating the minimum transaction fees for a series of trades where you can bundle trades in groups of 1 or 2.

72. Edit Distance

For a top-tier engineer at a company like Google, solving "Edit Distance" (Levenshtein Distance) isn't just about coding the DP table—it's about articulating the **recursive structure** of the problem and identifying why it behaves the way it does.

---

1. Problem Explanation

The goal is to find the **minimum number of operations** required to convert `word1` to `word2`. You have three allowed operations:

1. **Insert** a character.
2. **Delete** a character.
3. **Replace** a character.

The "Mental Model"

Imagine two pointers, `i` and `j`, at the end of `word1` and `word2`.

- If the characters at `word1[i]` and `word2[j]` match, we do nothing and move both pointers back.
- If they don't match, we must choose the "cheapest" path among inserting, deleting, or replacing.

---

2. Solution Explanation (Top-Down Intuition)

A Google L5/L6 would likely start with the **Top-Down (Recursive + Memoization)** approach because it directly mirrors the way we think about the problem.

The Logic Breakdown

Let's say we are comparing `word1 = "horse"` and `word2 = "ros"`.

Case A: Characters Match If we compare the 's' in "horse" and the 's' in "ros", they match. The cost is 0. We just need to find the edit distance for "hor" and "ro".

Case B: Characters Don't Match If we compare 'e' (from horse) and 's' (from ros), we have three choices:

1. **Replace:** Change 'e' to 's'. Now they match. We move both pointers.
2. **Delete:** Remove 'e' from `word1`. Now we compare "hors" to "ros".
3. **Insert:** Add 's' to the end of `word1`. Now the 's' matches, and we still need to compare "horse" to "ro".

ASCII Visualization: The Recursive Tree

```
Target: "ros" (j)
Source: "horse" (i)

Solve("horse", "ros")
  |
  |-- Match? 'e' != 's'
  |    |
  |    |-- Replace: 1 + Solve("hors", "ro")
  |    |-- Delete:  1 + Solve("hors", "ros")
  |    |-- Insert:  1 + Solve("horse", "ro")
```

Conversion to Bottom-Up (Tabulation)

To avoid the overhead of recursion, we use a 2D grid where `dp[i][j]` represents the edit distance between `word1[0...i]` and `word2[0...j]`.

Step 1: The Base Cases If `word1` is empty, the distance is the length of `word2` (all insertions). If `word2` is empty, the distance is the length of `word1` (all deletions).

```
Initial Table (word1="horse", word2="ros"):
      ""  r   o   s
  ""  0   1   2   3  <-- To make "" into "ros", insert 3 chars
  h   1   .   .   .
  o   2   .   .   .
  r   3   .   .   .
  s   4   .   .   .
  e   5   .   .   .
```

Step 2: Filling the Grid For each cell `(i, j)`, if characters don't match: `dp[i][j] = 1 + min(Top, Left, Diagonal)`

```
Visualizing the neighbors for dp[i][j]:
+-----------+-----------+
| Diagonal  |    Top    |  (Diagonal = Replace)
| (i-1,j-1) |  (i-1, j) |  (Top = Delete)
+-----------+-----------+
|   Left    |  CURRENT  |  (Left = Insert)
|  (i, j-1) |   (i, j)  |
+-----------+-----------+
```

---

3. Complexity Analysis

Time Complexity (TC)

We iterate through every cell in a 2D matrix of size `M * N` (where M and N are lengths of the words). Each cell takes constant time `O(1)` to compute.

```
TC Derivation:
Rows (M) * Columns (N) * Constant Work per cell
= O(M * N)
```

Space Complexity (SC)

We store the results in a 2D array of size `M * N`.

```
SC Derivation:
Matrix Storage = (M + 1) * (N + 1)
= O(M * N)
```

(Note: An L6 would mention that SC can be optimized to O(N) since we only ever need the current and previous row).

---

4. Solution Code

Python (Top-Down and Bottom-Up)

```python
class Solution:
    # --- Top-Down Approach (Memoization) ---
    def minDistanceTopDown(self, word1: str, word2: str) -> int:
        memo = {}

        # solve(i, j) returns the min distance for word1[0...i] and word2[0...j]
        def solve(i, j):
            if (i, j) in memo: return memo[(i, j)]

            # Base Cases: if one string is exhausted, return length of the other
            if i == 0: return j
```

```
            if j == 0: return i

            if word1[i-1] == word2[j-1]:
                res = solve(i-1, j-1)
            else:
                # 1 + min of (Insert, Delete, Replace)
                res = 1 + min(
                    solve(i, j-1),    # Insert
                    solve(i-1, j),    # Delete
                    solve(i-1, j-1)   # Replace
                )
            memo[(i, j)] = res
            return res

        return solve(len(word1), len(word2))

    # --- Bottom-Up Approach (Tabulation) ---
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        # dp[i][j] will hold distance between word1[:i] and word2[:j]
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(m + 1):
            for j in range(n + 1):
                if i == 0:
                    dp[i][j] = j
                elif j == 0:
                    dp[i][j] = i
                elif word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    # Choose the minimum cost among the three operations
                    dp[i][j] = 1 + min(dp[i-1][j],    # Delete
                                       dp[i][j-1],    # Insert
                                       dp[i-1][j-1])  # Replace
        return dp[m][n]
```

JavaScript (Top-Down and Bottom-Up)

```
/**
 * Bottom-Up Tabulation
 */
var minDistance = function(word1, word2) {
    const m = word1.length;
    const n = word2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));
```

```
    for (let i = 0; i <= m; i++) {
        for (let j = 0; j <= n; j++) {
            // Base case: empty strings
            if (i === 0) {
                dp[i][j] = j;
            } else if (j === 0) {
                dp[i][j] = i;
            } else if (word1[i - 1] === word2[j - 1]) {
                // Characters match, no new operation needed
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                // Take 1 + minimum of top, left, and diagonal cells
                dp[i][j] = 1 + Math.min(
                    dp[i - 1][j],    // Delete
                    dp[i][j - 1],    // Insert
                    dp[i - 1][j - 1] // Replace
                );
            }
        }
    }
    return dp[m][n];
};

/**
 * Top-Down Memoization
 */
var minDistanceTopDown = function(word1, word2) {
    const memo = new Map();

    function solve(i, j) {
        const key = `${i},${j}`;
        if (memo.has(key)) return memo.get(key);

        if (i === 0) return j;
        if (j === 0) return i;

        let res;
        if (word1[i - 1] === word2[j - 1]) {
            res = solve(i - 1, j - 1);
        } else {
            res = 1 + Math.min(
                solve(i, j - 1),    // Insert
                solve(i - 1, j),    // Delete
                solve(i - 1, j - 1) // Replace
            );
```

```
        }
        memo.set(key, res);
        return res;
    }

    return solve(word1.length, word2.length);
};
```

Terminology Note

Dynamic Programming (DP): A method for solving complex problems by breaking them down into simpler subproblems. It "remembers" the results of subproblems (memoization or tabulation) so it doesn't have to re-calculate them. In this problem, it helps because many different "edit paths" lead to comparing the same substrings.

Real-World / Interview Variants

- **Google:** "Spell Checker Suggestion" – When a user types a query, find words in the dictionary with the smallest edit distance.
- **Meta:** "Diff Tool" – Similar to `git diff`, identifying the minimum changes to turn one version of a file into another.
- **Bloomberg:** "Natural Language Processing" – Measuring the similarity between two financial entity names that might have typos (e.g., "Goldman Sachs" vs "Gollman Sach").

97. Interleaving String

An L5 or L6 engineer at Google doesn't just look for *a* solution; they look for the **structure** of the problem. They want to know why a simple greedy approach fails and how to build a robust state machine or recurrence relation to handle the complexity.

---

1. Problem Explanation

The goal is to determine if a string `s3` is formed by "interleaving" `s1` and `s2`.

The Rules:

- You must maintain the relative order of characters from `s1` and `s2`.
- You are essentially "weaving" them together.

Why it's tricky: If you have `s1 = "aab"`, `s2 = "aac"`, and `s3 = "aacaab"`, you might be tempted to take the first 'a' from `s1` or `s2`. A **Greedy** algorithm

(just picking whichever matches) will fail because a choice you make now might "starve" the solution later.

---

2. Solution Explanation (The DP Way)

This is a classic **Dynamic Programming** problem because it has "Optimal Substructure" and "Overlapping Subproblems."

The Core Logic (Top-Down Intuition)

Think of it as a pathfinding problem. You are at index `i` in `s1` and index `j` in `s2`. Your current position in `s3` is always `i + j`.

1. If `s1[i]` matches `s3[i+j]`, you *could* move forward in `s1`.
2. If `s2[j]` matches `s3[i+j]`, you *could* move forward in `s2`.
3. If both match, you must try **both** paths. If either path leads to the end, the answer is True.

Visualizing the Decision Tree (ASCII)

Let `s1 = "a"`, `s2 = "b"`, `s3 = "ab"`

```
Position: (i, j) where i is index in s1, j is index in s2
Target: s3[i+j]

State: (0, 0) -> s3[0] is 'a'
  |
  |-- Choice 1: Use s1[0] ('a')? Yes, matches! -> Move to (1, 0)
  |     |
  |     State: (1, 0) -> s3[1] is 'b'
  |       |
  |       |-- Choice: Use s2[0] ('b')? Yes! -> Move to (1, 1)
  |             |
  |             Result: SUCCESS (Both strings exhausted)
  |
  |-- Choice 2: Use s2[0] ('b')? No, 'b' != 'a'. -> DEAD END
```

The 2D DP Table (Bottom-Up)

We create a table `dp[i][j]` representing: "Can we form `s3[0...i+j-1]` using `s1[0...i-1]` and `s2[0...j-1]`?"

Let `s1 = "aab"`, `s2 = "axy"`, `s3 = "aaxaby"`

Step-by-Step Table Construction:

```
s1 = "aab", s2 = "axy", s3 = "aaxaby"
Table size: (len(s1)+1) x (len(s2)+1)
```

```
Initial State (Empty strings interleave to form empty s3):
      ""   a   x   y  (s2)
""   [T]  [ ] [ ] [ ]
a    [ ]  [ ] [ ] [ ]
a    [ ]  [ ] [ ] [ ]
b    [ ]  [ ] [ ] [ ]
(s1)

Base Case: Filling the first row (Using ONLY s2 to make s3)
      ""   a   x   y
""   [T]  [T] [F] [F]  <- "a" matches s3[0], but "ax" doesn't match s3[0:2] "aa"
a    [ ]  [ ] [ ] [ ]
a    [ ]  [ ] [ ] [ ]
b    [ ]  [ ] [ ] [ ]

Base Case: Filling the first column (Using ONLY s1 to make s3)
      ""   a   x   y
""   [T]  [T] [F] [F]
a    [T]  [ ] [ ] [ ]  <- s1[0] 'a' matches s3[0]
a    [T]  [ ] [ ] [ ]  <- s1[0:2] 'aa' matches s3[0:2]
b    [F]  [ ] [ ] [ ]  <- s1[0:3] 'aab' DOES NOT match s3[0:3] 'aax'

Final Logic for dp[i][j]:
dp[i][j] = (dp[i-1][j] AND s1[i-1] == s3[i+j-1]) OR
           (dp[i][j-1] AND s2[j-1] == s3[i+j-1])
```

---

3. Time and Space Complexity Analysis

Time Complexity (TC)

We visit every cell in the 2D matrix exactly once. Each cell takes constant time `O(1)` to calculate.

```
Rows (N) = Length of s1 + 1
Cols (M) = Length of s2 + 1

Total Operations = N * M
TC = O(N * M)
```

Space Complexity (SC)

We store the results in a 2D table.

```
Memory used by Grid:
[ ][ ][ ]... (M columns)
```

```
[ ][ ][ ]...
... (N rows)
```

`SC = O(N * M)`

Note: An L6 engineer would point out that we only need the "previous row" to calculate the "current row," allowing us to optimize Space to `O(min(N, M))`.

---

4. Solution Code

Top-Down (Memoization)

This is usually the most intuitive "interviewer-friendly" approach.

Python

```python
def isInterleave(s1, s2, s3):
    if len(s1) + len(s2) != len(s3):
        return False

    memo = {}

    # helper function explores the decision tree recursively
    def solve(i, j):
        # Base case: if both indices reach the end, we found a valid interleaving
        if i == len(s1) and j == len(s2):
            return True
        if (i, j) in memo:
            return memo[(i, j)]

        ans = False
        # If s1 has chars left and matches s3's current char, try moving i
        if i < len(s1) and s1[i] == s3[i + j]:
            ans = solve(i + 1, j)

        # If s2 has chars left and matches s3's current char, try moving j
        # We use 'or' because either path being true makes the whole thing true
        if not ans and j < len(s2) and s2[j] == s3[i + j]:
            ans = solve(i, j + 1)

        memo[(i, j)] = ans
        return ans

    return solve(0, 0)
```

JavaScript

```javascript
var isInterleave = function(s1, s2, s3) {
    if (s1.length + s2.length !== s3.length) return false;
    const memo = new Map();

    /**
     * Recursive helper with memoization key "i,j"
     */
    function solve(i, j) {
        if (i === s1.length && j === s2.length) return true;
        const key = `${i},${j}`;
        if (memo.has(key)) return memo.get(key);

        let res = false;
        // Try match s1
        if (i < s1.length && s1[i] === s3[i + j]) {
            res = solve(i + 1, j);
        }
        // Try match s2 (only if res is still false)
        if (!res && j < s2.length && s2[j] === s3[i + j]) {
            res = solve(i, j + 1);
        }

        memo.set(key, res);
        return res;
    }

    return solve(0, 0);
};
```

Bottom-Up (Tabulation)

Python

```python
def isInterleave(s1, s2, s3):
    N, M = len(s1), len(s2)
    if N + M != len(s3):
        return False

    # dp[i][j] means s1[:i] and s2[:j] interleave to form s3[:i+j]
    dp = [[False] * (M + 1) for _ in range(N + 1)]
    dp[0][0] = True

    for i in range(N + 1):
        for j in range(M + 1):
            # Check if we can reach this state by using a char from s1
```

```
            if i > 0:
                dp[i][j] = dp[i][j] or (dp[i-1][j] and s1[i-1] == s3[i+j-1])
            # Check if we can reach this state by using a char from s2
            if j > 0:
                dp[i][j] = dp[i][j] or (dp[i][j-1] and s2[j-1] == s3[i+j-1])

    return dp[N][M]
```

JavaScript

```
var isInterleave = function(s1, s2, s3) {
    const N = s1.length, M = s2.length;
    if (N + M !== s3.length) return false;

    const dp = Array.from({ length: N + 1 }, () => Array(M + 1).fill(false));
    dp[0][0] = true;

    for (let i = 0; i <= N; i++) {
        for (let j = 0; j <= M; j++) {
            // Update current state based on previous possible moves
            if (i > 0 && s1[i-1] === s3[i + j - 1]) {
                dp[i][j] = dp[i][j] || dp[i-1][j];
            }
            if (j > 0 && s2[j-1] === s3[i + j - 1]) {
                dp[i][j] = dp[i][j] || dp[i][j-1];
            }
        }
    }
    return dp[N][M];
};
```

Terminology & Real-World Application

Overlapping Subproblems: This means we solve the same "state" (e.g., index `i=2, j=3`) multiple times through different paths. Memoization saves the result so we only calculate it once.

Real-World/Interview Context:

- **Google:** Often asks this as a "Version Control" problem. Imagine two developers merging branches. Does the final master branch represent a valid sequence of their individual commits?
- **Meta:** Has used this in "Sequence Alignment" contexts (similar to genetics or diff-checkers) where you need to see if a combined log stream could have been produced by two independent servers.

- **Bloomberg:** Frequently focuses on the **Space Optimization** (reducing the 2D array to a 1D array) to test if the candidate understands the memory layout of DP.

63. Unique Paths II

An L5 or L6 engineer at Google doesn't just look for "the answer"; they look for the most robust, memory-efficient, and readable way to handle edge cases. For a problem like **Unique Paths II**, they see it as a classic "Grid DP" (Dynamic Programming) problem where the primary challenge isn't the math—it's handling the "noise" (the obstacles).

---

1. Problem Explanation

Imagine a robot on a grid. It starts at the top-left corner `(0, 0)` and wants to reach the bottom-right corner.

- **The Catch:** There are obstacles (represented by `1`) that the robot cannot pass through. Empty spaces are `0`.
- **The Rule:** The robot can only move **Down** or **Right**.
- **The Goal:** Count the total number of unique paths from start to finish.

Why is this DP?

At any given cell `(r, c)`, the robot could have only arrived from two places:

1. The cell above it: `(r-1, c)`
2. The cell to its left: `(r, c-1)`

Therefore: `Paths to (r, c) = Paths from Top + Paths from Left`.

---

2. Solution Explanation (The Evolution)

Phase 1: The Intuitive Top-Down Approach (Recursive)

Top-down is like asking the finish line: "How many ways can I get here?" The finish line asks its neighbors, and so on, until we hit the start.

Visualizing the logic (3x3 grid with obstacle at 1,1):

```
Grid Map:
[S][0][0]  S = Start (0,0)
[0][X][0]  X = Obstacle (1,1)
[0][0][E]  E = End (2,2)

Recursive Tree (Simplified):
```

```
            Count(2,2)
           /          \
      Count(1,2)      Count(2,1)
       /     \          /     \
Count(0,2) [OBSTACLE] [OBSTACLE] Count(2,0)
```

Phase 2: The Bottom-Up Approach (Iterative)

This is usually preferred by senior engineers for production because it avoids "Stack Overflow" errors on massive grids. We build a `dp` table of the same size.

Step-by-Step ASCII Walkthrough (3x3 Grid): `Grid: [[0,0,0], [0,1,0], [0,0,0]]`

Step 1: Initialize DP table with 0s.

```
DP Table:
0  0  0
0  0  0
0  0  0
```

Step 2: Handle the Start. If `grid[0][0]` isn't an obstacle, set `dp[0][0] = 1`.

```
1  0  0
0  0  0
0  0  0
```

Step 3: Fill the first row and first column. A robot can only reach `(0,1)` if `(0,0)` was reachable and `(0,1)` isn't blocked. If there's an obstacle, the rest of that row/column becomes unreachable (0).

```
1  1  1
1  0  0
1  0  0
```

Step 4: Fill the rest using `dp[r][c] = dp[r-1][c] + dp[r][c-1]`. Wait! If `grid[r][c] == 1`, we must set `dp[r][c] = 0`.

```
Processing (1,1): It's an obstacle! Set to 0.
1  1  1
1  0  0
1  0  0

Processing (1,2): Top is 1, Left is 0. 1 + 0 = 1.
1  1  1
1  0  1
1  0  0

Processing (2,1): Top is 0, Left is 1. 0 + 1 = 1.
```

```
1  1  1
1  0  1
1  1  0

Processing (2,2) [END]: Top is 1, Left is 1. 1 + 1 = 2.
1  1  1
1  0  1
1  1  2 <--- Result!
```

---

3. Complexity Analysis

We derive complexity by looking at how many "rooms" we visit and how much "extra paper" we use.

Time Complexity (TC)

```
Grid Dimensions: M rows, N columns
Total Cells: M * N

Operation per cell:
- Check if obstacle (O(1))
- Addition of two numbers (O(1))

Derivation:
M * N cells * O(1) work = O(M * N)
```

Space Complexity (SC)

```
Standard DP Table:
Requires a 2D array of size M * N.
Space = O(M * N)

Optimized (L5/L6 Level):
Since we only ever look at the "current row" and the "previous row",
we can actually use a single array of size N.
Space = O(N)
```

---

4. Solution Code

Top-Down (Memoization)

This approach uses a "cache" to remember results we've already calculated.

Python (Top-Down)

```python
def uniquePathsWithObstacles(obstacleGrid):
    rows = len(obstacleGrid)
    cols = len(obstacleGrid[0])
    memo = {}

    # dfs(r, c) calculates paths from (r, c) to the end
    def dfs(r, c):
        # Base Case: Out of bounds or hitting an obstacle
        if r >= rows or c >= cols or obstacleGrid[r][c] == 1:
            return 0
        # Base Case: Reached the goal
        if r == rows - 1 and c == cols - 1:
            return 1
        # Check cache
        if (r, c) in memo:
            return memo[(r, c)]

        # Recursive step: Sum of going Right and going Down
        memo[(r, c)] = dfs(r + 1, c) + dfs(r, c + 1)
        return memo[(r, c)]

    return dfs(0, 0)
```

JavaScript (Top-Down)

```javascript
const uniquePathsWithObstacles = function(obstacleGrid) {
    const rows = obstacleGrid.length;
    const cols = obstacleGrid[0].length;
    const memo = new Map();

    /**
     * Recursive helper that uses a string key for the memo map
     * to store results of sub-problems.
     */
    function dfs(r, c) {
        if (r >= rows || c >= cols || obstacleGrid[r][c] === 1) return 0;
        if (r === rows - 1 && c === cols - 1) return 1;

        const key = r + ',' + c;
        if (memo.has(key)) return memo.get(key);

        const result = dfs(r + 1, c) + dfs(r, c + 1);
        memo.set(key, result);
        return result;
```

```
    }

    return dfs(0, 0);
};
```

Bottom-Up (Tabulation - Space Optimized)

This is the "Pro" version. Instead of a full 2D grid, we use one 1D array representing the current row.

Python (Bottom-Up)

```python
def uniquePathsWithObstacles(obstacleGrid):
    if not obstacleGrid or obstacleGrid[0][0] == 1:
        return 0

    cols = len(obstacleGrid[0])
    # dp[c] stores the number of ways to reach column 'c' in the current row
    dp = [0] * cols
    dp[0] = 1

    for r in range(len(obstacleGrid)):
        for c in range(cols):
            if obstacleGrid[r][c] == 1:
                # If obstacle, no paths can pass through here
                dp[c] = 0
            elif c > 0:
                # Paths to this cell = paths from above (already in dp[c])
                # + paths from left (newly updated dp[c-1])
                dp[c] += dp[c-1]

    return dp[-1]
```

JavaScript (Bottom-Up)

```javascript
const uniquePathsWithObstacles = function(obstacleGrid) {
    if (obstacleGrid[0][0] === 1) return 0;

    const cols = obstacleGrid[0].length;
    const dp = new Array(cols).fill(0);
    dp[0] = 1;

    /**
     * Iteratively update the DP array.
     * dp[c] effectively acts as the 'Top' neighbor until it is updated,
```

```
     * at which point it becomes the 'Left' neighbor for the next column.
     */
    for (let r = 0; r < obstacleGrid.length; r++) {
        for (let c = 0; c < cols; c++) {
            if (obstacleGrid[r][c] === 1) {
                dp[c] = 0;
            } else if (c > 0) {
                dp[c] += dp[c - 1];
            }
        }
    }

    return dp[cols - 1];
};
```

Terminology Checklist

- **Dynamic Programming (DP):** A method for solving complex problems by breaking them down into simpler sub-problems. It "remembers" answers to avoid re-calculating them.
- **Memoization (Top-Down):** Storing results of expensive function calls and returning the cached result when the same inputs occur again.
- **Tabulation (Bottom-Up):** Filling a table (array) based on previously computed values, starting from the smallest possible sub-problem.
- **Space Optimization:** Recognizing that you only need the previous state (the row above) to calculate the current state, allowing you to reduce O(M*N) space to O(N).

Real-World Interview Variations

Companies like Google and Bloomberg rarely ask "Unique Paths" verbatim. They disguise it in scenarios like:

1. **Google (Map Routing):** "Given a city grid with construction zones (obstacles), find how many ways an autonomous delivery bot can reach a customer."
2. **Meta (UI Layout):** "A newsfeed needs to arrange items in a grid. Some slots are reserved for ads (obstacles). Calculate possible configurations for text blocks."
3. **Bloomberg (Trading Systems):** "A trade order must pass through several validation gates. Some gates are currently 'locked' (obstacles). How many sequences of validation steps can be taken?"

4. **The "Max Gold" Twist:** "Instead of 1s and 0s, each cell has a dollar value. Find the path that collects the *most* money while avoiding obstacles." (This combines Unique Paths logic with Max Path Sum).

122. Best Time to Buy and Sell Stock II

An L5/L6 (Senior/Staff) engineer at Google doesn't just look for a "solution"—they look for the most efficient, readable, and scalable approach. For this specific problem, while it can be solved with complex Dynamic Programming, a high-level engineer would identify the **Greedy** property immediately to achieve the most optimal result.

---

1. Problem Explanation

Imagine you have an array `prices` where `prices[i]` is the price of a given stock on day `i`. You want to maximize your profit. Unlike the first "Buy and Sell Stock" problem, here you can buy and sell **multiple times**.

The Constraints:

- You can only hold **one** share at a time (you must sell before you buy again).
- You can buy and sell on the same day (though in this specific problem, that results in 0 profit).

The Core Goal: Find every "upward slope" in the price graph and capture that profit.

---

2. Solution Explanation

The Intuition: "The Valley-Peak Crawler"

A Senior Engineer visualizes the stock prices as a mountain range. To get the maximum profit, you simply want to be "in" the market whenever the price is going up and "out" when it's going down.

ASCII Visualization 1: The Price Graph Let's look at `prices = [7, 1, 5, 3, 6, 4]`

```
Price
  ^
7 |  * (Day 0)
6 |              * (Day 4)
5 |        * (Day 2)
4 |                    * (Day 5)
```

```
3 |              * (Day 3)
2 |
1 |  * (Day 1)
  +---------------------------> Time
     0  1  2  3  4  5
```

Step-by-Step Walkthrough (The Greedy Approach) The most efficient way to solve this is to realize that a long climb (e.g., buying at 1 and selling at 6) is mathematically identical to buying and selling at every tiny step along the way.

Example 1: Multiple Small Profits `prices = [1, 2, 3, 4]`

- Buy at 1, Sell at 2 (Profit = 1)
- Buy at 2, Sell at 3 (Profit = 1)
- Buy at 3, Sell at 4 (Profit = 1)
- **Total Profit = 3**
- *Note: This is the same as Buying at 1 and Selling at 4 (4 - 1 = 3).*

ASCII Visualization 2: The Logic Flow We iterate through the array starting from the second element. If the current price is higher than yesterday's, we "take" that profit.

```
Prices: [7,  1,  5,  3,  6,  4]
Index:   0   1   2   3   4   5

Step 1 (Day 1): 1 < 7. No profit.
      [7 -> 1]  \ (Down)

Step 2 (Day 2): 5 > 1. PROFIT!
      [1 -> 5]  / (Up)  Profit += (5 - 1) = 4

Step 3 (Day 3): 3 < 5. No profit.
      [5 -> 3]  \ (Down)

Step 4 (Day 4): 6 > 3. PROFIT!
      [3 -> 6]  / (Up)  Profit += (6 - 3) = 3

Step 5 (Day 5): 4 < 6. No profit.
      [6 -> 4]  \ (Down)

Total Accumulated Profit: 4 + 3 = 7
```

---

3. Time and Space Complexity Analysis

An L6 engineer prioritizes efficiency. This solution is `O(n)` because we touch each price exactly once.

Time Complexity (TC)

```
Iteration:  1st pass through N elements
Operation:  Simple comparison and addition (O(1))
Total:      O(N)
            (Where N is the number of days/prices)
```

Space Complexity (SC)

```
Variables:  Only one integer (TotalProfit)
Storage:    No extra arrays or recursion stacks
Total:      O(1)
            (Constant Space)
```

---

4. Solution Code

While the Greedy approach is the "production-ready" answer, a Google interview might ask you to explain it via **Dynamic Programming (DP)** to test your depth.

Approach A: Top-Down (Memoization)

This approach mimics how a human thinks: "On day `i`, should I buy, sell, or do nothing?"

Python (Top-Down)

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        # memo stores (day_index, holding_stock_boolean)
        memo = {}

        # solve(index, holding) calculates max profit from 'index' onwards
        def solve(i, holding):
            if i == len(prices):
                return 0
            if (i, holding) in memo:
                return memo[(i, holding)]

            # Option 1: Do nothing today
            res = solve(i + 1, holding)
```

```
            if holding:
                # Option 2: Sell the stock (add current price to profit)
                res = max(res, prices[i] + solve(i + 1, False))
            else:
                # Option 2: Buy the stock (subtract current price from profit)
                res = max(res, -prices[i] + solve(i + 1, True))

            memo[(i, holding)] = res
            return res

        return solve(0, False)
```

Javascript (Top-Down)

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function(prices) {
    const memo = new Map();

    // Recursive function to explore buying/selling/skipping
    function solve(i, holding) {
        if (i === prices.length) return 0;
        const key = `${i}-${holding}`;
        if (memo.has(key)) return memo.get(key);

        // Skip current day
        let res = solve(i + 1, holding);

        if (holding) {
            // Sell: gain money + move to next day not holding
            res = Math.max(res, prices[i] + solve(i + 1, false));
        } else {
            // Buy: lose money + move to next day holding
            res = Math.max(res, -prices[i] + solve(i + 1, true));
        }

        memo.set(key, res);
        return res;
    }

    return solve(0, false);
};
```

Approach B: Bottom-Up (Tabulation)

This is the most "standard" DP approach, building a table of states.

Python (Bottom-Up)

```python
def maxProfit(prices):
    n = len(prices)
    if n == 0: return 0

    # dp[i][0] = max profit on day i NOT holding a stock
    # dp[i][1] = max profit on day i HOLDING a stock
    dp = [[0] * 2 for _ in range(n)]

    # Base case: Day 0
    dp[0][0] = 0
    dp[0][1] = -prices[0]

    for i in range(1, n):
        # We don't hold stock today if:
        # 1. We didn't hold it yesterday
        # 2. We held it yesterday and sold it today
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

        # We hold stock today if:
        # 1. We held it yesterday
        # 2. We didn't hold it yesterday and bought it today
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])

    return dp[n-1][0]
```

Javascript (Bottom-Up)

```javascript
var maxProfit = function(prices) {
    const n = prices.length;
    if (n === 0) return 0;

    // Use two variables instead of a full 2D array to optimize space (O(1))
    // cash: max profit if we don't own stock
    // hold: max profit if we own stock
    let cash = 0;
    let hold = -prices[0];

    for (let i = 1; i < n; i++) {
        let prevCash = cash;
```

```
        cash = Math.max(cash, hold + prices[i]);
        hold = Math.max(hold, prevCash - prices[i]);
    }

    return cash;
};
```

Terminology: The Greedy Algorithm

What it is: An algorithm that makes the locally optimal choice at each step with the hope of finding a global optimum. **Why it helps here:** In the "Stock II" problem, you are allowed to buy/sell instantly. Since any large profit over several days is just the sum of the daily price increases, capturing every single positive daily difference (local optimum) results in the maximum possible total profit (global optimum).

Real-World Interview Variations

Big Tech companies often disguise this problem to see if you can map it to the "Stock" pattern:

1. **The Server Load Problem (Google):** "You are given a list of projected server costs per hour for the next 24 hours. You can switch between two providers, but you can only be signed up for one at a time. Minimize your cost." (This is the inverse of the profit problem).
2. **The Crypto Arbitrage (Bloomberg):** "A trader has the ability to buy and sell Bitcoin within the same day across different time-stamped ticks. Given the limit of holding 1 BTC at a time, find the max profit."
3. **The Resource Rental (Meta):** "You are managing a rental fleet. A specific tool's value fluctuates daily. You can rent it out or keep it in the shop. Maximize the total value gained over a month."

309. Best Time to Buy and Sell Stock with Cooldown

A top-tier L5 or L6 engineer at Google doesn't just look for "the answer." They look for the **underlying state machine**. In a high-stakes interview, they would decompose this problem by identifying the constraints and the transitions between days.

1. Problem Explanation

You are given an array of stock prices. Each day, you can be in one of three states:

1. **Buying** a stock.
2. **Selling** a stock.
3. **Resting** (Cooldown).

The Catch: If you sell a stock on Monday, you **cannot** buy a stock on Tuesday. You must wait until Wednesday (at the earliest) to buy again. This "cooldown" is the non-trivial part that breaks the standard "Buy/Sell Stock" logic.

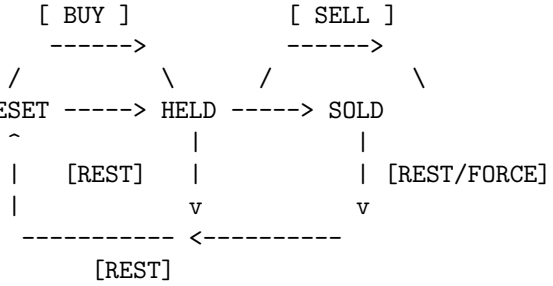
---

2. Solution Explanation

To an L6 engineer, this is a **Finite State Machine (FSM)** problem. We have three logical states we can be in at the end of any day:

- **State: HELD** (You own a stock. You are waiting to sell it).
- **State: SOLD** (You just sold a stock today. You are now forced into cooldown).
- **State: RESET** (You don't own a stock and you didn't just sell one. You are free to buy).

The State Transition Diagram (ASCII)



Transition Rules:

1. **From RESET:** You can stay in **RESET** (do nothing) or move to **HELD** (buy).
2. **From HELD:** You can stay in **HELD** (do nothing) or move to **SOLD** (sell).
3. **From SOLD:** You **must** move to **RESET** the next day (cooldown).

Step-by-Step Visualization (Prices: [1, 2, 3, 0, 2]) We track the max profit possible in each state at the end of the day. (Note: We start `held` at negative infinity because you can't have a stock without buying it).

Initial State: `reset = 0`, `held = -infinity`, `sold = -infinity`

Day 1 (Price: 1)

- `next_held`: max(held, reset - 1) = -1 <– (Bought stock at 1)
- `next_sold`: held + 1 = -inf <– (Can't sell yet)
- `next_reset`: max(reset, sold) = 0

```
Day 1: [R: 0, H: -1, S: -inf]
```

Day 2 (Price: 2)

- `next_held`: max(-1, 0 - 2) = -1 <– (Keep holding from Day 1)
- `next_sold`: -1 + 2 = 1 <– (Sold for profit of 1)
- `next_reset`: max(0, -inf) = 0

```
Day 2: [R: 0, H: -1, S: 1]
```

Day 3 (Price: 3) - COOLDOWN Triggered if we sold Day 2

- `next_held`: max(-1, 0 - 3) = -1
- `next_sold`: -1 + 3 = 2 <– (Sold for profit of 2)
- `next_reset`: max(0, 1) = 1 <– (The '1' from Day 2's SOLD moves here)

```
Day 3: [R: 1, H: -1, S: 2]
```

Day 4 (Price: 0)

- `next_held`: max(-1, 1 - 0) = 1 <– (Bought at price 0 using profit from Day 2)
- `next_sold`: -1 + 0 = -1
- `next_reset`: max(1, 2) = 2

```
Day 4: [R: 2, H: 1, S: -1]
```

Day 5 (Price: 2)

- `next_held`: max(1, 2 - 2) = 1
- `next_sold`: 1 + 2 = 3 <– (Final Profit!)
- `next_reset`: max(2, -1) = 2

```
Day 5: [R: 2, H: 1, S: 3] -> Max is 3.
```

---

3. Complexity Analysis

An L5/L6 engineer will point out that while Top-Down is easier to write, Bottom-Up with **Space Optimization** is the production-grade solution.

Time Complexity (TC) We iterate through the list of prices exactly once.

```
+-----------------------------------------------------+
| Operation             | Frequency | Cost per Op     |
```

```
+----------------------+-----------+-------------------+
| Iterate Prices Array | N times   | O(1) per state    |
+----------------------+-----------+-------------------+
| Total TC: O(N)                                       |
+------------------------------------------------------+
```

Space Complexity (SC) For the optimized Bottom-Up approach, we only store three variables (Reset, Held, Sold).

```
+------------------------------------------------------+
| Strategy              | Memory Usage                 |
+-----------------------+-----------+------------------+
| Top-Down (Memo)       | O(N) for recursion + map     |
| Bottom-Up (Full DP)   | O(N) for DP table            |
| Bottom-Up (Optimized) | O(1) for 3 state variables   |
+-----------------------+-----------+------------------+
| Total SC: O(1) (Optimized)                           |
+------------------------------------------------------+
```

---

4. Solution Code

Approach 1: Top-Down (Recursive with Memoization) This is the most intuitive "Brain to Code" path. We use a function `solve(index, can_buy, can_sell)`.

Python (Top-Down)

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        memo = {}

        # i: current day index
        # state: 0 = can buy, 1 = holding (can sell), 2 = cooldown
        def dp(i, state):
            if i == len(prices):
                return 0
            if (i, state) in memo:
                return memo[(i, state)]

            # Option 1: Do nothing (skip the day)
            res = dp(i + 1, state)

            if state == 0: # Can buy
                res = max(res, dp(i + 1, 1) - prices[i])
            elif state == 1: # Can sell
                # If we sell, the next state MUST be cooldown (state 2)
```

```
                res = max(res, dp(i + 1, 2) + prices[i])
            elif state == 2: # In cooldown
                # After cooldown day, we return to 'can buy' (state 0)
                res = max(res, dp(i + 1, 0))

            memo[(i, state)] = res
            return res

        return dp(0, 0)
```

Javascript (Top-Down)

```
var maxProfit = function(prices) {
    const memo = new Map();

    /**
     * @param {number} i - Current day index
     * @param {number} state - 0: Can Buy, 1: Holding, 2: Cooldown
     */
    function dp(i, state) {
        if (i >= prices.length) return 0;
        const key = `${i}-${state}`;
        if (memo.has(key)) return memo.get(key);

        // Always have the option to stay in the current logic but move to next day
        let res = dp(i + 1, state);

        if (state === 0) { // Buying
            res = Math.max(res, dp(i + 1, 1) - prices[i]);
        } else if (state === 1) { // Selling
            res = Math.max(res, dp(i + 1, 2) + prices[i]);
        } else if (state === 2) { // Just sold, must rest
            res = Math.max(res, dp(i + 1, 0));
        }

        memo.set(key, res);
        return res;
    }

    return dp(0, 0);
};
```

Approach 2: Bottom-Up (Iterative State Machine) This is the O(1) space version preferred in L6 interviews.

Python (Bottom-Up)

```python
def maxProfit(prices):
    if not prices: return 0

    # Initial states
    held = -float('inf') # Profit if we end the day holding a stock
    sold = -float('inf') # Profit if we end the day having just sold
    reset = 0            # Profit if we end the day in a neutral/cooldown-finished state

    for p in prices:
        prev_sold = sold
        # To reach SOLD state: we must have held a stock and sold it today
        sold = held + p
        # To reach HELD state: we either kept holding or bought from RESET
        held = max(held, reset - p)
        # To reach RESET state: we either stayed in reset or moved from a previous SOLD
        reset = max(reset, prev_sold)

    return max(sold, reset)
```

Javascript (Bottom-Up)

```javascript
var maxProfit = function(prices) {
    if (prices.length === 0) return 0;

    let held = -Infinity;
    let sold = -Infinity;
    let reset = 0;

    for (let p of prices) {
        let prevSold = sold;
        // Logic: You can only sell if you were holding
        sold = held + p;
        // Logic: You can only buy if you were in reset (not just sold)
        held = Math.max(held, reset - p);
        // Logic: Reset state is the max of previous reset or the cooldown after a sale
        reset = Math.max(reset, prevSold);
    }

    return Math.max(sold, reset);
};
```

Note 1: Terms & Techniques

- **Finite State Machine (FSM):** Instead of thinking about "what choice do I make," we think about "what state am I in." This helps simplify complex constraints like cooldowns.
- **State Compression:** By noticing we only need the values from `i-1` to calculate `i`, we reduced space from O(N) to O(1).

Note 2: Real World / Interview Variants

- **Google:** "Server Maintenance Windows" - You can run a heavy job, but after it finishes, the server needs a cool-down period before the next heavy job. Maximize total data processed.
- **Meta:** "Ad Campaign Pacing" - You can run an aggressive ad bid, but you must pause for a day to let the algorithm recalibrate before bidding again.
- **Bloomberg:** Often asked exactly as "Stock Cooldown" or "Stock with Transaction Fee," as it tests the ability to model financial constraints.

518. Coin Change II

This is a classic **Dynamic Programming (DP)** problem. At a top-tier level (L5/L6), the focus isn't just on getting the code to pass, but on understanding the **sub-problem structure** and optimizing the space complexity from a 2D grid down to a 1D array.

---

1. Problem Explanation

We are given an integer `amount` and an array of integers `coins`. We need to find the **number of combinations** that make up that amount.

The Key Distinction: This is about **combinations**, not **permutations**.

- If `amount = 3` and `coins = [1, 2]`:
- `1 + 2 = 3` is a valid combination.
- `2 + 1 = 3` is the **same** combination.
- We only count it once.

If we don't handle this carefully, we might accidentally count the same set of coins multiple times in different orders.

---

2. Solution Explanation

For a DP problem like this, the **Top-Down (Recursive + Memoization)** approach is usually the most intuitive because it mimics how a human thinks:

"To make 5, I can either use a 2 and then make 3, or I can skip the 2 entirely."

Step A: Top-Down Intuition

We define a function `change(index, remaining_amount)`. At each step, we have two choices to ensure we don't double-count:

1. **Use the current coin:** Subtract its value from the amount, but **stay at the same index** (since we can use coins infinitely).
2. **Skip the current coin:** Move to the next coin index and keep the amount the same.

Step B: The ASCII Visual Flow (Top-Down)

Let's use `amount = 5` and `coins = [1, 2, 5]`.

```
Decision Tree (Partial):
(Index, Amount)
      (0, 5)  <-- Start with Coin[0]=1
     /      \
  Use 1     Skip 1
  /           \
(0, 4)        (1, 5) <-- Move to Coin[1]=2
 /   \        /    \
...   ...   (1, 3) (2, 5)
```

Step C: Bottom-Up Intuition (The L5/L6 approach)

While recursion is intuitive, a senior engineer will typically implement the **Bottom-Up (Iterative)** approach for performance (avoiding stack overflow) and space optimization.

Imagine a table where rows are **coins** and columns are **amounts from 0 to 5**.

Base Case: There is exactly **1** way to make an amount of 0 (by choosing no coins).

```
Initial Table (DP[coin_index][amount]):
       Amt: 0  1  2  3  4  5
Coin 1:    1  0  0  0  0  0
Coin 2:    1  0  0  0  0  0
Coin 5:    1  0  0  0  0  0

Step 1: Process Coin 1
Ways to make amount 'i' = (Ways without 1) + (Ways to make 'i-1' using 1)
Amt: 0  1  2  3  4  5
     1  1  1  1  1  1  (Only one way for all: [1,1,1...])

Step 2: Process Coin 2 (Current + ways to make 'Amt - 2')
```

```
Amt: 0  1  2  3  4  5
     1  1  2  2  3  3
           ^  ^  ^  ^
           |  |  |  (DP[5] = DP[5] + DP[5-2]) -> 1 + 2 = 3 ways
           |  |  (DP[4] = DP[4] + DP[4-2]) -> 1 + 2 = 3 ways
           |  (DP[3] = DP[3] + DP[3-2]) -> 1 + 1 = 2 ways
           (DP[2] = DP[2] + DP[2-2]) -> 1 + 1 = 2 ways

Step 3: Process Coin 5
Amt: 0  1  2  3  4  5
     1  1  2  2  3  4  <-- Final Answer is 4
                     ^
                     (DP[5] = DP[5] + DP[5-5]) -> 3 + 1 = 4
```

---

3. Complexity Analysis

A Google-level interview requires deriving complexity by looking at the state space.

Time Complexity (TC)

We iterate through every coin, and for every coin, we iterate through every amount up to the target.

```
Coins (N) * Amount (A)
+--------------------------------------+
| For each coin in coins: (N iterations)|
|   For i from coin to amount: (A)     |
|      Update DP table                  |
+--------------------------------------+
Total TC: O(N * A)
```

Space Complexity (SC)

We can optimize this to use only a 1D array of size `amount + 1`.

```
+-----------------------+
| Array of size (A + 1) |
+-----------------------+
Total SC: O(A)
```

---

4. Solution Code

Python Snippets

```python
# TOP-DOWN APPROACH (Recursive with Memoization)
def change_top_down(amount, coins):
    memo = {}

    # This helper function explores two branches:
    # 1. Use the current coin (index)
    # 2. Skip to the next coin (index + 1)
    def dfs(index, current_amount):
        if current_amount == 0: return 1
        if current_amount < 0 or index == len(coins): return 0

        state = (index, current_amount)
        if state in memo: return memo[state]

        # Choice 1: Use the coin (can reuse it, so index stays same)
        # Choice 2: Skip the coin (move to index + 1)
        memo[state] = dfs(index, current_amount - coins[index]) + \
                      dfs(index + 1, current_amount)
        return memo[state]

    return dfs(0, amount)

# BOTTOM-UP APPROACH (Space Optimized 1D DP)
def change_bottom_up(amount, coins):
    # dp[i] will store the number of ways to make amount i
    dp = [0] * (amount + 1)
    dp[0] = 1 # Base case: 1 way to make amount 0

    # We iterate coin by coin to ensure we only count combinations,
    # not permutations (order doesn't matter).
    for coin in coins:
        for x in range(coin, amount + 1):
            # The number of ways to reach 'x' is its current ways
            # plus the ways we could reach 'x - coin'
            dp[x] += dp[x - coin]

    return dp[amount]
```

JavaScript Snippets

```javascript
/**
 * TOP-DOWN APPROACH
 * Uses a Map to store results of (index, amount) pairs to avoid redundant work.
```

```
 */
function changeTopDown(amount, coins) {
    const memo = new Map();

    function dfs(idx, rem) {
        if (rem === 0) return 1;
        if (rem < 0 || idx === coins.length) return 0;

        const key = `${idx}-${rem}`;
        if (memo.has(key)) return memo.get(key);

        // Result = ways including this coin + ways excluding this coin
        const result = dfs(idx, rem - coins[idx]) + dfs(idx + 1, rem);
        memo.set(key, result);
        return result;
    }

    return dfs(0, amount);
}

/**
 * BOTTOM-UP APPROACH
 * This is the standard "Knapsack-style" optimization.
 */
function changeBottomUp(amount, coins) {
    const dp = new Array(amount + 1).fill(0);
    dp[0] = 1;

    for (const coin of coins) {
        for (let i = coin; i <= amount; i++) {
            // Logic: New ways = Existing ways + ways to make (current - coin)
            dp[i] += dp[i - coin];
        }
    }

    return dp[amount];
}
```

Terminology & Real-World Context

Unbounded Knapsack: This problem is a variation of the "Knapsack Problem." It is "unbounded" because you have an infinite supply of each item (coin). DP helps here by breaking the large amount into smaller sub-amounts that are solved only once.

Why this helps: It prevents exponential time complexity. Without DP, the recursion tree would grow to a massive size. DP "remembers" that the ways to make amount 3 with a 1-cent coin is always the same, no matter how we got there.

Real-World Interview Variations

Google, Meta, and Bloomberg often disguise this problem in these ways:

1. **Step Climber (Meta):** "You can take steps of size 1, 2, or 5. How many ways to reach the top? (But order doesn't matter/Distinct sets only)."
2. **Resource Allocation (Google):** "You have a set of server types with different capacities. How many ways can you combine them to reach exactly X total capacity?"
3. **Currency Exchange (Bloomberg):** Often asked as part of a system design or coding round involving "making change" for a trading platform where only specific denominations are available.

416. Partition Equal Subset Sum

A Google L5 (Senior) or L6 (Staff) engineer doesn't just look for a "pass" on Leetcode; they look for **scalability, edge-case resilience, and memory optimization**. For "Partition Equal Subset Sum," they immediately recognize this as a variation of the **0/1 Knapsack Problem**.

---

1. Problem Explanation

The goal is to determine if an array of positive integers can be partitioned into two subsets such that the sum of elements in both subsets is equal.

The "Aha!" Moment: If the total sum of the array is S, and we want two equal subsets, each subset must sum to exactly **S / 2**.

- If **S** is odd, it is impossible to split it into two equal integers. We return `false` immediately.
- If **S** is even, the problem transforms: "Can we find a subset of numbers that adds up to exactly **Target (S / 2)**?"

---

2. Solution Explanation

We use **Dynamic Programming (DP)**. The most intuitive way to start is **Top-Down (Recursive with Memoization)** because it mimics how a human thinks: "Do I pick this number or skip it?"

Step A: The Decision Tree (Top-Down Intuition)

Imagine `nums = [1, 5, 11, 5]`. Total = 22. Target = 11. We start at index 0 and need to reach 11.

```
Level 0: Index 0 (val: 1), Target: 11
              /              \
        PICK 1               SKIP 1
       Target: 10           Target: 11
          /                    \
Level 1: Index 1 (val: 5)     Index 1 (val: 5)
      /         \              /         \
   PICK 5     SKIP 5       PICK 5     SKIP 5
  Target: 5  Target: 10   Target: 6  Target: 11
```

Step B: Bottom-Up DP (The Table Approach)

An L5 engineer prefers Bottom-Up for production because it avoids `StackOverflowError` on large inputs. We create a boolean table `dp[index][current_sum]`.

Logic: `dp[i][j]` is true if a sum `j` can be formed using a subset of the first `i` numbers.

ASCII Visualization: `nums = [1, 5, 11, 5]`, Target = 11 Initial State: (Row = numbers considered, Col = sum we want to reach) `T` = True, `.` = False

```
Sum ->  0  1  2  3  4  5  6  7  8  9  10 11
-----------------------------------------
Init    T  .  .  .  .  .  .  .  .  .  .  .  (Sum 0 is always possible)
```

After processing '1': (We can make sum 0 or sum 1)

```
Sum ->  0  1  2  3  4  5  6  7  8  9  10 11
-----------------------------------------
num 1:  T  T  .  .  .  .  .  .  .  .  .  .
```

After processing '5': (We take previous sums and add 5 to them)

```
Sum ->  0  1  2  3  4  5  6  7  8  9  10 11
-----------------------------------------
num 5:  T  T  .  .  .  T  T  .  .  .  .  .  (Sums possible: 0, 1, 0+5, 1+5)
```

After processing '11': (We take previous sums and add 11 to them)

```
Sum ->  0  1  2  3  4  5  6  7  8  9  10 11
-----------------------------------------
num 11: T  T  .  .  .  T  T  .  .  .  .  T  (11 is reached! 0+11 = 11)
```

Step C: Space Optimization (The L6 Move)

Notice that each row only depends on the row directly above it. We don't need a 2D grid. We can use a **1D array**. To avoid using the same number twice in one row, we iterate **backwards**.

```
Current DP State (Target = 11):
[T, F, F, F, F, F, F, F, F, F, F, F]

Processing num = 5:
Iterate j from 11 down to 5:
If dp[j - 5] is True, then dp[j] becomes True.
```

---

3. Complexity Analysis

An L5 would derive complexity by looking at the state space.

```
Time Complexity:
Number of states = (N * Target)
Work per state   = O(1)
Total TC         = O(N * S)
(where N is array length and S is the total sum)

Space Complexity:
Standard DP      = O(N * S) (2D Array)
Optimized DP     = O(S)     (1D Array)
```

---

4. Solution Code

Python

```python
# TOP-DOWN APPROACH (Recursive with Memoization)
def canPartition_TopDown(nums):
    total_sum = sum(nums)
    if total_sum % 2 != 0:
        return False

    target = total_sum // 2
    memo = {}

    # dfs(i, current_target) checks if we can reach current_target
    # using elements from index i onwards.
    def dfs(i, current_target):
        if current_target == 0: return True
        if i >= len(nums) or current_target < 0: return False
```

```
        state = (i, current_target)
        if state in memo: return memo[state]

        # Decision: Either include nums[i] or exclude it
        memo[state] = dfs(i + 1, current_target - nums[i]) or \
                      dfs(i + 1, current_target)
        return memo[state]

    return dfs(0, target)

# BOTTOM-UP APPROACH (Space Optimized 1D DP)
def canPartition_BottomUp(nums):
    total_sum = sum(nums)
    if total_sum % 2 != 0: return False

    target = total_sum // 2
    # dp[i] will be true if sum i is achievable
    dp = [False] * (target + 1)
    dp[0] = True

    for num in nums:
        # Iterate backwards to ensure we don't use the same
        # element multiple times for the same target sum.
        for j in range(target, num - 1, -1):
            if dp[j - num]:
                dp[j] = True

    return dp[target]
```

JavaScript

```
/**
 * TOP-DOWN APPROACH
 * Uses a Map for memoization to store (index, remainingTarget)
 */
function canPartitionTopDown(nums) {
    const totalSum = nums.reduce((a, b) => a + b, 0);
    if (totalSum % 2 !== 0) return false;

    const target = totalSum / 2;
    const memo = new Map();

    function dfs(i, currentTarget) {
        if (currentTarget === 0) return true;
        if (i >= nums.length || currentTarget < 0) return false;
```

```
        const key = `${i}-${currentTarget}`;
        if (memo.has(key)) return memo.get(key);

        const result = dfs(i + 1, currentTarget - nums[i]) ||
                       dfs(i + 1, currentTarget);

        memo.set(key, result);
        return result;
    }

    return dfs(0, target);
}

/**
 * BOTTOM-UP APPROACH (Highly Optimized)
 * This is what a Google interviewer looks for in the final 10 mins.
 */
function canPartitionBottomUp(nums) {
    const totalSum = nums.reduce((a, b) => a + b, 0);
    if (totalSum % 2 !== 0) return false;

    const target = totalSum / 2;
    const dp = new Array(target + 1).fill(false);
    dp[0] = true;

    for (const num of nums) {
        for (let j = target; j >= num; j--) {
            // If the sum (j - num) was possible,
            // then sum j is now possible by adding the current num.
            if (dp[j - num]) {
                dp[j] = true;
            }
        }
    }

    return dp[target];
}
```

Terminology

- **0/1 Knapsack:** A category of problems where you have items with certain weights/values and you must decide whether to "take it or leave it" (0 or 1) to reach a target.

- **Memoization:** An optimization technique where you store the results of expensive function calls (top-down) to avoid redundant calculations.
- **Space Optimization:** Reducing a 2D DP table to 1D when only the previous state is required.

Real-World / Interview Variations

1. **Google (Load Balancing):** "You have N tasks with specific execution times. Can you split them between two identical servers so both finish at the exact same time?"
2. **Meta (Ad Budgeting):** "Given a set of ad campaign costs, can we select a group of campaigns that exactly consume half of the total marketing budget?"
3. **Bloomberg (Portfolio Hedging):** "Can a set of financial assets be divided into two portfolios of equal total value to minimize risk?"

494. Target Sum

An L5 or L6 engineer at Google doesn't just "solve" a problem; they look for the underlying pattern, optimize for edge cases, and ensure the logic is modular. For **Target Sum**, the key is recognizing that this isn't just a "try everything" problem—it's a variation of the classic **0/1 Knapsack Problem**.

---

1. Problem Explanation

We are given an array of integers `nums` and a `target`. We must use **every** number in the array exactly once, assigning either a `+` or a `-` sign to each. The goal is to find the total number of ways to assign these signs so that the sum equals the `target`.

The Non-Trivial Insight: This looks like a branching tree where every number has two choices. However, we can transform this into a simpler math problem. Suppose we split our numbers into two sets:

- **P**: Numbers assigned a `+` sign.
- **N**: Numbers assigned a `-` sign.

We know:

1. Sum(P) - Sum(N) = target
2. Sum(P) + Sum(N) = total_sum (of all numbers)

By adding these two equations: `2 * Sum(P) = target + total_sum Sum(P) = (target + total_sum) / 2`

Conclusion: The problem is actually: "How many ways can we pick a subset of numbers that sum up to a specific value `S`?" where `S = (target + total_sum)`

/ 2.

---

2. Solution Explanation

Phase 1: Top-Down (Recursive with Memoization)

The most intuitive way to start is by thinking about the choices at each index.

The Decision Tree Logic: At each index `i`, we either add `nums[i]` to our current running total or subtract it.

```
Decision Tree Visualization (nums = [1, 1], target = 0)

          Start (Index 0, Current Sum 0)
                 /              \
         Add +1 /                \ Sub -1
               /                  \
        (Idx 1, Sum 1)          (Idx 1, Sum -1)
          /        \              /        \
     Add+1/    Sub-1\        Add+1/    Sub-1\
        /          \            /          \
  (Idx 2, Sum 2) (Idx 2, Sum 0) (Idx 2, Sum 0) (Idx 2, Sum -2)
      INVALID       SUCCESS        SUCCESS        INVALID
```

Memoization: We store `(index, current_sum)` in a map. If we ever reach the same index with the same sum again, we return the cached result.

Phase 2: Bottom-Up (Dynamic Programming)

To convert this to a 2D table, we use the subset sum logic derived in the problem explanation. We want to find how many ways to make the sum `S`.

DP Table Logic (nums = [1, 1, 2], S = 2):

- Rows: Numbers considered (0 to n)
- Cols: Target sums (0 to S)

Step-by-Step Table Fill:

```
Initial State (Ways to make sum 0 with 0 items is 1):
      Sum ->  0  1  2
Idx 0 (None)  1  0  0

Using num = 1:
Idx 1 (1)     1  1  0  (1 way to make 0, 1 way to make 1)

Using next num = 1:
Idx 2 (1,1)   1  2  1  (Ways to make 2: use the new 1 + old ways to make 1)
```

```
Using next num = 2:
Idx 3 (1,1,2) 1  2  2  (Ways to make 2: old ways to make 2 + ways to make 0)
```

Space Optimization: Notice that to calculate a row, we only need the row directly above it. We can collapse this into a 1D array, iterating **backwards** to avoid using the same number twice for the same target.

---

3. Complexity Analysis

Time Complexity (TC)

```
TC Calculation:
Number of States = (Number of elements) * (Possible Sum Range)
Work per state   = O(1) constant time addition
-----------------------------------------------------------
Total TC         = O(n * S)
(where n is array length and S is the target subset sum)
```

Space Complexity (SC)

```
SC Calculation:
Top-Down Approach:
  - Recursion Stack: O(n)
  - Memoization Map: O(n * S)
Bottom-Up Approach:
  - 2D Table: O(n * S)
  - Optimized 1D: O(S)
-----------------------------------------------------------
Optimized SC     = O(S)
```

---

4. Solution Code

Python

```python
def findTargetSumWays_TopDown(nums, target):
    total_sum = sum(nums)
    # Edge cases for math transformation
    if abs(target) > total_sum or (target + total_sum) % 2 != 0:
        return 0

    memo = {}

    # recursive_solve explores all sign combinations using memoization
    def recursive_solve(index, current_sum):
```

```
        state = (index, current_sum)
        if state in memo:
            return memo[state]

        if index == len(nums):
            return 1 if current_sum == target else 0

        # Choice 1: Add the number
        add = recursive_solve(index + 1, current_sum + nums[index])
        # Choice 2: Subtract the number
        subtract = recursive_solve(index + 1, current_sum - nums[index])

        memo[state] = add + subtract
        return memo[state]

    return recursive_solve(0, 0)

def findTargetSumWays_BottomUp(nums, target):
    total_sum = sum(nums)
    if abs(target) > total_sum or (target + total_sum) % 2 != 0:
        return 0

    subset_target = (target + total_sum) // 2

    # dp[i] represents the number of ways to reach sum i
    dp = [0] * (subset_target + 1)
    dp[0] = 1 # Base case: 1 way to make sum 0 (empty set)

    for num in nums:
        # We go backwards to ensure we don't use the current 'num'
        # more than once for the same subset
        for s in range(subset_target, num - 1, -1):
            dp[s] += dp[s - num]

    return dp[subset_target]
```

Javascript

```
/**
 * Top-Down Approach (Memoization)
 */
function findTargetSumWaysTopDown(nums, target) {
    const totalSum = nums.reduce((a, b) => a + b, 0);
    if (Math.abs(target) > totalSum || (target + totalSum) % 2 !== 0) return 0;

    const memo = new Map();
```

```
    function solve(index, currentSum) {
        const key = `${index},${currentSum}`;
        if (memo.has(key)) return memo.get(key);

        if (index === nums.length) {
            return currentSum === target ? 1 : 0;
        }

        const add = solve(index + 1, currentSum + nums[index]);
        const subtract = solve(index + 1, currentSum - nums[index]);

        memo.set(key, add + subtract);
        return add + subtract;
    }

    return solve(0, 0);
}

/**
 * Bottom-Up Approach (Iterative DP with Space Optimization)
 */
function findTargetSumWaysBottomUp(nums, target) {
    const totalSum = nums.reduce((a, b) => a + b, 0);
    if (Math.abs(target) > totalSum || (target + totalSum) % 2 !== 0) return 0;

    const subsetTarget = (target + totalSum) / 2;
    const dp = new Array(subsetTarget + 1).fill(0);
    dp[0] = 1;

    // Outer loop iterates through each available number
    for (const num of nums) {
        // Inner loop updates possible sums using the current number
        for (let s = subsetTarget; s >= num; s--) {
            dp[s] += dp[s - num];
        }
    }

    return dp[subsetTarget];
}
```

Terminology & Techniques

- **Subset Sum Transformation:** Converting a problem with `+/-` choices into a problem of picking a subset. This is helpful because it limits the search space to positive integers and fits the Knapsack pattern.
- **Memoization:** Storing the results of expensive function calls to avoid redundant "re-calculating" of the same sub-problems.
- **0/1 Knapsack Pattern:** A foundational DP pattern where you decide whether to "include" or "exclude" an item to reach a target.

Real-World / Interview Variations

Companies like Google and Meta rarely ask this exact Leetcode version now; they use variations:

1. **The "Load Balancer" Problem:** You have tasks with weights (CPU usage) and two servers. Assign signs (or assign to Server A/B) such that the difference in load is exactly `X`.
2. **Expression Generator:** Given a string of digits, how many ways can you insert `+` and `-` to reach a target? (Meta variant).
3. **Bloomberg "Profit/Loss" scenario:** Given a list of daily trades, how many combinations of "Long" or "Short" positions result in a net profit of `K`?

39. Combination Sum

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just look for a way to pass the test cases; they look for the **optimal strategy**, the **edge cases**, and the **scalability** of the logic.

For "Combination Sum," the core challenge is navigating a search space where you can reuse elements. A high-level engineer recognizes this immediately as a **Backtracking** problem that can be modeled as a **Decision Tree**.

---

1. Problem Explanation

Given an array of **distinct** integers `candidates` and a `target` integer, return a list of all **unique combinations** where the chosen numbers sum to the target.

The Twist: You may use the same number from `candidates` an unlimited number of times.

Key Constraints:

- All numbers (including target) are positive.
- The solution set must not contain duplicate combinations (e.g., `[2, 2, 3]` and `[3, 2, 2]` are the same).

2. Solution Explanation

To solve this, we use **Backtracking**. Think of it as an explorer in a maze. At every step, the explorer has two choices:

1. **Take the current number:** Add it to the current path and stay at the same index (since we can reuse it).
2. **Skip the current number:** Move to the next index and never look back at the current number for this specific path.

The Decision Tree Visualization

Let `candidates = [2, 3]` and `target = 5`.

```
Root: Sum=0, Index=0 (Val=2)
          /                  \
     ADD 2 (Index 0)        SKIP 2 (Move to Index 1, Val=3)
    Sum=2, Path=[2]         Sum=0, Path=[]
      /        \              /        \
   ADD 2      SKIP 2      ADD 3      SKIP 3
  Sum=4       Sum=2       Sum=3       Sum=0
  Path=[2,2]  Path=[2]    Path=[3]    Path=[]
   /    \       /   \       /   \         \
ADD 2  SKIP 2 ADD 3 SKIP 3 ADD 3 SKIP 3   (End)
Sum=6  Sum=4  Sum=5  Sum=2 Sum=6 Sum=3
(Over) (End)  (WIN!) (End) (Over) (End)
```

Detailed Step-by-Step Walkthrough

We will use a "Remaining Target" approach. Every time we pick a number, we subtract it from the target. If the target reaches 0, we found a match.

Example: `candidates = [2, 3, 6, 7]`, `target = 7`

```
STEP 1: Start at index 0 (val 2), Target 7.
        Path: [2]
        Remaining: 7 - 2 = 5

STEP 2: We can reuse 2. Stay at index 0.
        Path: [2, 2]
        Remaining: 5 - 2 = 3

STEP 3: Reuse 2 again. Stay at index 0.
        Path: [2, 2, 2]
        Remaining: 3 - 2 = 1
```

```
STEP 4: Reuse 2 again?
        Remaining: 1 - 2 = -1 (STOP! Too far. Backtrack to STEP 3)

STEP 5: From [2, 2, 2], skip 2 and move to index 1 (val 3).
        Path: [2, 2, 2, 3] -> (Over target, Backtrack to STEP 2)

STEP 6: From [2, 2], skip 2 and move to index 1 (val 3).
        Path: [2, 2, 3]
        Remaining: 3 - 3 = 0 (SUCCESS! Add [2, 2, 3] to results)

STEP 7: Backtrack all the way to the start and try starting with index 3 (val 7).
        Path: [7]
        Remaining: 7 - 7 = 0 (SUCCESS! Add [7] to results)
```

---

3. Complexity Analysis

An L6 engineer would derive the complexity by looking at the depth of the recursion tree.

Time Complexity (TC)

```
The height of the tree (H) is (Target / Minimum Candidate Value).
At each node, we have 2 choices (Include or Skip).

TC Diagram:
Total Nodes approx = 2 raised to the power of (Target / Min_Val)

In our case:
TC = O(2^T) where T = target/min_candidate
Note: We also multiply by 'k' (average length of combination)
to account for copying the path into the result list.
```

Space Complexity (SC)

```
The space is determined by the recursion stack and the current path storage.

SC Diagram:
Recursion Stack Depth = Target / Min_Value
Current Path Storage  = Target / Min_Value

SC = O(Target / Min_Value)
```

---

4. Solution Code

Top-Down (Recursive Backtracking)

This is the most intuitive approach for this problem because it mimics the decision tree.

Python (Top-Down)

```python
def combinationSum(candidates, target):
    results = []

    # backtrack function explores all valid paths
    # i: current index in candidates
    # current_path: list of numbers picked so far
    # total: current sum of numbers in current_path
    def backtrack(i, current_path, total):
        # Base Case: Found a valid combination
        if total == target:
            results.append(list(current_path))
            return

        # Base Case: Exceeded target or exhausted candidates
        if i >= len(candidates) or total > target:
            return

        # Choice 1: Include candidates[i]
        # We don't increment 'i' because we can reuse the same element
        current_path.append(candidates[i])
        backtrack(i, current_path, total + candidates[i])

        # Choice 2: Skip candidates[i] (Backtrack)
        # We must pop the element we just added to clean up the state
        current_path.pop()
        backtrack(i + 1, current_path, total)

    backtrack(0, [], 0)
    return results
```

JavaScript (Top-Down)

```javascript
var combinationSum = function(candidates, target) {
    const results = [];

    /**
     * Helper to explore combinations
     * @param {number} i - Current candidate index
```

```
     * @param {number[]} path - Current subset
     * @param {number} remaining - What's left to reach target
     */
    const backtrack = (i, path, remaining) => {
        if (remaining === 0) {
            results.push([...path]);
            return;
        }
        if (i === candidates.length || remaining < 0) {
            return;
        }

        // Include the current candidate
        path.push(candidates[i]);
        backtrack(i, path, remaining - candidates[i]);

        // Backtrack: remove the last element and try the next candidate
        path.pop();
        backtrack(i + 1, path, remaining);
    };

    backtrack(0, [], target);
    return results;
};
```

Bottom-Up (Dynamic Programming)

While backtracking is more common for "return all paths," DP is great for "how many paths" or if you want to build solutions iteratively. Here, `dp[i]` stores all combinations that sum to `i`.

Python (Bottom-Up)

```
def combinationSum(candidates, target):
    # dp[i] will store a list of all combinations that sum to i
    dp = [[] for _ in range(target + 1)]
    dp[0] = [[]] # Base case: one way to make 0 is an empty list

    for c in candidates:
        # For every possible sub-target from the candidate value to target
        for i in range(c, target + 1):
            # For every combination already found for sum (i - c)
            # Add the current candidate 'c' to it to form sum 'i'
            for combination in dp[i - c]:
                dp[i].append(combination + [c])
```

```
    return dp[target]
```

JavaScript (Bottom-Up)

```javascript
var combinationSum = function(candidates, target) {
    // Initialize DP array where dp[i] is a list of combinations summing to i
    const dp = Array.from({ length: target + 1 }, () => []);
    dp[0] = [[]];

    for (const cand of candidates) {
        // Iterate through all possible sums from the candidate value up to target
        for (let i = cand; i <= target; i++) {
            // Take existing combinations for (i - cand) and append current candidate
            for (const prevCombo of dp[i - cand]) {
                dp[i].push([...prevCombo, cand]);
            }
        }
    }

    return dp[target];
};
```

Terminology & Real-World Context

New Terms

- **Backtracking:** An algorithmic technique that builds a solution incrementally and abandons a path ("backtracks") as soon as it determines that the path cannot lead to a valid solution. It saves massive amounts of time compared to simple brute force.
- **Decision Tree:** A visual tool used to represent all possible choices and outcomes in a process.

Real-World Interview Variations

Google, Meta, and Bloomberg rarely ask "Combination Sum" directly anymore. Instead, they frame it as:

1. **Vending Machine Logic:** "Given coin denominations, return all ways to make change for X dollars."
2. **Resource Allocation:** "Given server capacities, find all combinations of servers that can handle exactly X load."
3. **Budgeting/Shopping:** "A user has a gift card of value X. Given a list of prices, what are all the ways they can spend exactly that amount?" (Common at Amazon/Bloomberg).

4. **Packet Switching:** "Find all combinations of packet sizes that can fill a specific buffer size."

40. Combination Sum II

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just look for *a* solution; they look for the most efficient way to handle **constraints** and **edge cases** while maintaining clean, readable code.

For "Combination Sum II," the "senior" insight is realizing this isn't just about recursion—it's about **pruning the decision tree** to handle duplicates without using a heavy `Set` object.

---

1. Problem Explanation

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target`.

The Constraints that make this tricky:

- Each number in `candidates` may only be used **once** in the combination.
- The solution set must **not contain duplicate combinations**.
- The input array can contain duplicate numbers (e.g., `[1, 1, 2, 5]`).

---

2. Solution Explanation

To solve this, we use **Backtracking with Sorting**.

The Strategy: "Don't Repeat Yourself"

If we have `[1, 1, 7]` and the target is `8`, we could pick the first `1` and the `7`, or the second `1` and the `7`. Both result in `[1, 7]`. To avoid this:

1. **Sort** the array first.
2. If the current element is the same as the previous element **at the same recursion level**, skip it.

Step-by-Step ASCII Visualization

Let's use `candidates = [1, 2, 2]` and `target = 3`.

Step 0: Sort the array `Sorted candidates: [1, 2, 2]`

Step 1: The Decision Tree Each level of the tree represents choosing one index from the remaining candidates.

```
Level 0: Start (Target: 3, Path: [])
|
|-- Choose index 0 (Value: 1) -> New Target: 2, Path: [1]
|   |
|   |-- Choose index 1 (Value: 2) -> New Target: 0, Path: [1, 2] SUCCESS!
|   |   (Backtrack: Remove 2)
|   |
|   |-- Choose index 2 (Value: 2) -> DUPLICATE SKIP!
|       (Index 2 has value '2', same as Index 1. We already explored '2' at this level.)
|
|-- Choose index 1 (Value: 2) -> New Target: 1, Path: [2]
|   |
|   |-- Choose index 2 (Value: 2) -> New Target: -1, Path: [2, 2] TOO BIG (Prune)
|
|-- Choose index 2 (Value: 2) -> DUPLICATE SKIP!
    (Index 2 is '2', same as Index 1 at this level. Skip.)
```

Visualizing the "Sibling Skip" Logic

This is the most non-trivial part. We skip duplicates among **siblings**, not among **ancestors**.

```
[ 1,  2,  2,  5 ]
  ^   ^   ^
  i  i+1 i+2

If we are at a loop for a specific level:
For i = 1: We pick '2'.
For i = 2: We see candidates[2] == candidates[1].
           Since i > start_index, we SKIP this '2'.
           This prevents two separate branches starting with '2'.
```

---

3. Complexity Analysis

An L6 engineer would derive complexity by looking at the state-space tree.

Time Complexity (TC)

```
TC = O( 2^N * N )

Explanation:
1. In the worst case (e.g., [1, 1, 1, 1], target 10), every element
   could either be included or excluded. This is 2^N combinations.
2. For each valid combination, we spend O(N) time to copy the
```

```
   current path into our result list.
3. Sorting takes O(N * log N), which is overshadowed by 2^N.
```

Space Complexity (SC)

```
SC = O( N )

Explanation:
1. The recursion stack goes at most N levels deep (the length of the array).
2. The 'path' array stores at most N elements.
3. Note: We do not count the output list in space complexity
   for algorithmic analysis, but if we did, it could be O(2^N * N).
```

---

4. Solution Code

Approach: Backtracking (Top-Down)

This is the most intuitive way to solve "Combination" problems because it mimics a decision tree.

Python (Top-Down)

```python
def combinationSum2(candidates, target):
    results = []
    candidates.sort() # Critical for duplicate handling

    def backtrack(start_index, current_target, path):
        # Base Case: We hit the target
        if current_target == 0:
            results.append(list(path))
            return

        # Base Case: Exceeded target
        if current_target < 0:
            return

        for i in range(start_index, len(candidates)):
            # NON-TRIVIAL: Skip duplicates
            # If the current element is same as previous AND it's not the
            # first element in this loop, it's a duplicate branch.
            if i > start_index and candidates[i] == candidates[i-1]:
                continue

            path.append(candidates[i])
            # Move to i + 1 because we can't reuse the same element
```

```
            backtrack(i + 1, current_target - candidates[i], path)
            path.pop() # Remove last element to try next candidate

    backtrack(0, target, [])
    return results
```

JavaScript (Top-Down)

```
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum2 = function(candidates, target) {
    const results = [];
    candidates.sort((a, b) => a - b);

    /**
     * Recursive function to explore combinations
     * @param {number} start - Current index in candidates
     * @param {number} remainder - Remaining sum needed
     * @param {number[]} path - Current numbers selected
     */
    function backtrack(start, remainder, path) {
        if (remainder === 0) {
            results.push([...path]);
            return;
        }

        for (let i = start; i < candidates.length; i++) {
            // Optimization: If the smallest available number is
            // larger than remainder, no point in continuing the loop
            if (candidates[i] > remainder) break;

            // Duplicate pruning
            if (i > start && candidates[i] === candidates[i - 1]) continue;

            path.push(candidates[i]);
            backtrack(i + 1, remainder - candidates[i], path);
            path.pop();
        }
    }

    backtrack(0, target, []);
    return results;
};
```

Approach: Dynamic Programming (Bottom-Up)

While backtracking is preferred for "return all paths" problems, you can solve this using a DP table where `dp[t]` stores all unique combinations that sum to `t`.

Python (Bottom-Up DP)

```python
def combinationSum2_dp(candidates, target):
    candidates.sort()
    # dp[i] will store a set of tuples (to handle uniqueness)
    dp = [set() for _ in range(target + 1)]
    dp[0].add(()) # Base case: sum 0 is an empty tuple

    for num in candidates:
        # Iterate backwards through DP table to ensure each
        # candidate is used only once for the current target
        for t in range(target, num - 1, -1):
            for prev_comb in dp[t - num]:
                dp[t].add(prev_comb + (num,))

    return [list(x) for x in dp[target]]
```

JavaScript (Bottom-Up DP)

```javascript
var combinationSum2_dp = function(candidates, target) {
    candidates.sort((a, b) => a - b);
    // Use an array of Sets of strings to manage unique combinations easily
    let dp = Array.from({ length: target + 1 }, () => new Set());
    dp[0].add("");

    for (let num of candidates) {
        for (let t = target; t >= num; t--) {
            for (let prev of dp[t - num]) {
                // Store as comma-separated string to keep Set uniqueness
                let newComb = prev === "" ? `${num}` : `${prev},${num}`;
                dp[t].add(newComb);
            }
        }
    }

    return Array.from(dp[target]).map(s => s.split(',').map(Number));
};
```

Note 1: Terms & Techniques

- **Backtracking:** A refined brute-force technique. Instead of generating all permutations and checking them, we build the solution incrementally and "abandon ship" (backtrack) as soon as we realize the current path cannot lead to a valid solution.
- **Pruning:** The act of cutting off branches of the recursion tree. In this problem, `i > start && candidates[i] == candidates[i-1]` is a pruning condition that saves us from exponential redundant work.
- **Top-Down vs Bottom-Up:** Top-down (Recursion) is usually more intuitive for "search" problems. Bottom-up (Iterative DP) is often more efficient for "counting" or "optimization" problems but can be memory-intensive when storing actual paths.

Note 2: Real-World Interview Variants

In Google, Meta, or Bloomberg interviews, they rarely ask the Leetcode version directly. They use these "hidden" variants:

1. **Google (Resource Allocation):** "Given a set of server types with different CPU capacities, find all unique ways to provision servers to hit exactly X total CPU, but you can only use each physical server from the inventory once."
2. **Meta (Ad Budgeting):** "A marketer has a list of specific ad campaign costs. Find all unique combinations of campaigns that exactly exhaust a budget of Y, ensuring no duplicate sets of campaigns are suggested."
3. **Bloomberg (Portfolio Construction):** "Given a list of bond denominations (some duplicates), find all ways to create a sub-portfolio that totals a specific par value."

64. Minimum Path Sum

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just solve the problem; they focus on **edge cases, memory efficiency, and clear communication.** For "Minimum Path Sum," they would immediately recognize this as a classic Grid Dynamic Programming (DP) problem and discuss the trade-offs between recursion and iteration.

---

1. Problem Explanation

You are given an `m x n` grid filled with non-negative numbers. You need to find a path from the **top-left** (0, 0) to the **bottom-right** (m-1, n-1) which **minimizes the sum** of all numbers along its path.

The Catch: You can only move either **down** or **right** at any point in time.

Visualizing the Constraints

Imagine a 3x3 grid:

```
[ 1, 3, 1 ]
[ 1, 5, 1 ]
[ 4, 2, 1 ]
```

- You start at `1` (top-left).
- You must end at `1` (bottom-right).
- Valid move: `(0,0) -> (0,1)` (Right) or `(0,0) -> (1,0)` (Down).
- Invalid move: `(1,1) -> (0,1)` (Up) or `(1,1) -> (1,0)` (Left) or Diagonals.

---

2. Solution Explanation

An L5+ engineer would start with the **Recursive (Top-Down)** intuition because it mirrors how humans think, then optimize it into **Iterative (Bottom-Up)** for production-grade performance.

Step A: The Top-Down Intuition (Recursive)

To find the min path to the end, you ask: "What was the best way to get to the cell just before the end?" Since we can only move Down or Right, the only way to reach the bottom-right `(2,2)` is from either the Top `(1,2)` or the Left `(2,1)`.

The Recurrence Relation: `Cost(r, c) = Grid(r, c) + min(Cost(r-1, c), Cost(r, c-1))`

Step B: The Bottom-Up Visualization (Tabulation)

We create a 2D array (DP table) of the same size. Each cell `dp[r][c]` will store the minimum cost to reach that specific cell.

Initial State (The Grid)

```
Grid:
1  3  1
1  5  1
4  2  1
```

Step 1: Initialize the starting point `dp[0][0]` is just the value of `grid[0][0]`.

```
DP Table:
1  .  .
```

```
.  .  .
.  .  .
```

Step 2: Fill the first row and first column (Base Cases) To reach any cell in the first row, you *must* come from the left. To reach any cell in the first column, you *must* come from above.

```
DP Table (Row 0): 1 -> (1+3) -> (4+1) = 5
DP Table (Col 0): 1 -> (1+1) -> (2+4) = 6


1  4  5
2  .  .
6  .  .
```

Step 3: Fill the rest of the grid For `dp[1][1]`, we look at its Top (4) and its Left (2). The minimum is 2. `dp[1][1] = grid[1][1] + min(2, 4) = 5 + 2 = 7`.

Filling (1,1) and (1,2):

```
1  4  5
2  7  (1 + min(7, 5)) = 6
6  .  .
```

Filling the final row (2,1) and (2,2):

```
1  4  5
2  7  6
6  (2 + min(6, 7)) = 8 | (1 + min(8, 6)) = 7
```

Final DP Table:

```
[ 1, 4, 5 ]
[ 2, 7, 6 ]
[ 6, 8, 7 ]  <-- The answer is 7.
```

---

3. Time and Space Complexity Analysis

Time Complexity (TC)

We visit every cell in the `m x n` grid exactly once. Inside each cell, we perform a simple addition and a `min()` comparison (constant time).

```
Complexity Derivation:
Rows (m) * Columns (n) * Constant Work (1)
Result: O(m * n)
```

Space Complexity (SC)

- **Standard DP:** We use a 2D array of size `m x n`.
- **Space-Optimized:** Notice we only ever need the **previous row** and the **current row**. We can reduce this to a 1D array of size `n`.

```
Standard SC: O(m * n)  (2D DP Table)
Optimized SC: O(n)     (1D DP Row)
```

---

4. Solution Code

Python Snippets

```python
# Top-Down Approach (Memoization)
def minPathSum_TopDown(grid):
    m, n = len(grid), len(grid[0])
    memo = {}

    # Recursive function with memoization to avoid redundant calculations
    def calculate(r, c):
        if r == 0 and c == 0: return grid[0][0]
        if r < 0 or c < 0: return float('inf')
        if (r, c) in memo: return memo[(r, c)]

        # Result = current cell value + min of path from above or left
        res = grid[r][c] + min(calculate(r - 1, c), calculate(r, c - 1))
        memo[(r, c)] = res
        return res

    return calculate(m - 1, n - 1)

# Bottom-Up Approach (Tabulation - Space Optimized)
def minPathSum_BottomUp(grid):
    m, n = len(grid), len(grid[0])
    # dp[j] stores the min cost to reach grid[i][j]
    dp = [float('inf')] * n
    dp[0] = 0

    for i in range(m):
        for j in range(n):
            if j == 0:
                # For the first column, we can only come from above
                dp[j] += grid[i][j]
            else:
                # For other cells, min(from above [dp[j]], from left [dp[j-1]])
                dp[j] = grid[i][j] + min(dp[j], dp[j - 1])
```

```
    return dp[-1]
```

JavaScript Snippets

```javascript
// Top-Down Approach (Memoization)
function minPathSumTopDown(grid) {
    const m = grid.length, n = grid[0].length;
    const memo = Array.from({ length: m }, () => Array(n).fill(-1));

    // Helper to find min cost to reach cell (r, c)
    function solve(r, c) {
        if (r === 0 && c === 0) return grid[0][0];
        if (r < 0 || c < 0) return Infinity;
        if (memo[r][c] !== -1) return memo[r][c];

        memo[r][c] = grid[r][c] + Math.min(solve(r - 1, c), solve(r, c - 1));
        return memo[r][c];
    }

    return solve(m - 1, n - 1);
}

// Bottom-Up Approach (Tabulation - Space Optimized)
function minPathSumBottomUp(grid) {
    const m = grid.length, n = grid[0].length;
    // We only need a 1D array to track the "previous row" results
    let dp = new Array(n).fill(Infinity);
    dp[0] = 0;

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (j === 0) {
                // First column: can only add the value from the cell above
                dp[j] += grid[i][j];
            } else {
                // Pick the minimum path between coming from top or left
                dp[j] = grid[i][j] + Math.min(dp[j], dp[j - 1]);
            }
        }
    }

    return dp[n - 1];
}
```

Terminology & Algorithm Deep Dive

- **Dynamic Programming (DP):** A method for solving complex problems by breaking them down into simpler subproblems. It "remembers" (memoizes) the results of subproblems to avoid re-calculating them.
- **Overlapping Subproblems:** In this grid, many paths lead to the same cell. DP ensures we only calculate the "best" way to reach that cell once.
- **Space Optimization:** Since `dp[i][j]` only depends on `dp[i-1][j]` (the cell directly above in the previous row) and `dp[i][j-1]` (the cell to the left in the current row), we don't need a full 2D grid. We can overwrite the 1D array as we go.

Real World / Interview Variations

1. **Google (Map Routing):** "Given a grid of traffic latency, find the fastest route from point A to B." This is essentially Minimum Path Sum with weights.
2. **Meta (Energy Efficiency):** "Find a path through a data center grid that passes through the least amount of heat-intensive zones."
3. **Bloomberg (Stock Arbitrage/Path):** "Given a matrix of price slippage, find a sequence of trades (moving only forward in time/index) that minimizes total slippage."
4. **The "Max" Variation:** Instead of Minimum Path Sum, find the **Maximum Path Sum** (e.g., collecting the most gold coins in a grid). The logic is identical, just swap `min()` for `max()`.

22. Generate Parentheses

An L5 or L6 engineer at Google doesn't just look for a "working" solution; they look for the most efficient **state-space exploration** and clear **invariant-based reasoning**. For "Generate Parentheses," they immediately recognize this as a combinatorial search problem that can be pruned using simple rules.

---

1. Problem Explanation

Given `n` pairs of parentheses, you need to generate all combinations of well-formed parentheses.

What is "Well-formed"?

1. **Balance:** You can't have more closing brackets `)` than opening brackets `(` at any point.
2. **Completeness:** The total number of `(` must equal the total number of `)` (both equal to `n`).

Example: n = 2

- Possible: `(())`, `()()`
- Impossible: `())(` (Closing bracket appears before a second opening bracket is available).

---

2. Solution Explanation

The most intuitive way to solve this is **Top-Down Recursion with Backtracking**. We treat the process as building a string character by character.

The Invariants (Rules)

- **Rule A:** We can add `(` if the `open_count` is less than `n`.
- **Rule B:** We can add `)` if the `close_count` is less than the `open_count`.

Step-by-Step ASCII Visualization (n = 2)

We start with an empty string `""` and counts `(0, 0)`.

```
Level 0:                [ "" (0,0) ]
                          /
Level 1:           [ "(" (1,0) ]  <-- Only '(' is possible (Rule B prevents ')')
                     /        \
Level 2:      ["( (" (2,0)]  ["() " (1,1)]
                \              /
Level 3:      ["(( )" (2,1)] ["()( " (2,1)]
                  \              \
Level 4:      ["(())" (2,2)] ["()()" (2,2)]
                  |              |
                VALID          VALID
```

Full Decision Tree Trace (n = 3)

Observe how the "Rule B" prunes invalid branches early.

```
Decision Tree for n=3:
(O = Open count, C = Close count)

                    (0,0) ""
                      |
                    (1,0) "("
                   /        \
            (2,0) "(("      (1,1) "()"
           /      \           \
      (3,0) "((("  (2,1) "(()"  (2,1) "()("
        |           /    \       /    \
```

```
(3,1) "(((()" (3,1) "(((()" (2,2) "(())" (3,1) "()(("
      |           |          |           |
(3,2) "((()))" (3,2) "((()))" (3,2) "(()))" (3,2) "()(()"
      |           |          |           |
(3,3) "((()))" (3,3) "(()())" (3,3) "(())()" (3,3) "()(())" ... and so on
```

Why Top-Down first?

Top-down (Recursive Backtracking) is more intuitive here because the problem asks for **all permutations** that meet a specific criteria. You are essentially traversing a tree of possibilities. Bottom-up (DP) is possible but less "natural" because the subproblems (e.g., "what can I make with 2 pairs?") are used as building blocks inside larger strings, which requires more complex string concatenation logic.

---

3. Complexity Analysis

Time Complexity (TC)

This problem follows the **Catalan Number** sequence. The number of valid combinations is the n-th Catalan number.

```
TC derivation:
The number of valid sequences is: C(n) = (1 / (n + 1)) * (2n choose n)

In Big O terms, this grows roughly as:
O( (4^n) / (n * sqrt(n)) )

Why?
At each of the 2n steps, we have at most 2 choices.
However, most branches are pruned.
The actual number of valid nodes visited is proportional
to the n-th Catalan number.
```

Space Complexity (SC)

```
SC derivation:
1. Recursion Stack: The depth of the tree is 2n.
   So, O(n) stack space.
2. Output Storage: We store C(n) strings,
   each of length 2n.

Total SC: O(n) for the call stack.
(O(n * C(n)) if you count the result list).
```

---

4. Solution Code

Approach 1: Top-Down (Backtracking)

This is the preferred interview solution. It uses the "state" of open and close counts to decide the next move.

Python (Top-Down)

```python
def generateParenthesis(n):
    res = []

    # helper function to perform backtracking
    # s: current string being built
    # left: count of '(' used
    # right: count of ')' used
    def backtrack(s, left, right):
        # Base case: string is complete
        if len(s) == 2 * n:
            res.append(s)
            return

        # Rule A: Can we add an opening bracket?
        if left < n:
            backtrack(s + "(", left + 1, right)

        # Rule B: Can we add a closing bracket?
        # Must have fewer ')' than '(' to keep it well-formed
        if right < left:
            backtrack(s + ")", left, right + 1)

    backtrack("", 0, 0)
    return res
```

Javascript (Top-Down)

```javascript
/**
 * @param {number} n
 * @return {string[]}
 */
var generateParenthesis = function(n) {
    const result = [];

    /**
     * Recursive helper to explore valid paths
     * @param {string} current - The string built so far
     * @param {number} open - Number of '(' used
```

```
     * @param {number} close - Number of ')' used
     */
    const backtrack = (current, open, close) => {
        if (current.length === 2 * n) {
            result.push(current);
            return;
        }

        if (open < n) {
            backtrack(current + "(", open + 1, close);
        }
        if (close < open) {
            backtrack(current + ")", open, close + 1);
        }
    };

    backtrack("", 0, 0);
    return result;
};
```

Approach 2: Bottom-Up (Dynamic Programming)

The logic here is: `Result(n) = "(" + Result(i) + ")" + Result(n-1-i)` where `i` ranges from `0` to `n-1`. This effectively says every valid string of length `n` can be broken into a pair of brackets surrounding a valid string, followed by another valid string.

Python (Bottom-Up)

```python
def generateParenthesis(n):
    # dp[i] will store a list of all valid strings with i pairs
    dp = [[] for _ in range(n + 1)]
    dp[0] = [""] # Base case: 0 pairs is an empty string

    for i in range(1, n + 1):
        for j in range(i):
            # Form: ( left_part ) right_part
            # left_part uses 'j' pairs, right_part uses 'i - 1 - j' pairs
            for left in dp[j]:
                for right in dp[i - 1 - j]:
                    dp[i].append("(" + left + ")" + right)

    return dp[n]
```

Javascript (Bottom-Up)

```javascript
/**
 * @param {number} n
 * @return {string[]}
 */
var generateParenthesis = function(n) {
    // dp[i] is an array of valid strings for i pairs
    const dp = Array.from({ length: n + 1 }, () => []);
    dp[0] = [""];

    for (let i = 1; i <= n; i++) {
        for (let j = 0; j < i; j++) {
            // Combinations of ( dp[j] ) + dp[i-1-j]
            const leftPart = dp[j];
            const rightPart = dp[i - 1 - j];

            for (const left of leftPart) {
                for (const right of rightPart) {
                    dp[i].push("(" + left + ")" + right);
                }
            }
        }
    }
    return dp[n];
};
```

Technical Terms & Algorithms

- **Backtracking:** A refinement of brute force. It builds candidates for the solution and abandons a candidate ("backtracks") as soon as it determines the candidate cannot lead to a valid solution. In this problem, we backtrack when `close > open`.
- **Catalan Number:** A sequence of natural numbers that occurs in various counting problems. It specifically counts the number of ways to form valid expressions with n pairs of parentheses.
- **Pruning:** The act of cutting off branches in a decision tree (like not adding a `)` if it would make the string invalid). This is what makes the algorithm efficient.

Real-World / Interview Variations

1. **Google (HTML Tag Generator):** "Given `n` pairs of open/close tags (like `<div>` and `</div>`), generate all valid nested structures." (Same logic, different strings).

2. **Meta (Expression Validator/Generator):** "Given a set of operators and operands, generate all possible valid mathematical groupings using parentheses." (Variation: "241. Different Ways to Add Parentheses").
3. **Bloomberg (Financial Data Nesting):** "You have a stream of start-transaction and end-transaction logs. Write a function to simulate all possible sequences where no transaction ends before it starts and all transactions are closed."
4. **System Design Context:** Understanding these patterns helps in designing **parsers** (like JSON or XML parsers) where you must maintain a stack to ensure every opening symbol has a corresponding closing symbol.

77. Combinations

An L5 (Senior) or L6 (Staff) engineer doesn't just look for *a* solution; they look for the most efficient **backtracking template** that is readable, maintainable, and explains the "why" behind every state change.

For "77. Combinations," the core challenge is generating all possible subsets of a specific size k from a range n, without duplicates and without caring about order (e.g., [1,2] is the same as [2,1]).

---

1. Problem Explanation

Given two integers **n** and **k**, return all possible combinations of **k** numbers chosen from the range **[1, n]**.

Key Rules:

- **No duplicates:** Once you pick 1 and 2, you cannot pick 2 and 1.
- **Fixed length:** Every combination must be exactly length **k**.
- **Range:** You only use numbers from 1 up to **n**.

Example: $n = 4$, $k = 2$ Output: `[[1,2], [1,3], [1,4], [2,3], [2,4], [3,4]]`

---

2. Solution Explanation (Backtracking)

An L5+ engineer views this as a **Decision Tree**. At each step, you decide which number to include next. To avoid duplicates like `[2,1]` after you've already found `[1,2]`, we enforce an **ascending order rule**: the next number we pick must always be greater than the last one we picked.

The Mental Model: The State-Space Tree

Imagine we are building a path. If $n = 4$ and $k = 2$:

```
Level 0:                 [ ]  (Empty Start)
                        /  |  \   \
Level 1:              [1]  [2]  [3]  [4]
                     / | \  | \   |   (No more options for 4)
Level 2:       [1,2][1,3][1,4] [2,3][2,4] [3,4]
```

Step-by-Step Visualization (n=4, k=2)

Step 1: Start at 1 We pick `1`. Now we need 1 more number ($k = 1$ remaining). We can only pick from `[2, 3, 4]`.

```
Current Path: [1]
Options left: [2, 3, 4]
Result so far: []
```

Step 2: Pick 2 Path becomes `[1, 2]`. Length is $k = 2$. **SUCCESS!** Add to results.

```
Current Path: [1, 2]  <-- Success!
Result so far: [[1, 2]]
```

Step 3: Backtrack Pop 2 out. Now try the next option: `3`.

```
Current Path: [1, 3]  <-- Success!
Result so far: [[1, 2], [1, 3]]
```

Step 4: Pruning (The "L6" Optimization) An experienced engineer realizes you don't always need to check every number. If you need 2 more numbers to finish your combination, but there is only 1 number left in the range, you can stop immediately. This is called **Pruning**.

ASCII Backtracking Trace ($n = 4, k = 2$)

```
DFS(start=1, path=[])

  Pick 1 -> DFS(start=2, path=[1])
      Pick 2 -> [1,2] (SAVE)
      Pick 3 -> [1,3] (SAVE)
      Pick 4 -> [1,4] (SAVE)

  Pick 2 -> DFS(start=3, path=[2])
      Pick 3 -> [2,3] (SAVE)
      Pick 4 -> [2,4] (SAVE)

  Pick 3 -> DFS(start=4, path=[3])
      Pick 4 -> [3,4] (SAVE)
```

```
Pick 4 -> DFS(start=5, path=[4])
    (Loop doesn't run, no numbers left)
```

---

3. Complexity Analysis

Time Complexity (TC)

The number of combinations is determined by the binomial coefficient "n choose k".

`TC = O( k * (n! / (k!(n-k)!)) )`

- **Why?** There are `n! / (k!(n-k)!)` distinct combinations.
- **The 'k' factor:** For each valid combination found, we spend `O(k)` time to copy the current path into our result list.

Space Complexity (SC)

`SC = O(k)`

- **Why?** The recursion stack goes as deep as `k`. The temporary `path` array also stores `k` elements. (Note: We usually don't count the output list in SC analysis for interviews unless asked).

---

4. Solution Code

Top-Down Approach (Backtracking/Recursion)

This is the most intuitive "Google-style" solution because it maps directly to the decision tree.

Python (Top-Down)

```python
def combine(n, k):
    res = []

    # backtrack(start, path) helps us explore all combinations
    # start: the number we begin picking from (to avoid duplicates)
    # path: the current combination being built
    def backtrack(start, path):
        # Base Case: If the combination is the right length, save it
        if len(path) == k:
            res.append(list(path))
            return

        # Optimization (Pruning):
```

```
        # Only loop if there are enough numbers left to reach k
        # numbers_needed = k - len(path)
        # numbers_available = n - i + 1
        for i in range(start, n + 1):
            path.append(i)          # 1. Choose
            backtrack(i + 1, path) # 2. Explore
            path.pop()              # 3. Un-choose (Backtrack)

    backtrack(1, [])
    return res
```

JavaScript (Top-Down)

```
var combine = function(n, k) {
    const res = [];

    /**
     * @param {number} start - The next number to consider
     * @param {number[]} path - Current building combination
     */
    function backtrack(start, path) {
        if (path.length === k) {
            res.push([...path]); // Copy path
            return;
        }

        for (let i = start; i <= n; i++) {
            path.push(i);
            backtrack(i + 1, path);
            path.pop(); // Backtrack step
        }
    }

    backtrack(1, []);
    return res;
};
```

Bottom-Up Approach (Iterative / Lexicographic)

A Senior engineer might use this to avoid recursion limit issues. We treat the combination as a sequence that we "increment" to the next valid state.

Python (Bottom-Up)

```python
def combine(n, k):
    # Initialize the first combination: [1, 2, ... k]
    nums = list(range(1, k + 1)) + [n + 1]
    res = []
    j = 0
    while j < k:
        res.append(nums[:k])
        j = 0
        # Find the first number that can be incremented
        while j < k and nums[j + 1] == nums[j] + 1:
            nums[j] = j + 1
            j += 1
        nums[j] += 1
    return res
```

JavaScript (Bottom-Up)

```javascript
var combine = function(n, k) {
    let result = [];
    let path = Array.from({ length: k }, (_, i) => i + 1);
    path.push(n + 1); // Sentinel value

    let j = 0;
    while (j < k) {
        result.push(path.slice(0, k));
        j = 0;
        // Logic to find the next lexicographical combination
        while (j < k && path[j + 1] === path[j] + 1) {
            path[j] = j + 1;
            j++;
        }
        path[j]++;
    }
    return result;
};
```

Terminology: Backtracking

Backtracking is a refined version of Brute Force. Instead of generating every possible permutation and filtering, it builds the solution one piece at a time. If it realizes a branch cannot possibly lead to a valid solution (like needing 2 numbers but only having 1 left), it "backtracks" to the previous step and tries a different path.

Real-World / Interview Variations

Companies like **Google, Meta, and Bloomberg** rarely ask this directly anymore. Instead, they mask it in scenarios:

1. **Google (Team Formation):** "Given a list of employees with different skills, find all possible teams of size K that satisfy a certain diversity metric." (This is combinations + a filter condition).
2. **Meta (Ad Targeting):** "Pick K target interests from a list of N available interests to test an ad campaign. Return all possible combinations of interests."
3. **Bloomberg (Portfolio Construction):** "A trader wants to pick 3 stocks from a list of 10 to create a basket. Show all possible baskets."
4. **General (Sum Variation):** "Find all combinations of K numbers that sum up to a target value" (Combination Sum III).

46. Permutations

To solve "46. Permutations" like a high-performing L5 or L6 engineer, one doesn't just write code that "works." You focus on **memory efficiency**, **recursion tree pruning**, and **clarity of state**.

At this level, the problem is viewed as a **State Space Search**.

---

1. Problem Explanation

The goal is to take a list of unique integers (e.g., `[1, 2, 3]`) and return every possible way to arrange them.

The Constraint: * You must use **all** numbers in each arrangement.

- The order matters. `[1, 2]` is different from `[2, 1]`.

Real-world analogy: If you have 3 books on a shelf, how many ways can you line them up?

1. Pick the 1st book (3 choices).
2. Pick the 2nd book (2 choices left).
3. Pick the 3rd book (1 choice left). Total = 3 * 2 * 1 = 6 ways.

---

2. Solution Explanation: Backtracking

The most efficient way to solve this is **Backtracking**. Think of it as a "Decision Tree" where each level of the tree represents a slot in our resulting array.

The Strategy: "Swap and Proceed"

Instead of creating new arrays at every step (which is slow), an L6 engineer would use **in-place swapping**. This minimizes memory overhead.

Step-by-Step Visualization [1, 2, 3]

We use a pointer `start`. Everything to the left of `start` is "fixed," and everything to the right is "available to swap."

Level 0: Start = 0 We can swap the element at index 0 with index 0, 1, or 2.

```
Decision Tree for [1, 2, 3]
---------------------------

Initial State: [1, 2, 3], start = 0

Phase 1: Swap index 0 with 0 (Fix 1 at first spot)
   [1, 2, 3]
    ^
    |-- Now move to start = 1. Remaining options: swap index 1 with 1 or 2.
    |
    |-- Swap 1 with 1: [1, 2, 3] -> (start=2) -> [1, 2, 3] (RESULT!)
    |-- Swap 1 with 2: [1, 3, 2] -> (start=2) -> [1, 3, 2] (RESULT!)

Phase 2: Swap index 0 with 1 (Fix 2 at first spot)
   [2, 1, 3]
    ^
    |-- Now move to start = 1. Remaining options: swap index 1 with 1 or 2.
    |
    |-- Swap 1 with 1: [2, 1, 3] -> (start=2) -> [2, 1, 3] (RESULT!)
    |-- Swap 1 with 2: [2, 3, 1] -> (start=2) -> [2, 3, 1] (RESULT!)

Phase 3: Swap index 0 with 2 (Fix 3 at first spot)
   [3, 2, 1]
    ^
    |-- Now move to start = 1. Remaining options: swap index 1 with 1 or 2.
    |
    |-- Swap 1 with 1: [3, 2, 1] -> (start=2) -> [3, 2, 1] (RESULT!)
    |-- Swap 1 with 2: [3, 1, 2] -> (start=2) -> [3, 1, 2] (RESULT!)
```

The "Backtrack" Visualized

The "magic" happens when the recursion returns. If we swapped 1 and 2 to explore a path, we **must swap them back** before trying the next path.

```
[ 1, 2, 3 ]  (Start)
      |
```

```
   SWAP(0, 1)
      |
[ 2, 1, 3 ]  (Explore this branch...)
      |
   BACKTRACK (SWAP(0, 1) again)
      |
[ 1, 2, 3 ]  (Back to original state to try SWAP(0, 2))
```

---

3. Time and Space Complexity Analysis

Time Complexity (TC)

We are generating N! (N factorial) permutations. At each leaf node of our tree, we copy the array of size N into our result list.

```
TC Derivation:
Nodes in Tree:
Level 0: 1 node
Level 1: N nodes
Level 2: N * (N-1) nodes
...
Level N: N! nodes

Total Nodes ~ N!
Work at each leaf: O(N) to copy array.

Total TC = O(N * N!)
```

Space Complexity (SC)

We consider the recursion stack and the storage for the output.

```
SC Derivation:
Recursion Depth: O(N)  (The height of the tree)
Output Storage:  O(N * N!) (To store all permutations)

If excluding output: O(N)
If including output: O(N * N!)
```

---

4. Solution Code

Top-Down Approach (Backtracking with Swapping)

This is the "Gold Standard" for performance as it avoids extra memory allocations for "used" sets.

Python

```python
class Solution:
    def permute(self, nums):
        res = []

        # backtrack(index) explores all permutations
        # for the subarray starting from 'index'
        def backtrack(start):
            # Base case: if we reached the end, we found a permutation
            if start == len(nums):
                res.append(nums[:])
                return

            for i in range(start, len(nums)):
                # 1. Action: Swap the current element with the 'start' element
                nums[start], nums[i] = nums[i], nums[start]

                # 2. Recurse: Move to the next position
                backtrack(start + 1)

                # 3. Backtrack: Undo the swap to restore the original array
                nums[start], nums[i] = nums[i], nums[start]

        backtrack(0)
        return res
```

JavaScript

```javascript
var permute = function(nums) {
    const res = [];

    /**
     * Recursive function to swap elements and build permutations.
     * @param {number} start - The current index we are fixing.
     */
    function backtrack(start) {
        if (start === nums.length) {
            res.push([...nums]);
            return;
        }

        for (let i = start; i < nums.length; i++) {
            // Swap
            [nums[start], nums[i]] = [nums[i], nums[start]];
```

```
            // Recurse
            backtrack(start + 1);

            // Backtrack
            [nums[start], nums[i]] = [nums[i], nums[start]];
        }
    }

    backtrack(0);
    return res;
};
```

Bottom-Up Approach (Iterative / BFS-style)

This approach builds permutations by inserting the next number into every possible gap of existing permutations.

Python

```
class Solution:
    def permute(self, nums):
        # Start with an empty permutation in a list
        perms = [[]]

        for n in nums:
            new_perms = []
            for p in perms:
                # Insert the new number 'n' into every possible index of 'p'
                for i in range(len(p) + 1):
                    new_p = p[:i] + [n] + p[i:]
                    new_perms.append(new_p)
            perms = new_perms

        return perms
```

JavaScript

```
var permute = function(nums) {
    let perms = [[]];

    for (let n of nums) {
        let new_perms = [];
        for (let p of perms) {
            // Create a new permutation by inserting n at every index
            for (let i = 0; i <= p.length; i++) {
```

```
                let copy = [...p];
                copy.splice(i, 0, n);
                new_perms.push(copy);
            }
        }
        perms = new_perms;
    }

    return perms;
};
```

---

5. Key Terms & Concepts

- **Backtracking:** An algorithmic technique that builds a solution incrementally and abandons a path ("backtracks") as soon as it determines that this path cannot lead to a valid full solution.
- **State Space Tree:** A visual representation of all possible states the problem can be in. In this case, it's the tree of all possible swaps.
- **In-Place Transformation:** Modifying the input array itself (and then reverting it) rather than creating new copies. This is what differentiates an L6 solution from a junior solution; it shows awareness of **Space Complexity** and **Garbage Collection** overhead.

---

6. Real-World / Indirect Interview Variations

Google and Meta rarely ask "Find all permutations" directly anymore. Instead, they disguise it:

1. **The "Matchstick" Problem (Google):** Given N matchsticks of different lengths, can you form a square? (This is a permutation/partition problem).
2. **Task Scheduler (Meta):** If you have N tasks with dependencies, find all valid sequences in which they can be executed.
3. **Optimal Route (Bloomberg):** Given a set of city coordinates, find the shortest path that visits all cities exactly once (Traveling Salesperson - requires exploring permutations).
4. **Letter Case Permutation:** Instead of numbers, you are given a string like "a1b2" and told to generate all variations of upper/lower case.

51. N-Queens

An L5/L6 engineer doesn't just jump into coding; they focus on **scalability, constraints, and pattern recognition**. For the N-Queens problem, they

recognize this as a "Constraint Satisfaction Problem" that requires a systematic search of the state space.

Here is how a top-tier engineer would break this down.

---

1. Problem Explanation

The N-Queens problem asks us to place `n` queens on an `n x n` chessboard such that no two queens attack each other. In chess, a queen can attack vertically, horizontally, and diagonally.

The Constraints:

- Only **one** queen per row.
- Only **one** queen per column.
- Only **one** queen per "positive" diagonal (bottom-left to top-right).
- Only **one** queen per "negative" diagonal (top-left to bottom-right).

---

2. Solution Explanation

An L5 engineer would immediately notice that since every row must have exactly one queen, we can iterate row-by-row. This reduces the problem to finding the correct **column** for each row.

The Core Logic: Backtracking

We use a "Recursive Backtracking" approach. We try a position; if it's valid, we move to the next row. If we hit a dead end, we "backtrack" (remove the queen) and try the next possible column.

Non-Trivial Insight: Diagonal Indexing

Checking every cell for safety is slow ($O(n)$). An efficient solution uses Sets to track occupied columns and diagonals in $O(1)$ time.

1. **Columns:** Tracked by the column index `c`.
2. **Positive Diagonals (`/`)**: In any `/` diagonal, the sum of row and column (`r + c`) is constant.
3. **Negative Diagonals (`\`)**: In any `\` diagonal, the difference of row and column (`r - c`) is constant.

Step-by-Step ASCII Visualization (N=4)

Step 1: Start at Row 0. Try Col 0.

```
Q . . .  (Placed at 0,0)
. . . .
. . . .
. . . .
Sets: Cols{0}, PosDiag{0}, NegDiag{0}
```

Step 2: Row 1. Col 0 and 1 are attacked. Try Col 2.

```
Q . . .
. . Q .  (Placed at 1,2)
. . . .
. . . .
Sets: Cols{0,2}, PosDiag{0,3}, NegDiag{0,-1}
```

Step 3: Row 2. All columns (0,1,2,3) are under attack!

- Col 0: Attacked by (0,0)
- Col 1: Attacked by (1,2) diagonally
- Col 2: Attacked by (1,2)
- Col 3: Attacked by (1,2) diagonally **ACTION: BACKTRACK to Row 1.**

Step 4: Back in Row 1. Move queen from Col 2 to Col 3.

```
Q . . .
. . . Q  (Placed at 1,3)
. . . .
. . . .
```

Step 5: Row 2. Try Col 1.

```
Q . . .
. . . Q
. Q . .  (Placed at 2,1)
. . . .
Sets: Cols{0,3,1}, PosDiag{0,4,3}, NegDiag{0,-2,1}
```

Step 6: Row 3. All columns attacked. BACKTRACK to Row 2, then Row 1, then Row 0.

Step 7: Back in Row 0. Move queen to Col 1. Repeat... Final valid configuration found:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

---

3. Time and Space Complexity Analysis

The engineer derives complexity by visualizing the "Decision Tree."

Time Complexity (TC)

In the first row, we have N choices. In the second, roughly $N-2$ choices. This forms a factorial-style progression.

```
TC Derivation:
Level 0: N choices
Level 1: N-2 choices
Level 2: N-4 choices
...
Total Nodes approx = N * (N-2) * (N-4) ...
Complexity = O(N!)
```

Note: While actually upper-bounded by O(N!), the pruning makes it much faster in practice.

Space Complexity (SC)

We store the board, the recursion stack, and the sets for lookups.

```
SC Derivation:
Board Storage     = N * N = O(N^2)
Recursion Stack   = Max depth N = O(N)
Lookup Sets       = 3 sets of size approx N = O(N)
Total Space       = O(N^2)
```

---

4. Solution Code

N-Queens is traditionally a **Top-Down** (Recursive) problem. A "Bottom-Up" (Iterative) approach for N-Queens mimics the recursion stack manually using a loop and a stack.

Python Implementation

```python
# Top-Down (Recursive Backtracking) - Most Intuitive
def solveNQueens_TopDown(n):
    res = []
    board = [["." for _ in range(n)] for _ in range(n)]

    cols = set()
    posDiag = set() # (r + c)
    negDiag = set() # (r - c)
```

```
    # backtrack(r) attempts to place a queen in row 'r'
    def backtrack(r):
        if r == n:
            # Found a valid configuration
            copy = ["".join(row) for row in board]
            res.append(copy)
            return

        for c in range(n):
            if c in cols or (r + c) in posDiag or (r - c) in negDiag:
                continue

            # Place queen
            cols.add(c)
            posDiag.add(r + c)
            negDiag.add(r - c)
            board[r][c] = "Q"

            backtrack(r + 1)

            # Undo placement (Backtrack)
            cols.remove(c)
            posDiag.remove(r + c)
            negDiag.remove(r - c)
            board[r][c] = "."

    backtrack(0)
    return res

# Bottom-Up (Iterative using manual Stack)
def solveNQueens_BottomUp(n):
    res = []
    # stack stores (current_row, current_columns_placed_list)
    stack = [(0, [])]

    while stack:
        r, queens = stack.pop()
        if r == n:
            # Convert queen positions to board format
            board = []
            for col in queens:
                board.append("." * col + "Q" + "." * (n - col - 1))
            res.append(board)
            continue

        for c in range(n - 1, -1, -1): # Reverse to maintain order similar to recursion
```

```
            # Check safety manually or with sets derived from 'queens'
            safe = True
            for prev_r, prev_c in enumerate(queens):
                if c == prev_c or abs(r - prev_r) == abs(c - prev_c):
                    safe = False
                    break
            if safe:
                stack.append((r + 1, queens + [c]))
    return res
```

JavaScript Implementation

```
/**
 * Top-Down Approach
 * Explores choices row by row using recursion.
 */
function solveNQueens_TopDown(n) {
    const res = [];
    const board = Array.from({ length: n }, () => Array(n).fill("."));

    const cols = new Set();
    const posDiag = new Set();
    const negDiag = new Set();

    function backtrack(r) {
        if (r === n) {
            res.push(board.map(row => row.join("")));
            return;
        }

        for (let c = 0; c < n; c++) {
            if (cols.has(c) || posDiag.has(r + c) || negDiag.has(r - c)) continue;

            board[r][c] = "Q";
            cols.add(c);
            posDiag.add(r + c);
            negDiag.add(r - c);

            backtrack(r + 1);

            board[r][c] = ".";
            cols.remove(c); // Fixed logic: backtracking clean up
            cols.delete(c);
            posDiag.delete(r + c);
            negDiag.delete(r - c);
        }
```

```
    }
    backtrack(0);
    return res;
}

/**
 * Bottom-Up / Iterative Approach
 * Uses a stack to simulate the call tree.
 */
function solveNQueens_BottomUp(n) {
    const res = [];
    const stack = [[0, []]]; // [row, cols_placed]

    while (stack.length > 0) {
        const [r, queens] = stack.pop();

        if (r === n) {
            res.push(queens.map(c => ".".repeat(c) + "Q" + ".".repeat(n - c - 1)));
            continue;
        }

        for (let c = n - 1; c >= 0; c--) {
            let safe = true;
            for (let prevR = 0; prevR < queens.length; prevR++) {
                let prevC = queens[prevR];
                if (c === prevC || Math.abs(r - prevR) === Math.abs(c - prevC)) {
                    safe = false;
                    break;
                }
            }
            if (safe) {
                stack.push([r + 1, [...queens, c]]);
            }
        }
    }
    return res;
}
```

Note 1: Terms and Techniques

- **Backtracking:** A refinement of brute force. It builds candidates toward a solution but abandons a candidate ("backtracks") as soon as it determines the candidate cannot lead to a valid solution. This is the "Gold Standard" for combinatorial problems.

- **Pruning:** The act of not exploring branches of a tree that we know are invalid (e.g., if a column is already taken).

Note 2: Real-World / Indirect Interview Variations

Google and Meta often wrap N-Queens in "story" problems:

1. **The "Safe Warehouse" (Google):** You have a warehouse grid and need to place security cameras (like Queens) that can see in all 8 directions. What is the maximum number of cameras you can place such that no camera is "watching" another?
2. **The "Non-Attacking Kings/Knights" (Bloomberg):** Similar to N-Queens, but with different movement patterns. This tests if your "safety check" logic is modular.
3. **Service Mesh Routing (Meta):** You have N services that need to be assigned to N servers. Certain services cannot be on the same "diagonal" of a network latency matrix. Find all valid assignments.

52. N-Queens II

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just look for a way to pass the test cases; they look for the most efficient way to prune the search space and manage state. For **N-Queens II**, the key is not just recursion, but **Backtracking with Bitmasking** or optimized boolean lookups.

---

1. Problem Explanation

The goal is to place `n` queens on an `n x n` chessboard such that no two queens attack each other. In chess, a queen can attack horizontally, vertically, and diagonally.

Because we must place `n` queens on `n` rows, we know exactly one queen must exist in every row. The challenge is picking a column for each row that doesn't conflict with queens already placed in previous rows.

The Constraints:

- **Vertical:** No two queens in the same column.
- **Diagonal 1 (Top-Left to Bottom-Right):** Differences between row and column indices are constant.
- **Diagonal 2 (Top-Right to Bottom-Left):** Sums of row and column indices are constant.

---

2. Solution Explanation

We use **Backtracking**. Think of it as a Depth-First Search (DFS) through a tree of possibilities.

The Visualization of State Tracking

Instead of checking the whole board every time, we maintain three "lookup" sets (or bitmasks) to tell us if a column or diagonal is already under attack.

Example: n = 4 We try to place a queen in Row 0, then Row 1, and so on.

Step 1: Row 0 We place a Queen at (0, 0).

```
Q . . .  <- Row 0 (Placed at Col 0)
. . . .
. . . .
. . . .
```

- Occupied Columns: {0}
- Occupied Diagonals (row-col): {0}
- Occupied Anti-Diagonals (row+col): {0}

Step 2: Row 1 We can't use Col 0 (Vertical) or Col 1 (Diagonal). Let's try Col 2.

```
Q . . .
. . Q .  <- Row 1 (Placed at Col 2)
. . . .
. . . .
```

- Occupied Columns: {0, 2}
- Occupied Diagonals (row-col): {0, -1}
- Occupied Anti-Diagonals (row+col): {0, 3}

Step 3: Row 2 Check all columns:

- Col 0: Taken
- Col 1: (2,1) -> row-col = 1 (Clean), row+col = 3 (TAKEN!)
- Col 2: Taken
- Col 3: (2,3) -> row-col = -1 (TAKEN!) **Result:** No spots left! Backtrack to Row 1.

Step 4: Backtrack to Row 1 Move Queen from Col 2 to Col 3.

```
Q . . .
. . . Q  <- Row 1 (Placed at Col 3)
. . . .
. . . .
```

Why this is "Non-Trivial"

The diagonal math is the clever part:

1. **Main Diagonal:** For any cell `(r, c)`, all cells on its top-left to bottom-right diagonal have the same value for `r - c`.
2. **Anti-Diagonal:** All cells on the top-right to bottom-left diagonal have the same value for `r + c`.

---

3. Time and Space Complexity Analysis

```
TIME COMPLEXITY DERIVATION:
Each row 'n' has at most 'n' choices.
However, each choice reduces the available choices for the next row.
Upper Bound: O(n!)
The actual number of nodes explored is significantly lower
due to pruning (checking conflicts).
Final: O(n!)

SPACE COMPLEXITY DERIVATION:
1. Recursion Stack: O(n) depth for n rows.
2. State tracking (Sets/Booleans):
   - Columns: O(n)
   - Diagonals: O(2n - 1)
   - Anti-Diagonals: O(2n - 1)
Final: O(n)
```

---

4. Solution Code

Top-Down (Recursive Backtracking)

This is the most intuitive approach for N-Queens because the problem is naturally a state-space search.

Python (Top-Down)

```python
class Solution:
    def totalNQueens(self, n: int) -> int:
        # We use sets for O(1) lookup of attacked paths
        cols = set()
        pos_diag = set() # (r + c)
        neg_diag = set() # (r - c)

        def backtrack(r):
            # Base Case: All rows filled
```

```
            if r == n:
                return 1

            count = 0
            for c in range(n):
                # Check if the current position is under attack
                if c in cols or (r + c) in pos_diag or (r - c) in neg_diag:
                    continue

                # 'Choose' - Place queen and mark paths
                cols.add(c)
                pos_diag.add(r + c)
                neg_diag.add(r - c)

                # 'Explore' - Move to the next row
                count += backtrack(r + 1)

                # 'Unchoose' - Backtrack for other possibilities
                cols.remove(c)
                pos_diag.remove(r + c)
                neg_diag.remove(r - c)

            return count

        return backtrack(0)
```

JavaScript (Top-Down)

```
/**
 * Uses a recursive DFS approach to explore all valid queen placements.
 * @param {number} n - The size of the board
 */
var totalNQueens = function(n) {
    let count = 0;
    const cols = new Set();
    const posDiag = new Set();
    const negDiag = new Set();

    function backtrack(r) {
        if (r === n) {
            count++;
            return;
        }

        for (let c = 0; c < n; c++) {
            if (cols.has(c) || posDiag.has(r + c) || negDiag.has(r - c)) {
```

```
                continue;
            }

            cols.add(c);
            posDiag.add(r + c);
            negDiag.add(r - c);

            backtrack(r + 1);

            cols.delete(c);
            posDiag.delete(r + c);
            negDiag.delete(r - c);
        }
    }

    backtrack(0);
    return count;
};
```

Bottom-Up (Iterative with Stack)

N-Queens is rarely solved "Bottom-Up" in the traditional DP sense (table filling) because we don't need results of subproblems to build the current one; we are searching for valid permutations. However, we can simulate the recursion using an explicit stack.

Python (Bottom-Up / Iterative)

```python
def totalNQueensIterative(n):
    # Stack stores (row_index, current_cols, current_posDiag, current_negDiag)
    # We use frozensets for hashability in the stack if needed,
    # but here simple tuples/sets work.
    stack = [(0, set(), set(), set())]
    count = 0

    while stack:
        r, cols, pos, neg = stack.pop()

        if r == n:
            count += 1
            continue

        for c in range(n):
            if c not in cols and (r+c) not in pos and (r-c) not in neg:
                # Create new states for the next "level"
                new_cols = cols | {c}
```

```
                new_pos = pos | {r + c}
                new_neg = neg | {r - c}
                stack.append((r + 1, new_cols, new_pos, new_neg))
    return count
```

JavaScript (Bottom-Up / Iterative)

```
/**
 * Simulates the backtracking process using an explicit stack.
 */
var totalNQueensIterative = function(n) {
    let count = 0;
    // Each stack element: [row, colsSet, posDiagSet, negDiagSet]
    let stack = [[0, new Set(), new Set(), new Set()]];

    while (stack.length > 0) {
        let [r, cols, posDiag, negDiag] = stack.pop();

        if (r === n) {
            count++;
            continue;
        }

        for (let c = 0; c < n; c++) {
            if (!cols.has(c) && !posDiag.has(r + c) && !negDiag.has(r - c)) {
                let nextCols = new Set(cols).add(c);
                let nextPos = new Set(posDiag).add(r + c);
                let nextNeg = new Set(negDiag).add(r - c);
                stack.push([r + 1, nextCols, nextPos, nextNeg]);
            }
        }
    }
    return count;
};
```

Key Terms

- **Backtracking:** An algorithmic technique that builds candidates for solutions incrementally and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution.
- **Pruning:** Cutting off branches of the search tree that are guaranteed to be invalid. In this problem, checking the sets *before* recursing is pruning.

Real-World Interview Variants (Google, Meta, Bloomberg)

These companies rarely ask the "textbook" N-Queens. They ask variants like:

1. **The "Safety Guard" Problem:** Placing security cameras in a museum grid where cameras see in 8 directions, but some walls block the view (Google).
2. **Resource Allocation:** Scheduling `n` tasks on `n` processors where certain tasks conflict if run at the same time or on specific types of hardware (Meta).
3. **The "Non-Attacking Kings/Knights":** Similar logic but with different movement patterns (Bloomberg).
4. **Validating a Sudoku Board:** This is essentially a smaller-scale version of the same "set-based" constraint checking.

78. Subsets

An L5 (Senior) or L6 (Staff) engineer at Google doesn't just look for *a* solution; they look for the most extensible, memory-efficient, and "idiomatic" way to handle combinatorial search. For **Leetcode 78: Subsets**, they recognize this as a "Power Set" problem, which is the foundation for almost all backtracking and bit manipulation challenges.

---

1. Problem Explanation

Given an integer array `nums` of **unique** elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets.

In plain English: You are standing in front of a buffet. For every single item offered, you have a binary choice: **Take it** or **Leave it**.

If the input is `[1, 2, 3]`, your choices look like this:

- Empty set: `[]`
- Single items: `[1], [2], [3]`
- Pairs: `[1,2], [1,3], [2,3]`
- The whole thing: `[1,2,3]`

The total number of subsets will always be 2 raised to the power of N (where N is the number of elements).

---

2. Solution Explanation (Backtracking & Cascading)

We will focus on the **Backtracking (Decision Tree)** approach as it is the most intuitive for L5+ interviews, followed by the **Cascading (Iterative)** approach.

The Decision Tree Strategy

Think of this as a depth-first traversal. At each level of the tree, we decide whether to include the current number in our "current path."

]

Step-by-Step ASCII Visualization (`nums = [1, 2]`) We start with an empty path `[]` and an index `0`.

Level 0: Decision for nums[0] (which is 1)

```
State: index=0, path=[]
      /              \
    INCLUDE 1       EXCLUDE 1
    /                  \
 path=[1]              path=[]
```

Level 1: Decision for nums[1] (which is 2)

```
           [ROOT]
          /      \
      [1]          []          <-- Choices for '1'
     /   \        /   \
  [1,2]  [1]    [2]   []       <-- Choices for '2'
    ^     ^      ^     ^
    |     |      |     |
   Leaf  Leaf   Leaf  Leaf     <-- These are your 4 subsets!
```

Detailed Step-by-Step Trace for `[1, 2, 3]`

```
START: backtrack(0, [])
|
|-- Choice: Include 1?
|   |-- YES: path=[1] -> backtrack(1, [1])
|   |   |-- Include 2?
|   |   |   |-- YES: path=[1, 2] -> backtrack(2, [1, 2])
|   |   |   |   |-- Include 3?
|   |   |   |   |   |-- YES: [1, 2, 3] -> (Base Case: Add to Results)
|   |   |   |   |   |-- NO:  [1, 2]    -> (Base Case: Add to Results)
|   |   |-- NO:  path=[1] -> backtrack(2, [1])
|   |   |   |-- Include 3?
|   |   |   |   |-- YES: [1, 3]    -> (Base Case: Add to Results)
|   |   |   |   |-- NO:  [1]       -> (Base Case: Add to Results)
|
|-- Choice: Include 1?
|   |-- NO: path=[] -> backtrack(1, [])
|   |   |-- Include 2?
|   |   |   |-- YES: path=[2] -> backtrack(2, [2])
```

```
|   |   |   |   |-- Include 3?
|   |   |   |   |   |-- YES: [2, 3]    -> (Base Case: Add to Results)
|   |   |   |   |   |-- NO:  [2]       -> (Base Case: Add to Results)
|   |   |-- NO:  path=[] -> backtrack(2, [])
|   |   |   |-- Include 3?
|   |   |   |   |-- YES: [3]       -> (Base Case: Add to Results)
|   |   |   |   |-- NO:  []        -> (Base Case: Add to Results)
```

---

3. Time and Space Complexity Analysis

For an input of size `N`, we are generating `2^N` subsets. For each subset, we perform a copy operation into our result list, which takes `O(N)` time.

Time Complexity (TC)

```
Total Subsets: 2^N
Work per subset: N (copying the array)
---------------------------------------
Total TC: O(N * 2^N)
```

Space Complexity (SC)

```
Recursion Stack Depth: N (The height of our decision tree)
Temporary Path Storage: N (To store the current subset being built)
---------------------------------------
Total SC: O(N)
(Note: We usually ignore the output list space in complexity analysis)
```

---

4. Solution Code

Approach A: Top-Down (Recursive Backtracking)

This is the "standard" interview approach. It mimics the decision tree.

Python

```python
def subsets(nums):
    res = []

    # helper function to perform DFS/Backtracking
    def backtrack(index, path):
        # Base case: if we've made a choice for every number
        if index == len(nums):
            res.append(list(path)) # append a copy of the current path
```

```
            return

        # Choice 1: Include nums[index]
        path.append(nums[index])
        backtrack(index + 1, path)

        # Choice 2: Exclude nums[index]
        path.pop() # Undo the choice (backtrack)
        backtrack(index + 1, path)

    backtrack(0, [])
    return res
```

JavaScript

```
var subsets = function(nums) {
    const res = [];

    // helper function using recursion and backtracking
    const backtrack = (index, path) => {
        if (index === nums.length) {
            res.push([...path]); // push a shallow copy
            return;
        }

        // Option 1: Include the current element
        path.push(nums[index]);
        backtrack(index + 1, path);

        // Option 2: Exclude the current element
        path.pop();
        backtrack(index + 1, path);
    };

    backtrack(0, []);
    return res;
};
```

Approach B: Bottom-Up (Iterative Cascading)

This is often more efficient in practice as it avoids the overhead of the recursion stack. We start with an empty set and for every new number, we clone all existing sets and add the new number to them.

Python

```python
def subsets_iterative(nums):
    # Start with the empty set
    result = [[]]

    for num in nums:
        # For every existing subset, create a new one including 'num'
        new_subsets = []
        for curr in result:
            new_subsets.append(curr + [num])

        # Add the new subsets to our total list
        result.extend(new_subsets)

    return result
```

JavaScript

```javascript
var subsetsIterative = function(nums) {
    let result = [[]];

    for (let num of nums) {
        let size = result.length;
        // Iterate through existing subsets and add the current number
        for (let i = 0; i < size; i++) {
            let nextSubset = [...result[i], num];
            result.push(nextSubset);
        }
    }

    return result;
};
```

Technical Terms & Context

- **Backtracking:** A refinement of brute force. It builds a solution piece by piece and "backtracks" (removes the last piece) as soon as it determines that the current path cannot be completed or after it has recorded a valid solution.
- **Cascading:** An iterative method where you build the solution set by adding to what you already have. It's essentially "Mathematical Induction" in code form.
- **Power Set:** The set of all subsets of a set, including the empty set and the set itself.

Real-World Interview Variations

Google, Meta, and Bloomberg rarely ask the "raw" version. They use these variations:

1. **Google (File Permissions):** "Given a list of user roles `[Admin, Editor, Viewer]`, generate all possible permission combinations a user might have." (This is exactly the Subsets problem).
2. **Meta (Privacy Settings):** "A user has N privacy toggles. List all possible configurations of these toggles to test system edge cases."
3. **Bloomberg (Portfolio Construction):** "Given a set of stocks, find all possible portfolio combinations that contain at least 2 stocks but no more than 5." (Subsets with a size constraint).
4. **Generic Variation (Subsets II):** "What if the input array contains duplicates (e.g., `[1, 2, 2]`)? How do you ensure the output doesn't have duplicate subsets?" (Requires sorting and a skipping condition).

90. Subsets II

For a top-tier L5 or L6 engineer at Google, the key to solving "Subsets II" isn't just getting the code to pass; it's about **clarity of thought regarding the state space** and handling the **symmetry breaking** required to avoid duplicates.

---

1. Problem Explanation

The goal is to find all unique subsets (the power set) of an integer array `nums` that contains **duplicates**.

The Catch: If the input is `[1, 2, 2]`, a standard subset algorithm would produce two identical `[1, 2]` subsets because it treats the first `2` and the second `2` as distinct entities. We must ensure our result only contains one `[1, 2]`.

The Strategic Insight: To handle duplicates efficiently, we must **sort** the input. Sorting allows us to keep identical numbers next to each other, making it easy to say: "If I just skipped a '2', I shouldn't start a new branch with another '2' at this same level."

---

2. Solution Explanation

We use a **Backtracking (Depth-First Search)** approach. Imagine a decision tree where at each step, you decide which element to add to your current subset.

The "Symmetry Breaking" Rule

To avoid duplicates, we follow this rule: *At any level of the recursion, if the current element is the same as the previous element AND we haven't just come from that previous element, skip it.*

ASCII Step-by-Step Visualization

Input: `nums = [1, 2, 2]` (Sorted)

Level 0: Start with an empty set `[]`

```
Decision Tree:
                      [ ]  (Root)
          /            |            \
       Add 1         Add 2         Skip 2 (Duplicate check!)
        /              |              \
      [1]             [2]             (Skipped)
```

Level 1: Decisions from `[1]` From `[1]`, we can look at the remaining elements `[2, 2]`.

```
          [1]
         /   \
      Add 2   Skip 2 (Duplicate)
       /       \
    [1,2]      (Skipped)
      |
    Level 2: Add 2
      |
    [1,2,2]
```

Level 1: Decisions from `[2]` From `[2]`, we look at the remaining element `[2]`.

```
          [2]
           |
         Add 2
           |
         [2,2]
```

Complete Trace Diagram

Each node represents a call to our backtrack function. `i` is the starting index.

```
Backtrack(start_index, current_path)

Index:  0   1   2
Nums:  [1,  2,  2]
```

```
Level 0: backtrack(0, [])
|
|-- i=0: [1] -> backtrack(1, [1])
|   |-- i=1: [1,2] -> backtrack(2, [1,2])
|   |   |-- i=2: [1,2,2] -> backtrack(3, [1,2,2]) -> BASE CASE (End)
|   |
|   |-- i=2: Skip because nums[2] == nums[1] (Duplicate at this level!)
|
|-- i=1: [2] -> backtrack(2, [2])
|   |-- i=2: [2,2] -> backtrack(3, [2,2]) -> BASE CASE
|
|-- i=2: Skip because nums[2] == nums[1] (Duplicate at this level!)

Final Result: [[], [1], [1,2], [1,2,2], [2], [2,2]]
```

---

3. Time and Space Complexity Analysis

```
TIME COMPLEXITY (TC)
--------------------
1. Sorting: O(N * log(N))
2. Recursion: In the worst case (no duplicates), there are 2^N subsets.
3. Copying: For each subset, we copy it into the result list, taking O(N).

Total TC: O(N * 2^N)

Derivation:
Each element has two choices (In or Out).
Total Nodes in Tree = 2^N
Work per node = O(N) (to copy array)
Result = N * 2^N


SPACE COMPLEXITY (SC)
---------------------
1. Recursion Stack: O(N) (Depth of the tree)
2. Current Path: O(N)
3. Result Storage: O(N * 2^N) (Usually excluded in interview analysis unless specified)

Total SC: O(N) (Auxiliary space)
```

---

4. Solution Code

Top-Down (Backtracking/DFS)

This is the most intuitive "Google-style" solution because it treats the problem as a state-space search.

Python

```python
def subsetsWithDup(nums):
    res = []
    # Sort is non-trivial: it groups duplicates to allow the i > start check
    nums.sort()

    def backtrack(start, path):
        # We append a copy of the current path at every node in the tree
        res.append(list(path))

        for i in range(start, len(nums)):
            # If current element is same as previous AND it's not the
            # first element in this loop level, it's a duplicate branch.
            if i > start and nums[i] == nums[i-1]:
                continue

            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop() # Standard backtracking: clean up state

    backtrack(0, [])
    return res
```

JavaScript

```javascript
var subsetsWithDup = function(nums) {
    const res = [];
    nums.sort((a, b) => a - b);

    /**
     * @param {number} start - The index to begin searching from
     * @param {number[]} path - The current subset being built
     */
    function backtrack(start, path) {
        res.push([...path]);

        for (let i = start; i < nums.length; i++) {
            // Symmetry breaking: skip duplicates at the same recursive depth
            if (i > start && nums[i] === nums[i - 1]) continue;
```

```
            path.push(nums[i]);
            backtrack(i + 1, path);
            path.pop(); // Backtrack
        }
    }

    backtrack(0, []);
    return res;
};
```

Bottom-Up (Iterative)

The iterative approach builds subsets layer by layer. To handle duplicates, if we encounter a duplicate number, we only add it to the subsets generated in the *previous* step.

Python

```
def subsetsWithDupIterative(nums):
    nums.sort()
    res = [[]]
    last_size = 0

    for i in range(len(nums)):
        # If current is duplicate, start adding only from the subsets
        # created in the previous iteration to avoid double-counting.
        start = last_size if i > 0 and nums[i] == nums[i-1] else 0

        last_size = len(res)
        for j in range(start, last_size):
            res.append(res[j] + [nums[i]])

    return res
```

JavaScript

```
var subsetsWithDupIterative = function(nums) {
    nums.sort((a, b) => a - b);
    let res = [[]];
    let lastSize = 0;

    for (let i = 0; i < nums.length; i++) {
        // If duplicate, only extend the subsets added in the last round
        let start = (i > 0 && nums[i] === nums[i-1]) ? lastSize : 0;
```

```
        lastSize = res.length;
        for (let j = start; j < lastSize; j++) {
            res.push([...res[j], nums[i]]);
        }
    }

    return res;
};
```

Technical Terms & Concepts

- **Symmetry Breaking:** A technique used in combinatorial problems to prevent the algorithm from exploring two paths that are logically identical (like picking the first '2' vs picking the second '2').
- **Power Set:** The set of all possible subsets of a set, including the empty set and the set itself.
- **Backtracking:** An algorithmic technique that builds a solution incrementally and "backtracks" (removes the last addition) as soon as it determines that the current path cannot lead to a valid/unique solution.

Real-World / Indirect Interview Variations

Google and Meta often ask "masked" versions of this problem:

1. **Unique Combinations of Ingredients:** "Given a list of available ingredients (some duplicate), find all unique recipes you can make."
2. **Permission Groups:** "Given a list of user permissions (with some overlapping roles), generate all unique permission sets a user could have."
3. **Bloomberg - Financial Portfolios:** "Given a list of assets with potentially the same tickers/values, list all unique sub-portfolios that can be constructed."
4. **Target Sum Variations:** Problems like "Combination Sum II" use the exact same "sort and skip" logic but add a constraint that the subset must sum to a specific value.

131. Palindrome Partitioning

For an L5/L6 engineer at a place like Google, the goal isn't just to "get the code to pass." It's about writing a solution that is **modular, efficient, and readable**.

When tackling **131. Palindrome Partitioning**, a senior engineer recognizes this as a **Combinatorial Search** problem. They would immediately identify

that we are exploring a state-space tree and that we can optimize it using **Dynamic Programming (DP)** or **Memoization** to avoid redundant palindrome checks.

---

1. Problem Explanation

Given a string `s`, partition `s` such that every substring in the partition is a **palindrome**. Return all possible palindrome partitioning of `s`.

What is a Palindrome? A string that reads the same backward as forward (e.g., "aba", "racecar").

What is a Partition? A way to "cut" the string into segments. If the string is "aab", one partition is `["a", "a", "b"]`, and another is `["aa", "b"]`. We only care about partitions where **every** piece is a palindrome.

---

2. Solution Explanation

An L5+ approach uses **Backtracking** with a **Precomputed DP Table**.

The Non-Trivial Part: Avoiding Redundant Work

Checking if a substring is a palindrome takes O(N) time. If we do this inside our backtracking loop, we repeat the same checks constantly.

- **Optimization:** We precompute a 2D boolean table `isPalindrome[i][j]` where `True` means `s[i...j]` is a palindrome.

Step-by-Step Logic

1. **Precompute Palindromes:** Use DP to fill a table for all `i, j` pairs.
2. **Backtrack:** Start at index 0. Try every possible "cut" from `0` to `k`.
3. **Validate:** If `s[0...k]` is a palindrome (check our table), add it to a temporary list and recurse starting from `k + 1`.
4. **Base Case:** If we reach the end of the string, the current list is a valid partition.

ASCII Visualization: The State-Space Tree

Input: `"aab"`

```
Level 0: Start at index 0 [ "" ]
          /                  \
      "a" is pal? YES    "aa" is pal? YES
        /                      \
Level 1: Index 1 ["a"]     Index 2 ["aa"]
```

```
      /        \                |
"a" is pal?  "ab" NO      "b" is pal? YES
    |          |                |
Level 2: Index 2 ["a","a"] Level 3: Index 3 ["aa","b"] -> SUCCESS!
    |
"b" is pal? YES
    |
Level 3: Index 3 ["a","a","b"] -> SUCCESS!
```

ASCII Visualization: DP Palindrome Table

For string "aab":

```
      0 1 2  (Indices)
      a a b
    ---------
0 a | T T F |  <- Is "a" pal? T. Is "aa" pal? T. Is "aab" pal? F.
1 a | . T F |  <- Is "a" pal? T. Is "ab" pal? F.
2 b | . . T |  <- Is "b" pal? T.
```

---

3. Complexity Analysis

Time Complexity (TC)

The worst case occurs when all substrings are palindromes (e.g., "aaaaa").

```
TC = (Precomputation) + (Backtracking)
   = O(N^2) + O(N * 2^N)

Why 2^N?
In a string of length N, there are N-1 potential "cut" positions.
Each position can either be a cut or not a cut.
Total combinations = 2^(N-1).
We multiply by N because we copy the current list into the result.

Final TC: O(N * 2^N)
```

Space Complexity (SC)

```
SC = (DP Table) + (Recursion Stack) + (Result Storage)
   = O(N^2) + O(N) + O(2^N * N)

DP Table: N x N matrix to store palindrome status.
Recursion Stack: Max depth is N.
Result: Up to 2^N partitions, each with N characters.
```

```
Final SC: O(N^2)
```

---

4. Solution Code

Approach 1: Top-Down (Backtracking + Memoization)

This is the most intuitive for interviews. We "explore" the string and remember if we've seen a substring before.

Python (Top-Down)

```python
def partition(s):
    n = len(s)
    # Precompute palindromes using DP
    dp = [[False] * n for _ in range(n)]
    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                dp[i][j] = True if length <= 2 else dp[i+1][j-1]

    res = []

    # solve is a recursive function that explores all valid partitions
    # start_idx: the current character we are considering
    # current_list: the list of palindromes found so far in this path
    def solve(start_idx, current_list):
        if start_idx == n:
            res.append(list(current_list))
            return

        for end_idx in range(start_idx, n):
            if dp[start_idx][end_idx]:
                current_list.append(s[start_idx:end_idx+1])
                solve(end_idx + 1, current_list)
                current_list.pop() # Backtrack

    solve(0, [])
    return res
```

JavaScript (Top-Down)

```javascript
var partition = function(s) {
    const n = s.length;
```

```
    const dp = Array.from({ length: n }, () => Array(n).fill(false));

    // Fill DP table: O(N^2)
    for (let len = 1; len <= n; len++) {
        for (let i = 0; i <= n - len; i++) {
            let j = i + len - 1;
            if (s[i] === s[j]) {
                dp[i][j] = (len <= 2) ? true : dp[i + 1][j - 1];
            }
        }
    }

    const result = [];

    /**
     * Recursive backtracking function
     * @param {number} start - Current index in string s
     * @param {string[]} path - Accumulator for current palindromes
     */
    function backtrack(start, path) {
        if (start === n) {
            result.push([...path]);
            return;
        }

        for (let end = start; end < n; end++) {
            if (dp[start][end]) {
                path.push(s.substring(start, end + 1));
                backtrack(end + 1, path);
                path.pop(); // Revert state for next iteration
            }
        }
    }

    backtrack(0, []);
    return result;
};
```

Approach 2: Bottom-Up (Iterative DP)

This approach builds the partitions from the end of the string to the beginning, storing results in a table.

Python (Bottom-Up)

```python
def partition(s):
    n = len(s)
    # partitions_at[i] stores all valid partitions for the suffix s[i:]
    partitions_at = [[] for _ in range(n + 1)]
    partitions_at[n] = [[]]

    # Re-use palindrome logic
    is_pal = [[False] * n for _ in range(n)]
    for i in range(n - 1, -1, -1):
        for j in range(i, n):
            if s[i] == s[j] and (j - i < 2 or is_pal[i+1][j-1]):
                is_pal[i][j] = True

    # Build solutions from right to left
    for i in range(n - 1, -1, -1):
        for j in range(i, n):
            if is_pal[i][j]:
                substring = s[i:j+1]
                for p in partitions_at[j+1]:
                    partitions_at[i].append([substring] + p)

    return partitions_at[0]
```

JavaScript (Bottom-Up)

```javascript
var partition = function(s) {
    const n = s.length;
    const isPal = Array.from({ length: n }, () => Array(n).fill(false));

    // table[i] will store all valid partitions of the substring s.substring(i)
    const table = Array.from({ length: n + 1 }, () => []);
    table[n] = [[]];

    for (let i = n - 1; i >= 0; i--) {
        for (let j = i; j < n; j++) {
            if (s[i] === s[j] && (j - i < 2 || isPal[i + 1][j - 1])) {
                isPal[i][j] = true;
                let sub = s.substring(i, j + 1);
                // For every valid partition starting after this palindrome,
                // prepend this palindrome to it.
                for (let nextPart of table[j + 1]) {
                    table[i].push([sub, ...nextPart]);
                }
            }
        }
    }
```

```
    return table[0];
};
```

Terminology & Techniques

1. **Backtracking:** A refined "brute force" that builds candidates and abandons them ("backtracks") as soon as it determines they cannot lead to a valid solution. In this problem, if a substring isn't a palindrome, we don't bother recursing further down that branch.
2. **State-Space Search:** Visualizing a problem as a tree where each node is a state (current index) and edges are choices (where to cut).
3. **Precomputation:** Calculating values (like palindrome status) beforehand to speed up the main logic from O(N) to O(1) during the search.

Real-World / Interview Variants

Google, Meta, and Bloomberg often wrap this "partitioning" logic in more complex scenarios:

- **Google (Search/Maps):** "Given a string of digits, return all possible valid IP addresses." (Similar partitioning logic, but checks for 0-255 instead of palindromes).
- **Meta (Social Graph/Privacy):** "Find all ways to break a post's text into valid dictionary words (Word Break II)."
- **Bloomberg (Finance/Tickers):** "Given a stream of ticker symbols without spaces (e.g., 'AAPLMSFTGOOG'), identify all valid ways to segment them."

120. Triangle

To solve the "Triangle" problem like a high-level engineer at a top-tier firm, we don't just jump into the code. We focus on **identifying the optimal substructure**, **visualizing the state transitions**, and then **optimizing the space complexity**.

---

1. Problem Explanation

Imagine you have a triangle of numbers. You start at the very top and want to reach the bottom row. At each step, you can move to the adjacent numbers in the row below.

The goal is to find the **minimum path sum**.

The "Adjacent" Rule

If you are at index `i` in the current row, you can only move to:

1. Index `i` in the next row.
2. Index `i + 1` in the next row.

Example Triangle:

```
   [2]           <- Row 0
  [3, 4]         <- Row 1
 [6, 5, 7]       <- Row 2
[4, 1, 8, 3]     <- Row 3
```

- From **2**, you can go to **3** or **4**.
- From **3**, you can go to **6** or **5**.
- From **4**, you can go to **5** or **7**.

---

2. Solution Explanation

Why this is a Dynamic Programming (DP) problem

At any node, the "best" path depends on the "best" path of its children. This is the **Optimal Substructure**. Since multiple paths might converge on the same node in the next row, we have **Overlapping Subproblems**.

The Intuitive Way: Bottom-Up (Tabulation)

While Top-Down (starting from the peak) feels natural, **Bottom-Up** is the "engineer's choice" here. Why? Because every path must end at the bottom. If we start from the bottom row and "bubble up" the minimum costs, the answer will naturally sit at the very top (the peak).

Step-by-Step Visualization Initial State (The Input):

```
Row 0:      2
Row 1:     3 4
Row 2:    6 5 7
Row 3:   4 1 8 3  <-- We start processing from the row ABOVE this.
```

Step 1: Process Row 2 For each element in Row 2, we look at its two possible children in Row 3 and pick the smaller one.

- For **6**: min(4, 1) is 1. New value = 6 + 1 = **7**
- For **5**: min(1, 8) is 1. New value = 5 + 1 = **6**
- For **7**: min(8, 3) is 3. New value = 7 + 3 = **10**

Updated Triangle State:

```
    2
   3 4
  7 6 10  <-- Values updated
 4 1 8 3
```

Step 2: Process Row 1

- For **3**: min(7, 6) is 6. New value = 3 + 6 = **9**
- For **4**: min(6, 10) is 6. New value = 4 + 6 = **10**

Updated Triangle State:

```
    2
   9 10  <-- Values updated
  7 6 10
 4 1 8 3
```

Step 3: Process Row 0 (The Peak)

- For **2**: min(9, 10) is 9. New value = 2 + 9 = **11**

Final Answer: 11

Converting to Top-Down (Memoization)

In Top-Down, we start at the peak `(0, 0)` and ask: "What is the min path from here?" The answer is: `Triangle[row][col] + min(findPath(row+1, col), findPath(row+1, col+1))`. We use a "Memo" (a cache) to store results so we don't re-calculate the same position twice.

---

3. Complexity Analysis

Time Complexity (TC)

We visit every single element in the triangle exactly once.

```
Row 0: 1 element
Row 1: 2 elements
...
Row N: N elements

Total visits = 1 + 2 + 3 + ... + N
TC = O(N^2) where N is the number of rows.
```

Space Complexity (SC)

- **Naive DP:** O(N^2) to store a copy of the triangle.

- **Optimized Bottom-Up:** O(N). Since we only ever look at the row below us, we only need a 1D array of size N to store the "row below" results.
- **In-place:** O(1) if we are allowed to modify the input triangle.

---

4. Solution Code

Bottom-Up (Space Optimized)

This is the most efficient version. We use a single array `dp` initialized with the values of the bottom row.

Python (Bottom-Up)

```python
def minimumTotal(triangle):
    # 'dp' starts as the bottom row of the triangle
    # It will store the minimum path sum from that point down
    dp = triangle[-1][:]

    # Start from the second-to-last row and move up to the top
    for row in range(len(triangle) - 2, -1, -1):
        for col in range(len(triangle[row])):
            # The min path for current cell is:
            # its value + min of the two values directly below it
            dp[col] = triangle[row][col] + min(dp[col], dp[col + 1])

    return dp[0]
```

Javascript (Bottom-Up)

```javascript
/**
 * @param {number[][]} triangle
 * @return {number}
 */
var minimumTotal = function(triangle) {
    let n = triangle.length;
    // Initialize dp array with the bottom row
    let dp = [...triangle[n - 1]];

    // Bubble up the minimum values to the top
    for (let row = n - 2; row >= 0; row--) {
        for (let col = 0; col <= row; col++) {
            // Compare the two options below and add the smaller one
            dp[col] = triangle[row][col] + Math.min(dp[col], dp[col + 1]);
        }
    }
```

```
    return dp[0];
};
```

Top-Down (Memoization)

This approach mimics how a human thinks: "Start at the top and explore."

Python (Top-Down)

```python
def minimumTotal(triangle):
    memo = {}

    def solve(row, col):
        # Base case: if we reach the bottom, return 0
        if row == len(triangle):
            return 0

        # Check if we already calculated this position
        state = (row, col)
        if state in memo:
            return memo[state]

        # Calculate min of both paths below
        res = triangle[row][col] + min(solve(row + 1, col), solve(row + 1, col + 1))

        # Save to cache
        memo[state] = res
        return res

    return solve(0, 0)
```

Javascript (Top-Down)

```javascript
var minimumTotal = function(triangle) {
    const memo = new Map();

    /**
     * Recursive helper to find the min path from a specific cell
     * @param {number} row - current row index
     * @param {number} col - current column index
     */
    function solve(row, col) {
        if (row === triangle.length) return 0;
```

```
        const key = `${row},${col}`;
        if (memo.has(key)) return memo.get(key);

        const leftPath = solve(row + 1, col);
        const rightPath = solve(row + 1, col + 1);

        const result = triangle[row][col] + Math.min(leftPath, rightPath);

        memo.set(key, result);
        return result;
    }

    return solve(0, 0);
};
```

---

5. Important Terms

- **Dynamic Programming (DP):** An optimization technique that breaks a problem into smaller subproblems and stores their results to avoid redundant work.
- **Tabulation (Bottom-Up):** Solving the smallest subproblems first and using those to build up to the main problem. It's usually more space-efficient and avoids recursion limits.
- **Memoization (Top-Down):** Starting with the big problem and "remembering" the results of sub-calls. It's often more intuitive to write initially.

---

6. Real-World / Interview Variants

Google, Meta, and Bloomberg rarely ask the "Triangle" problem verbatim anymore. They use these variations:

1. **The "Energy Grid" (Google):** You have a grid where each cell represents energy consumption. Find the path from top to bottom with the least energy cost, but you can only move "down-left," "down," or "down-right" (Cheapest Path in a Grid).
2. **The "Dungeon Crawler" (Meta):** You have a knight in a grid with health points (positive or negative). Find the minimum initial health needed to reach the bottom-right. This adds a constraint: health can never drop to zero.
3. **The "Stock/Price Ladder" (Bloomberg):** Given a hierarchy of financial decisions (represented as a tree or triangle), find the path of maximum profit or minimum risk.

1048. Longest String Chain

This problem is a classic example of **Dynamic Programming (DP)** applied to strings. A top-tier engineer (L5/L6) looks at this and immediately sees it as a **Longest Path problem in a Directed Acyclic Graph (DAG)**.

---

1. Problem Explanation

We are given a list of `words`. A `word1` is a **predecessor** of `word2` if and only if we can insert exactly one letter anywhere in `word1` to make it equal to `word2`.

A **word chain** is a sequence of words `[word_1, word_2, ..., word_k]` where each word is a predecessor of the next. Our goal is to find the length of the **longest possible chain**.

The "Predecessor" Rule Visualized

To be a predecessor, `wordA` must be exactly 1 character shorter than `wordB`, and all characters of `wordA` must appear in `wordB` in the same relative order.

```
Example: "abc" -> "abac" (Valid: inserted 'a' at index 2)
Example: "abc" -> "abce" (Valid: inserted 'e' at end)
Example: "abc" -> "abd"  (Invalid: "d" replaces "c", not an insertion)
Example: "abc" -> "abcd" (Valid: inserted 'd')
```

---

2. Solution Explanation

An L5+ engineer will recognize that the most intuitive way to solve this is **Bottom-Up DP with a Hash Map**.

Why Bottom-Up?

1. **Sorting simplifies everything:** If we sort the words by length, we ensure that when we process a word of length 5, all possible predecessors (length 4) have already been calculated.
2. **Efficiency:** Instead of comparing every word to every other word (O(N^2)), we take a word, delete one character at a time to generate all possible "parents," and check if those parents exist in our DP table.

The Algorithm Step-by-Step

1. **Sort** the words by length.
2. Create a **Hash Map** (`dp`) where `dp[word]` stores the length of the longest chain ending at `word`.
3. For each word:

- Start its chain length at 1.
- Try removing one character at every possible position.
- If the resulting "prev" word exists in our `dp` map, update the current word's max chain: `dp[word] = max(dp[word], dp[prev] + 1)`.

ASCII Visualization: The Logic Flow

Imagine the input: `["a", "b", "ba", "bca", "bda", "bdca"]`

Step 1: Sort by length `["a", "b", "ba", "bca", "bda", "bdca"]` (Already sorted)

Step 2: Processing "a" and "b" (Length 1)

```
Word: "a"
  Subsets: "" (Not in map)
  dp = {"a": 1}

Word: "b"
  Subsets: "" (Not in map)
  dp = {"a": 1, "b": 1}
```

Step 3: Processing "ba" (Length 2)

```
Word: "ba"
  Try removing 'b' -> "a" (Found in dp! Length = dp["a"] + 1 = 2)
  Try removing 'a' -> "b" (Found in dp! Length = dp["b"] + 1 = 2)
  dp = {"a": 1, "b": 1, "ba": 2}
```

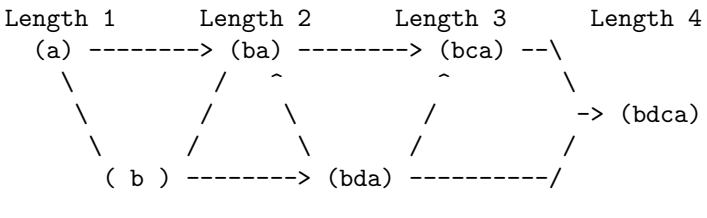
Step 4: Processing "bca" and "bda" (Length 3)

```
Word: "bca"
  Try removing 'b' -> "ca" (No)
  Try removing 'c' -> "ba" (Found! Length = dp["ba"] + 1 = 3)
  Try removing 'a' -> "bc" (No)
  dp["bca"] = 3

Word: "bda"
  Try removing 'd' -> "ba" (Found! Length = dp["ba"] + 1 = 3)
  dp["bda"] = 3
```

Step 5: Processing "bdca" (Length 4)

```
Word: "bdca"
  Try removing 'b' -> "dca" (No)
  Try removing 'd' -> "bca" (Found! Length = 3 + 1 = 4)
  Try removing 'c' -> "bda" (Found! Length = 3 + 1 = 4)
  dp["bdca"] = 4
```

The Chain Graph Visualization




```
Path 1: a -> ba -> bca -> bdca (Length 4)
Path 2: b -> ba -> bda -> bdca (Length 4)
Max Length = 4
```

---

3. Complexity Analysis

An L5 engineer must justify the complexity. Note that `N` is the number of words and `L` is the maximum length of a word.

Time Complexity (TC)

```
1. Sorting:
   O(N * log N * L)  <-- L because string comparisons take time.

2. Iterating through N words:
   For each word (N):
     Iterate through its characters (L):
       Create a substring (L):
       Hash map lookup (L):

Total TC Calculation:
O(N * log N * L + N * L * L)
=> O(N * log N * L + N * L^2)
```

Space Complexity (SC)

```
1. Hash Map (dp):
   Stores N words as keys.
   Each word has length L.

Total SC Calculation:
O(N * L)
```

---

4. Solution Code

Approach 1: Bottom-Up (Iterative) - The "Industry Standard"

This is usually preferred in production for avoiding recursion depth limits.

Python (Bottom-Up)

```python
def longestStrChain(words):
    # Sort by length so predecessors are always processed first
    words.sort(key=len)
    dp = {}
    max_chain = 0

    for word in words:
        # Every word is a chain of at least length 1
        dp[word] = 1
        # Try removing each character to find a potential predecessor
        for i in range(len(word)):
            prev = word[:i] + word[i+1:]
            if prev in dp:
                dp[word] = max(dp[word], dp[prev] + 1)

        max_chain = max(max_chain, dp[word])

    return max_chain
```

Javascript (Bottom-Up)

```javascript
/**
 * @param {string[]} words
 * @return {number}
 */
var longestStrChain = function(words) {
    words.sort((a, b) => a.length - b.length);
    const dp = new Map();
    let maxChain = 0;

    for (const word of words) {
        let currentMax = 1;
        // Generate all possible predecessors by deleting one char
        for (let i = 0; i < word.length; i++) {
            const prev = word.slice(0, i) + word.slice(i + 1);
            if (dp.has(prev)) {
                currentMax = Math.max(currentMax, dp.get(prev) + 1);
            }
        }
```

```
        dp.set(word, currentMax);
        maxChain = Math.max(maxChain, currentMax);
    }
    return maxChain;
};
```

Approach 2: Top-Down (Recursive with Memoization)

This approach feels more like a "DFS" on a graph. We check a word, and recursively ask: "What is the longest chain ending at my predecessor?"

Python (Top-Down)

```
def longestStrChain(words):
    word_set = set(words)
    memo = {}

    # This function returns the max chain length ending at 'word'
    def dfs(word):
        if word not in word_set:
            return 0
        if word in memo:
            return memo[word]

        res = 1
        for i in range(len(word)):
            prev = word[:i] + word[i+1:]
            # Recurse: chain length is 1 + max chain of predecessor
            res = max(res, 1 + dfs(prev))

        memo[word] = res
        return res

    return max(dfs(w) for w in words)
```

Javascript (Top-Down)

```
var longestStrChain = function(words) {
    const wordSet = new Set(words);
    const memo = new Map();

    /**
     * Recursively finds the longest chain for a given word.
     * Uses memoization to avoid redundant sub-problem calculations.
     */
```

```
    function dfs(word) {
        if (!wordSet.has(word)) return 0;
        if (memo.has(word)) return memo.get(word);

        let maxLen = 1;
        for (let i = 0; i < word.length; i++) {
            const prev = word.slice(0, i) + word.slice(i + 1);
            maxLen = Math.max(maxLen, 1 + dfs(prev));
        }

        memo.set(word, maxLen);
        return maxLen;
    }

    let globalMax = 0;
    for (const word of words) {
        globalMax = Math.max(globalMax, dfs(word));
    }
    return globalMax;
};
```

Note 1: Key Terms & Techniques

- **DAG (Directed Acyclic Graph):** The words form a graph where edges only go from shorter words to longer words. Since you can't go from a long word back to a shorter one by adding characters, there are no cycles.
- **State Compression:** Instead of a full graph adjacency list, we use the "deletion" trick to find neighbors on the fly. This saves huge amounts of memory.
- **Sorting as Pre-processing:** In DP, sorting is often used to establish a "topological order," ensuring that sub-problems are solved before the problems that depend on them.

Note 2: Real-World Interview Variations

Companies like **Google** and **Bloomberg** rarely ask this verbatim anymore. They use variations:

1. **The Library Dependency Problem:** You have a list of software libraries. Library A is a "base" for B if B has all of A's features plus one. Find the longest dependency chain.
2. **The Evolution Tree:** Given DNA sequences, find the longest evolutionary path where each step adds one mutation (nucleotide).
3. **Bloomberg (Financial Data):** Given a series of ticker symbols that grow in length (e.g., A, AB, ABC), find the longest valid sequence of

symbol “upgrades” based on specific character rules.

354. Russian Doll Envelopes

This problem is a classic at top-tier companies because it tests your ability to take a 2D problem and “collapse” it into a 1D problem using a very specific sorting trick.

An L5/L6 engineer would immediately recognize this as a variation of the **Longest Increasing Subsequence (LIS)** problem.

---

1. Problem Explanation

You are given a 2D array of integers `envelopes` where `envelopes[i] = [width, height]`.

One envelope can fit into another if and only if **both** the width and height of one envelope are strictly greater than the width and height of the other envelope. You need to find the maximum number of envelopes you can “Russian doll” (nest inside one another).

The Strict Constraint: If Envelope A is `[w1, h1]` and Envelope B is `[w2, h2]`: To put A inside B: `w1 < w2` AND `h1 < h2`.

Example: Input: `[[5,4],[6,4],[6,7],[2,3]]` Output: `3` Explanation: `[2,3] => [5,4] => [6,7]`

---

2. Solution Explanation

The “Aha!” Moment: Sorting

If we sort the envelopes by **width**, we only need to worry about the **heights**. However, there is a catch: if two envelopes have the same width, how do we handle their heights?

The Trick:

1. Sort width in **ascending** order.
2. If widths are equal, sort height in **descending** order.

Why descending height for same width? Because we need **strictly** greater dimensions. If we have `[3, 4]` and `[3, 5]`, we cannot put one inside the other. By sorting heights descending (`[3, 5], [3, 4]`), when we look for an increasing subsequence of heights, we will never pick both since 4 is not greater than 5.

Step-by-Step Visualization

Input: `[[5,4],[6,4],[6,7],[2,3]]`

Step 1: Sort

- Sort by Width (Asc), then Height (Desc).
- Result: `[[2,3], [5,4], [6,7], [6,4]]`

Step 2: Extract Heights

- We only care about the heights now: `[3, 4, 7, 4]`
- Now find the **Longest Increasing Subsequence (LIS)** of these heights.

Step 3: Find LIS (The Patience Sorting Approach) We maintain a list `tails`. For each height `h`, we use binary search to find the smallest element in `tails` that is greater than or equal to `h`. If it exists, replace it. If not, append `h`.

ASCII Trace of LIS on `[3, 4, 7, 4]`:

```
Processing 3:
tails = [3]
Diagram: [3]

Processing 4:
4 > 3, so append.
tails = [3, 4]
Diagram: [3] -> [4]

Processing 7:
7 > 4, so append.
tails = [3, 4, 7]
Diagram: [3] -> [4] -> [7]

Processing 4:
Search for first element >= 4. It's 4 itself.
Replace 4 with 4 (no change).
tails = [3, 4, 7]

Result: Length is 3.
```

Why Top-Down is less intuitive here

Usually, we start with Top-Down (Recursion + Memoization). `solve(index, prev_index)`:

- If `height[index] > height[prev_index]`: we can choose to take it or leave it.
- Otherwise: we must leave it.

This is `O(N^2)`. At L5 level, you know `O(N^2)` will TLE (Time Limit Exceeded) for `N = 100,000`. Therefore, the **Binary Search + Greedy** approach (Patience Sorting) is the "senior" way to solve this.

---

3. Time and Space Complexity Analysis

The complexity is dominated by the Sorting and the Binary Search LIS.

TIME COMPLEXITY:

Step	Operation	Complexity
Sorting	N * log(N)	O(N log N)
LIS (N elements)	N iterations	
Binary Search	log(N) per iteration	O(N log N)
TOTAL	O(N log N + N log N)	O(N log N)

SPACE COMPLEXITY:

Component	Usage	Complexity
Tails Array	Stores the LIS sequence	O(N)
Recursion Stack	(Only for Top-Down)	O(N)
TOTAL		O(N)

---

4. Solution Code

Python Snippet

```python
class Solution:
    # BOTTOM-UP (Optimized O(N log N) with Binary Search)
    # This is the industry standard for this problem.
    def maxEnvelopes(self, envelopes: list[list[int]]) -> int:
        if not envelopes: return 0

        # Sort width ascending, height descending for same width
        envelopes.sort(key=lambda x: (x[0], -x[1]))

        # We only need the heights now
        heights = [e[1] for e in envelopes]
```

```
    # tails[i] stores the smallest tail of all increasing
    # subsequences of length i+1
    tails = []

    for h in heights:
        # Binary search to find the insertion point
        import bisect
        idx = bisect.bisect_left(tails, h)

        if idx < len(tails):
            tails[idx] = h
        else:
            tails.append(h)

    return len(tails)

# TOP-DOWN (O(N^2) - Educational Only, will TLE on large inputs)
# Use memoization to store results of subproblems
def maxEnvelopesTopDown(self, envelopes: list[list[int]]) -> int:
    envelopes.sort()
    memo = {}

    def solve(idx, prev_idx):
        state = (idx, prev_idx)
        if idx == len(envelopes):
            return 0
        if state in memo:
            return memo[state]

        # Option 1: Skip current envelope
        res = solve(idx + 1, prev_idx)

        # Option 2: Take current if it fits
        curr_w, curr_h = envelopes[idx]
        if prev_idx == -1:
            res = max(res, 1 + solve(idx + 1, idx))
        else:
            prev_w, prev_h = envelopes[prev_idx]
            if curr_w > prev_w and curr_h > prev_h:
                res = max(res, 1 + solve(idx + 1, idx))

        memo[state] = res
        return res

    return solve(0, -1)
```

JavaScript Snippet

```javascript
/**
 * BOTTOM-UP O(N log N)
 * Uses the Patience Sorting algorithm to find LIS in log time.
 */
var maxEnvelopes = function(envelopes) {
    if (!envelopes.length) return 0;

    // Sort: Width ASC, Height DESC
    envelopes.sort((a, b) => {
        if (a[0] !== b[0]) return a[0] - b[0];
        return b[1] - a[1];
    });

    const tails = [];

    for (let i = 0; i < envelopes.length; i++) {
        let h = envelopes[i][1];

        // Binary search for insertion point in tails
        let left = 0, right = tails.length;
        while (left < right) {
            let mid = Math.floor((left + right) / 2);
            if (tails[mid] < h) left = mid + 1;
            else right = mid;
        }

        if (left === tails.length) {
            tails.push(h);
        } else {
            tails[left] = h;
        }
    }

    return tails.length;
};

/**
 * TOP-DOWN O(N^2)
 * Demonstrates the recursive relation but is slow for N > 5000.
 */
var maxEnvelopesTopDown = function(envelopes) {
    envelopes.sort((a, b) => a[0] - b[0] || a[1] - b[1]);
    const n = envelopes.length;
    const memo = new Map();
```

```
    function dp(idx, prevIdx) {
        if (idx === n) return 0;
        const key = `${idx},${prevIdx}`;
        if (memo.has(key)) return memo.get(key);

        // Choice 1: Don't include
        let res = dp(idx + 1, prevIdx);

        // Choice 2: Include (if valid)
        if (prevIdx === -1 ||
           (envelopes[idx][0] > envelopes[prevIdx][0] &&
            envelopes[idx][1] > envelopes[prevIdx][1])) {
            res = Math.max(res, 1 + dp(idx + 1, idx));
        }

        memo.set(key, res);
        return res;
    }

    return dp(0, -1);
};
```

Technical Terms

- **LIS (Longest Increasing Subsequence):** The longest sequence of elements in an array that are strictly increasing.
- **Patience Sorting:** An algorithm inspired by a card game that finds the LIS in `O(N log N)` time. It uses "piles" (the `tails` array) to keep track of potential sequences.
- **Lexicographical Sort:** Sorting based on multiple criteria (first by width, then by height).

Real World / Interview Variations

1. **Bloomberg (Supply Chain):** "You have a list of shipping containers with dimensions. Find the max number of containers you can stack inside one another."
2. **Google (Task Scheduling):** "Given tasks with start and end times, but tasks also have a 'priority' and 'complexity' score. Find the longest sequence of tasks where both score A and score B increase."
3. **Meta (Ad Targeting):** "Find the maximum chain of user segments where each segment is a strict subset of the next (based on population

size and interest density)."
4. **Tower of Babylon:** A 3D version where you can rotate blocks. (Requires trying all 3 rotations as separate envelopes).

44. Wildcard Matching

An L5 or L6 engineer at Google doesn't just look for a "pass" on Leetcode; they look for the most robust, scalable, and readable solution while identifying the "gotchas" that break naive implementations.

For **Wildcard Matching**, the core challenge is the `*` character, which introduces a "branching" decision: should it match nothing, one character, or many?

---

1. Problem Explanation

We are given two strings: `s` (the text) and `p` (the pattern). We need to determine if the pattern matches the entire text.

- `?`: Matches any **single** character.
- `*`: Matches any **sequence** of characters (including an empty sequence).

The "Non-Trivial" Part: The difficulty lies in the `*`. If you see `a*b` and your text is `aaab`, the `*` could represent `aa`. But how do you know when to stop the `*` and start matching the `b`? This is a classic state-space search problem.

---

2. Solution Explanation

The most intuitive way to approach this is **Top-Down with Memoization**. Why? Because it mirrors how a human thinks: "I'll try to match this character, and if it's a star, I'll try all possibilities."

The Logic (Recursive Thinking)

1. If both strings are empty, it's a match.
2. If the pattern is empty but text isn't, it's a fail.
3. If characters match or pattern is `?`, move both pointers forward.
4. If pattern is `*`:

- **Choice A:** Treat `*` as empty (skip it).
- **Choice B:** Treat `*` as matching the current character of `s` (stay on `*` but move `s` pointer).

ASCII Visualization: The Decision Tree

Text: `cb`, Pattern: `a*`

```
Step 1: match('cb', 'a*')
        'c' != 'a' -> FAIL!

Text: `ab`, Pattern: `a*`

Step 1: match('ab', 'a*')
        'a' == 'a' -> Move both.

Step 2: match('b', '*')
        Found a '*'. Two branches:
        /                          \
    Choice A (Empty)           Choice B (Match 'b')
    match('', '')              match('', '*')
    [SUCCESS]                  [Found '*' again...]
                               /              \
                          Match empty      Match char (none left)
                          [SUCCESS]        [FAIL]
```

Bottom-Up DP Table Visualization

Let `s = "adceb"` and `p = "*a*b"`. We create a 2D grid where `dp[i][j]` is true if `s[0...i-1]` matches `p[0...j-1]`.

```
Pattern (j) ->  ""    * a     * b
Text (i)       [0]   [1]   [2]   [3]   [4]
-------------------------------------------
""      [0] |  T  |  T  |  F  |  F  |  F  |
"a"     [1] |  F  |  T  |  T  |  T  |  F  |
"d"     [2] |  F  |  T  |  F  |  T  |  F  |
"c"     [3] |  F  |  T  |  F  |  T  |  F  |
"e"     [4] |  F  |  T  |  F  |  T  |  F  |
"b"     [5] |  F  |  T  |  F  |  T  |  T  |  <-- Result: Match!
```

Key Transitions:

1. If `p[j-1] == s[i-1]` or `p[j-1] == '?'`: `dp[i][j] = dp[i-1][j-1]` (Diagonal move)
2. If `p[j-1] == '*'`: `dp[i][j] = dp[i-1][j]` (Star matches char) OR `dp[i][j-1]` (Star is empty).

---

3. Complexity Analysis

Time Complexity (TC)

We visit every cell in the 2D grid of size `(N+1) * (M+1)`.

```
TC = Number of States * Work per State
Number of States = (Length of S) * (Length of P)
Work per State = O(1) (simple boolean checks)

Total TC = O(N * M)
```

Space Complexity (SC)

We store the results in a 2D table or a memoization map.

```
SC = Table Size
Size = (N + 1) * (M + 1)

Total SC = O(N * M)
```

Note: This can be optimized to O(M) for Bottom-Up by only keeping the previous row.

---

4. Solution Code

Python (Top-Down & Bottom-Up)

```python
class Solution:
    # --- Top-Down Approach (Memoization) ---
    def isMatch(self, s: str, p: str) -> bool:
        memo = {}

        # Recursive function to check matching from indices i and j
        def solve(i, j):
            if (i, j) in memo: return memo[(i, j)]

            # Base Case: Pattern exhausted
            if j == len(p):
                return i == len(s)

            # Base Case: Text exhausted
            if i == len(s):
                # Pattern must only contain '*' to match empty text
                return p[j] == '*' and solve(i, j + 1)

            match = False
            # Current characters match or pattern has '?'
            if p[j] == s[i] or p[j] == '?':
                match = solve(i + 1, j + 1)
            # Pattern has '*'
            elif p[j] == '*':
```

```
                # solve(i+1, j): '*' matches current char, stay on '*'
                # solve(i, j+1): '*' matches nothing, move pattern
                match = solve(i + 1, j) or solve(i, j + 1)

            memo[(i, j)] = match
            return match

        return solve(0, 0)

    # --- Bottom-Up Approach (Tabulation) ---
    def isMatchTab(self, s: str, p: str) -> bool:
        n, m = len(s), len(p)
        dp = [[False] * (m + 1) for _ in range(n + 1)]

        # Base Case: Empty string matches empty pattern
        dp[0][0] = True

        # Handle patterns like "***" matching empty string
        for j in range(1, m + 1):
            if p[j-1] == '*':
                dp[0][j] = dp[0][j-1]

        for i in range(1, n + 1):
            for j in range(1, m + 1):
                if p[j-1] == s[i-1] or p[j-1] == '?':
                    dp[i][j] = dp[i-1][j-1]
                elif p[j-1] == '*':
                    dp[i][j] = dp[i-1][j] or dp[i][j-1]

        return dp[n][m]
```

JavaScript (Top-Down & Bottom-Up)

```
/**
 * Top-Down with Memoization
 */
var isMatch = function(s, p) {
    const memo = new Map();

    function solve(i, j) {
        const key = `${i},${j}`;
        if (memo.has(key)) return memo.get(key);

        // Pattern exhausted
        if (j === p.length) return i === s.length;
```

```
        // Text exhausted
        if (i === s.length) {
            const res = p[j] === '*' && solve(i, j + 1);
            memo.set(key, res);
            return res;
        }

        let result = false;
        if (p[j] === s[i] || p[j] === '?') {
            result = solve(i + 1, j + 1);
        } else if (p[j] === '*') {
            // Choice 1: Use '*' to match char. Choice 2: Use '*' as empty.
            result = solve(i + 1, j) || solve(i, j + 1);
        }

        memo.set(key, result);
        return result;
    }
    return solve(0, 0);
};

/**
 * Bottom-Up Tabulation
 */
var isMatchTab = function(s, p) {
    const n = s.length;
    const m = p.length;
    const dp = Array.from({ length: n + 1 }, () => Array(m + 1).fill(false));

    dp[0][0] = true;

    // Fill first row for leading '*' in pattern
    for (let j = 1; j <= m; j++) {
        if (p[j - 1] === '*') dp[0][j] = dp[0][j - 1];
    }

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (p[j - 1] === s[i - 1] || p[j - 1] === '?') {
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] === '*') {
                dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            }
        }
    }
    return dp[n][m];
```

```
};
```

Terminology: Dynamic Programming (DP)

What is it? It's a method for solving complex problems by breaking them down into simpler subproblems. **Why it helps here:** Without DP, this problem is exponential (O(2^N)). You would re-calculate the same `(i, j)` match status thousands of times. DP "remembers" the answer to a specific sub-match so you only compute it once.

Real-World / Indirect Interview Variants

Companies like Google and Meta rarely ask the literal "Leetcode 44" anymore. They use variants:

1. **File System Search (Google):** "Implement a function that finds all files matching a pattern like `*.txt` or `log_202?_*` in a directory tree."
2. **Log Filtering (Bloomberg):** "Write a high-performance log parser that filters entries based on user-defined wildcards. Optimize for the case where the pattern has many stars."
3. **Router/URL Matching (Meta):** "Given a URL and a list of route patterns (e.g., `/user/*/settings`), identify which route matches. Note: stars here might only match between slashes."

10. Regular Expression Matching

An L5 or L6 engineer at Google doesn't just jump into coding this; they treat it as a **recursive decision tree** problem. This is a classic "Hard" problem because of the `*` wildcard, which introduces a "choice" at every step.

---

1. Problem Explanation

We are matching a string `s` against a pattern `p`.

- `.` matches any single character.
- `*` matches **zero or more** of the preceding element.

The "Tricky" Part: The `*` is not a standalone wildcard. It modifies the character before it. For example, `a*` can be:

- Empty string (zero `a`'s)
- `a`
- `aa`
- `aaa`… and so on.

2. Solution Explanation

The most intuitive way to solve this is **Top-Down Dynamic Programming (Recursion + Memoization)**. Why? Because at any point where we see a `*`, we have to make a choice:

1. **Ignore the * expression:** Treat `a*` as an empty string.
2. **Use the * expression:** If the current character matches, keep the `*` and move forward in the string `s`.

The Visual Logic Flow

Imagine we are matching `s = "aaab"`, `p = "a*b"`.

```
Step 1: Compare s[0] ('a') with p[0..1] ('a*')
        Does 'a' match 'a'? Yes.
        CHOICE A: Skip 'a*' (match zero). Does "aaab" match "b"? NO.
        CHOICE B: Use 'a*' (match one). Does "aab" match "a*b"? (Recurse)

Step 2: Compare s[1] ('a') with p[0..1] ('a*')
        CHOICE B: Use 'a*' again. Does "ab" match "a*b"? (Recurse)

Step 3: Compare s[2] ('a') with p[0..1] ('a*')
        CHOICE B: Use 'a*' again. Does "b" match "a*b"? (Recurse)

Step 4: Compare s[3] ('b') with p[0..1] ('a*')
        'b' does NOT match 'a'.
        MUST CHOICE A: Skip 'a*'. Does "b" match "b"? YES.
```

The Recursive Tree (ASCII Visualization)

Let `match(i, j)` be the function checking `s[i:]` and `p[j:]`.

```
                  match("aa", "a*")
                  /               \
          (Skip *)   /                 \ (Use *)
          match("aa", "")           match("a", "a*")
              | (False)             /              \
                        (Skip *)   /                \ (Use *)
                               match("a", "")      match("", "a*")
                                   | (False)       /             \
                                       (Skip *)   /               \ (Use *)
                                            match("", "")        match(error)
                                                | (True!)
```

Transition to Bottom-Up (Tabulation)

While Top-Down is intuitive, Bottom-Up is often preferred in production for avoiding stack overflow. We build a 2D table `dp[i][j]` representing whether `s[i:]` matches `p[j:]`.

---

3. Time and Space Complexity

Let `N` be the length of string `s` and `M` be the length of pattern `p`.

```
TIME COMPLEXITY DERIVATION:
Each state is defined by the pair (i, j).
There are N + 1 possible values for i.
There are M + 1 possible values for j.
Total unique states = (N + 1) * (M + 1).
Work per state = Constant (O(1)) because we only check 1 or 2 next states.
Total Time = O(N * M)

SPACE COMPLEXITY DERIVATION:
We store the result of each state in a 2D table or memo hash.
Table size = (N + 1) * (M + 1).
Total Space = O(N * M)
```

---

4. Solution Code

Top-Down Approach (Recursion + Memoization)

This is usually the "Interview Favorite" because it follows the logical thought process directly.

Python

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        memo = {}

        # i: index in s, j: index in p
        def dp(i, j):
            if (i, j) in memo:
                return memo[(i, j)]

            # Base case: if pattern is exhausted, s must be exhausted too
            if j == len(p):
                return i == len(s)
```

```
            # Check if current characters match
            first_match = i < len(s) and (p[j] == s[i] or p[j] == '.')

            # Handle '*' logic
            if j + 1 < len(p) and p[j+1] == '*':
                # Choice 1: Skip the '*' element (match zero)
                # Choice 2: Use the '*' element (if first_match is true)
                res = dp(i, j + 2) or (first_match and dp(i + 1, j))
            else:
                # Regular single character match
                res = first_match and dp(i + 1, j + 1)

            memo[(i, j)] = res
            return res

        return dp(0, 0)
```

JavaScript

```
var isMatch = function(s, p) {
    const memo = new Map();

    /**
     * Recursive helper with memoization
     * i: pointer for string s
     * j: pointer for pattern p
     */
    function dp(i, j) {
        const key = `${i},${j}`;
        if (memo.has(key)) return memo.get(key);

        if (j === p.length) return i === s.length;

        const firstMatch = i < s.length && (p[j] === s[i] || p[j] === '.');

        let res;
        if (j + 1 < p.length && p[j + 1] === '*') {
            // Either we skip the x* (dp(i, j+2))
            // Or we consume s[i] if it matches and stay at j (dp(i+1, j))
            res = dp(i, j + 2) || (firstMatch && dp(i + 1, j));
        } else {
            res = firstMatch && dp(i + 1, j + 1);
        }

        memo.set(key, res);
        return res;
    }
```

```
    return dp(0, 0);
};
```

Bottom-Up Approach (Tabulation)

Python

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        # dp[i][j] will be true if s[i:] matches p[j:]
        dp = [[False] * (len(p) + 1) for _ in range(len(s) + 1)]
        dp[len(s)][len(p)] = True

        for i in range(len(s), -1, -1):
            for j in range(len(p) - 1, -1, -1):
                first_match = i < len(s) and (p[j] == s[i] or p[j] == '.')

                if j + 1 < len(p) and p[j+1] == '*':
                    # Skip or Use
                    dp[i][j] = dp[i][j+2] or (first_match and dp[i+1][j])
                else:
                    dp[i][j] = first_match and dp[i+1][j+1]

        return dp[0][0]
```

JavaScript

```javascript
var isMatch = function(s, p) {
    const dp = Array.from({ length: s.length + 1 }, () => Array(p.length + 1).fill(false));

    // Base case: Empty string matches empty pattern
    dp[s.length][p.length] = true;

    for (let i = s.length; i >= 0; i--) {
        for (let j = p.length - 1; j >= 0; j--) {
            const firstMatch = i < s.length && (p[j] === s[i] || p[j] === '.');

            if (j + 1 < p.length && p[j + 1] === '*') {
                // dp[i][j+2] handles matching zero characters
                // (firstMatch && dp[i+1][j]) handles matching 1 or more characters
                dp[i][j] = dp[i][j + 2] || (firstMatch && dp[i + 1][j]);
            } else {
                dp[i][j] = firstMatch && dp[i + 1][j + 1];
            }
        }
```

```
    }

    return dp[0][0];
};
```

Terminology & Real-World Use

Memoization

- **What:** Storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- **Why:** Without it, this problem is exponential (2^{N+M}). With it, it's linear relative to the state space ($N * M$).

Dynamic Programming (DP)

- **What:** Breaking down a problem into simpler sub-problems and solving them just once.
- **Why:** In Regex, matching the whole string depends on whether the substrings matched.

Real-World Interview Variants

1. **Google:** "Design a simplified glob-matching engine for a file system search (supporting `*` and `?`)."
2. **Meta:** "Implement a basic version of `grep` that only supports literal characters and `.` and `*`."
3. **Bloomberg:** "Given a financial ticker pattern (e.g., `TSL*`), validate if a stream of incoming tickers matches the subscription pattern."

132. Palindrome Partitioning II

A top-tier L5/L6 engineer doesn't just jump into coding; they focus on **efficiency**, **state reduction**, and **bottleneck identification**. For this specific problem, they recognize that the "naive" approach is exponential, and the standard DP is O(n^3), but the "Senior" way to solve it is O(n^2) by pre-calculating palindromes or expanding from centers.

---

1. Problem Explanation

The goal is to take a string `s` and cut it into several pieces such that **every single piece** is a palindrome. We want to find the **minimum** number of cuts needed to achieve this.

Example: `s = "aab"`

- Option 1: `["a", "a", "b"]` -> 2 cuts
- Option 2: `["aa", "b"]` -> 1 cut (Both "aa" and "b" are palindromes)
- **Result:** 1

The Logic: If the entire string is already a palindrome, cuts = 0. If not, we try cutting it at every possible position and see which branch leads to the fewest total cuts.

---

2. Solution Explanation

To solve this efficiently, we use **Dynamic Programming**.

The Core Bottleneck

A common mistake is checking `isPalindrome(sub)` inside the DP loop. That adds an $O(n)$ check inside an $O(n^2)$ DP, leading to $O(n^3)$. An L5 engineer will **pre-process** all palindromic substrings in $O(n^2)$ first.

Step 1: Pre-computing Palindromes (The Boolean Matrix)

We create a 2D table `isPal[i][j]` which is true if `s[i...j]` is a palindrome.

```
String: "ababa"
Index:  01234

ASCII Matrix (isPal[i][j]):
      a  b  a  b  a
      0  1  2  3  4
0 a [ T, F, T, F, T ]  <- "a", "aba", "ababa" are palindromes
1 b [ ., T, F, T, F ]  <- "b", "bab" are palindromes
2 a [ ., ., T, F, T ]  <- "a", "aba" are palindromes
3 b [ ., ., ., T, F ]
4 a [ ., ., ., ., T ]
```

Step 2: The DP Strategy (Bottom-Up)

We define `dp[i]` as the minimum cuts needed for the prefix `s[0...i]`.

Logic for `dp[i]`:

1. If `s[0...i]` is a palindrome, `dp[i] = 0`.
2. Otherwise, we try all possible "last cuts" at index `j`. If `s[j...i]` is a palindrome, then `dp[i]` could be `dp[j-1] + 1`. We take the minimum of all such `j`.

Visualizing the DP transitions for "aab":

```
Index: 0 1 2
Char:  a a b

Initial DP: [0, 0, 0] (representing min cuts for s[0..i])

i = 0 ("a"):
  Is "a" a palindrome? Yes.
  dp[0] = 0

i = 1 ("aa"):
  Is "aa" a palindrome? Yes.
  dp[1] = 0

i = 2 ("aab"):
  Is "aab" a palindrome? No.
  Try cut after index 0: s[0..0]="a", s[1..2]="ab". "ab" is NOT pal.
  Try cut after index 1: s[0..1]="aa", s[2..2]="b".
     "b" is a palindrome!
     dp[2] = dp[1] + 1 = 0 + 1 = 1.

Final DP array: [0, 0, 1] -> Result is 1.
```

---

3. Time and Space Complexity Analysis

An L5 engineer would explain the complexity by breaking down the two distinct phases: Pre-processing and Main DP.

```
PHASE 1: Palindrome Table Generation
- Nested loops (i and j) to fill the matrix.
- Work per cell: O(1)
- Total: O(n^2)

PHASE 2: Minimum Cut DP
- Outer loop runs n times (i from 0 to n-1).
- Inner loop runs up to n times (j from 0 to i).
- Work per iteration: O(1) (lookup in our pre-processed table).
- Total: O(n^2)

OVERALL TIME COMPLEXITY:
  O(n^2) + O(n^2) = O(n^2)

OVERALL SPACE COMPLEXITY:
  O(n^2) for the boolean matrix + O(n) for the DP array.
  Total: O(n^2)
```

---

4. Solution Code

Top-Down Approach (Memoization)

This approach is often more intuitive during an interview as it follows a "natural" recursive breakdown.

Python (Top-Down)

```python
def minCut(s):
    n = len(s)
    # Pre-compute palindromes to avoid O(N) check inside recursion
    is_pal = [[False] * n for _ in range(n)]
    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                is_pal[i][j] = True if length <= 2 else is_pal[i+1][j-1]

    memo = {}

    # solve(i) returns the min cuts for the suffix s[i:]
    def solve(i):
        if i == n or is_pal[i][n-1]:
            return 0
        if i in memo:
            return memo[i]

        res = float('inf')
        # Try cutting at every j such that s[i:j+1] is a palindrome
        for j in range(i, n):
            if is_pal[i][j]:
                # If we cut here, we add 1 cut + whatever is needed for the rest
                res = min(res, 1 + solve(j + 1))

        memo[i] = res
        return res

    return solve(0)
```

JavaScript (Top-Down)

```javascript
/**
 * @param {string} s
 * @return {number}
```

```javascript
 */
var minCut = function(s) {
    const n = s.length;
    const isPal = Array.from({ length: n }, () => Array(n).fill(false));

    // Pre-calculate all palindrome substrings
    for (let len = 1; len <= n; len++) {
        for (let i = 0; i <= n - len; i++) {
            let j = i + len - 1;
            if (s[i] === s[j]) {
                isPal[i][j] = (len <= 2) ? true : isPal[i + 1][j - 1];
            }
        }
    }

    const memo = new Map();

    // Recursive function to find min cuts for substring starting at 'start'
    function solve(start) {
        if (start >= n || isPal[start][n - 1]) return 0;
        if (memo.has(start)) return memo.get(start);

        let minCuts = n; // Max possible cuts is n-1
        for (let end = start; end < n; end++) {
            if (isPal[start][end]) {
                // If s[start...end] is a palindrome, cut after 'end'
                minCuts = Math.min(minCuts, 1 + solve(end + 1));
            }
        }
        memo.set(start, minCuts);
        return minCuts;
    }

    return solve(0);
};
```

Bottom-Up Approach (Tabulation)

This is the "gold standard" for production-level DP because it avoids recursion depth limits.

Python (Bottom-Up)

```python
def minCut(s):
    n = len(s)
```

```python
    if n <= 1: return 0

    # is_pal[i][j] is true if s[i...j] is a palindrome
    is_pal = [[False] * n for _ in range(n)]
    # dp[i] = min cuts for prefix s[0...i]
    dp = [0] * n

    for i in range(n):
        min_cuts = i # maximum cuts needed for s[0...i] is i (e.g., "abc" -> a|b|c)
        for j in range(i + 1):
            # Check if s[j...i] is a palindrome
            if s[i] == s[j] and (i - j <= 2 or is_pal[j + 1][i - 1]):
                is_pal[j][i] = True
                # If s[j...i] is a palindrome and j is 0, no cuts needed
                if j == 0:
                    min_cuts = 0
                else:
                    # Otherwise, min cuts is 1 + cuts for prefix before j
                    min_cuts = min(min_cuts, dp[j - 1] + 1)
        dp[i] = min_cuts

    return dp[n - 1]
```

JavaScript (Bottom-Up)

```javascript
var minCut = function(s) {
    const n = s.length;
    if (n <= 1) return 0;

    const isPal = Array.from({ length: n }, () => Array(n).fill(false));
    const dp = new Array(n).fill(0);

    for (let i = 0; i < n; i++) {
        let minCuts = i;
        for (let j = 0; j <= i; j++) {
            // Check if characters match and the inner part is a palindrome
            if (s[i] === s[j] && (i - j <= 2 || isPal[j + 1][i - 1])) {
                isPal[j][i] = true;

                // If the whole prefix s[0...i] is a palindrome, 0 cuts
                if (j === 0) {
                    minCuts = 0;
                } else {
                    // Current min is the best of (previous best + 1 new cut)
                    minCuts = Math.min(minCuts, dp[j - 1] + 1);
                }
```

```
            }
        }
        dp[i] = minCuts;
    }

    return dp[n - 1];
};
```

Terminology & Techniques

- **Pre-computation:** The act of calculating the `isPal` matrix beforehand. This is crucial because it turns a repetitive $O(n)$ check into a $O(1)$ lookup.
- **State Transition:** In DP, this is the rule `dp[i] = min(dp[j-1] + 1)`. It effectively says: "The best way to cut this string is to find the best way to cut a smaller piece and add one more cut."
- **Expansion from Center:** An L6 might mention that the `isPal` matrix can be filled by picking every character as a "center" and expanding outwards. This is an alternative to the $O(n^2)$ nested loop used above.

Real-World Interview Variants

Companies like Google or Meta rarely ask the Leetcode version verbatim. They use variants:

1. **Storage Optimization (Bloomberg):** "How can you solve this using only $O(n)$ space?" (Hint: Manacher's Algorithm + DP).
2. **Streaming Data (Google):** "If characters are being added to the string one by one, how do you update the minimum cuts in real-time?"
3. **Grouping (Meta):** "Divide a list of users into the minimum number of groups where each group meets a specific 'balance' criteria (similar to being a palindrome)."
4. **Justification (General):** "Print the actual partitions, not just the count." (This requires path reconstruction using the DP table).

877. Stone Game

An L5 or L6 engineer at Google doesn't just look for *a* solution; they look for the *optimal* solution while identifying the underlying mathematical patterns that others might miss.

For **Stone Game**, there is a "clever" observation that solves it in one line, but an interviewer will expect you to derive the Dynamic Programming (DP) approach first to prove you can handle complex state transitions.

---

1. Problem Explanation

You are given an array `piles` where `piles[i]` is the number of stones in the `i-th` pile. Two players, Alice and Bob, take turns picking a pile from either the **beginning** or the **end** of the row. Alice goes first.

The goal is to determine if Alice can win, assuming both play optimally.

The Constraints & Rules:

- The total number of piles is **even**.
- The total number of stones is **odd** (meaning there are no ties).
- Alice wants to maximize (Alice_score - Bob_score).
- Bob wants to minimize (Alice_score - Bob_score) (or maximize his own).

---

2. Solution Explanation

The Core Intuition: Minimax

In zero-sum games (or games where one's gain is another's loss), we use the **Minimax** principle. Since Alice wants to maximize her lead, and Bob wants to reduce it, we can think of Alice adding to the score and Bob subtracting from it.

Step 1: Top-Down DP (Recursive with Memoization)

We define a function `solve(left, right)` which returns the maximum relative score Alice can get from the subarray `piles[left...right]`.

Visualizing the Choices: If the piles are `[5, 3, 4, 5]`: Alice can pick the first `5` or the last `5`.

```
Initial State: [5, 3, 4, 5]
               ^        ^
            Left      Right

Choice A: Pick Left (5)
New State: 5 - solve(3, 4, 5)  <-- We subtract the next result because
                                   it's now the other player's turn to maximize.

Choice B: Pick Right (5)
New State: 5 - solve(5, 3, 4)
```

ASCII Decision Tree (Step-by-Step):

```
Level 0: Alice's Turn [5, 3, 4, 5]
         /                      \
   Pick Left(5)              Pick Right(5)
       /                          \
```

```
Level 1: Bob [3, 4, 5]          Level 1: Bob [5, 3, 4]
    /           \                   /           \
Pick(3)       Pick(5)           Pick(5)       Pick(4)
  /               \               /               \
Level 2: Alice [4, 5]         Level 2: Alice [3, 4]
...and so on.
```

Step 2: Bottom-Up DP (Iterative)

We use a 2D table `dp[i][j]` where `i` is the start index and `j` is the end index. `dp[i][j]` represents the max relative score for the range `[i...j]`.

The Transition: `dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])`

Building the Table for `[5, 3, 4, 5]`:

1. **Base Case:** Ranges of length 1 (Diagonal).

```
     0    1    2    3
0   [5]   .    .    .
1    .   [3]   .    .
2    .    .   [4]   .
3    .    .    .   [5]
```

2. **Length 2:** `[5,3]`, `[3,4]`, `[4,5]` `dp[0][1] = max(5-3, 3-5) = 2` `dp[1][2] = max(3-4, 4-3) = 1` `dp[2][3] = max(4-5, 5-4) = 1`

```
     0    1    2    3
0    5    2    .    .
1    .    3    1    .
2    .    .    4    1
3    .    .    .    5
```

3. **Length 4 (Final Answer):** `dp[0][3] = max(5 - dp[1][3], 5 - dp[0][2])` … and so on until `dp[0][3]` is filled.

---

3. Complexity Analysis

Time Complexity (TC)

We iterate through all possible subarrays. For an array of size N, the number of subarrays is roughly (N * N) / 2.

```
ASCII TC Derivation:
Each cell in the N x N matrix is calculated once.
Calculation per cell = O(1) (simple max/subtraction)
Total Operations = N * (N + 1) / 2
Simplified TC = O(N^2)
```

Space Complexity (SC)

We store the results in a 2D table or a recursion stack.

```
ASCII SC Derivation:
Matrix Size = N rows * N columns
Memory = O(N * N)
Note: Can be optimized to O(N) by only keeping the previous row.
Simplified SC = O(N^2)
```

---

4. Solution Code

Python

```python
class Solution:
    # TOP-DOWN APPROACH (Memoization)
    def stoneGame(self, piles: list[int]) -> bool:
        memo = {}

        def solve(i, j):
            # Base case: only one pile left
            if i == j:
                return piles[i]
            if (i, j) in memo:
                return memo[(i, j)]

            # The current player picks i or j,
            # and we subtract the result of the next turn
            pick_left = piles[i] - solve(i + 1, j)
            pick_right = piles[j] - solve(i, j - 1)

            memo[(i, j)] = max(pick_left, pick_right)
            return memo[(i, j)]

        return solve(0, len(piles) - 1) > 0

    # BOTTOM-UP APPROACH (Tabulation)
    def stoneGameBottomUp(self, piles: list[int]) -> bool:
        n = len(piles)
        # dp[i][j] stores the max relative score for piles[i...j]
        dp = [[0] * n for _ in range(n)]

        # Base case: single piles
        for i in range(n):
            dp[i][i] = piles[i]
```

```
        # Build table for lengths 2 to n
        for length in range(2, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])

        return dp[0][n-1] > 0
```

Javascript

```
/**
 * TOP-DOWN APPROACH
 */
var stoneGame = function(piles) {
    const memo = new Map();

    function solve(i, j) {
        if (i === j) return piles[i];
        const key = `${i}-${j}`;
        if (memo.has(key)) return memo.get(key);

        // Player takes piles[i] or piles[j] and subtracts opponent's best score
        const res = Math.max(piles[i] - solve(i + 1, j), piles[j] - solve(i, j - 1));

        memo.set(key, res);
        return res;
    }

    return solve(0, piles.length - 1) > 0;
};

/**
 * BOTTOM-UP APPROACH
 */
var stoneGameBottomUp = function(piles) {
    const n = piles.length;
    const dp = Array.from({ length: n }, () => Array(n).fill(0));

    // Base Case: diagonal (single piles)
    for (let i = 0; i < n; i++) {
        dp[i][i] = piles[i];
    }

    // Fill the table based on subproblem size (length)
    for (let len = 2; len <= n; len++) {
        for (let i = 0; i <= n - len; i++) {
```

```
            let j = i + len - 1;
            dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
        }
    }

    return dp[0][n - 1] > 0;
};
```

Technical Deep Dive: The Mathematical "Cheat"

An L6 engineer would point out that **Alice always wins.**

Why? The piles have an even length. Alice can choose to take *only* the even-indexed piles or *only* the odd-indexed piles.

1. She calculates the sum of `piles[0, 2, 4...]` and `piles[1, 3, 5...]`.
2. Since the total sum is odd, one of these sums **must** be larger than the other.
3. She can force the game to ensure she gets whichever set is larger.

Conclusion: The answer is always `return True`. However, understanding the DP is what proves your ability to solve variations where the number of piles is not even or Alice isn't allowed to pick from both ends.

Real-World / Interview Variations

1. Predict the Winner (Meta/Google): Identical to this but usually with more constraints or different scoring (e.g., you can pick 1, 2, or 3 items).

2. Resource Allocation (Bloomberg): Two competing systems trying to claim tasks from a shared queue where only the ends are accessible. How do you maximize your system's "priority points"?

3. Optimal Account Balancing (Google/Finance): Determining the best sequence of transactions to settle debts, often using similar "state-based" DP where you pick from ends of a sorted list.

Terminology:

- **Minimax:** A decision rule used in game theory for minimizing the possible loss for a worst-case scenario.
- **Zero-Sum Game:** A situation in which one participant's gain or loss is exactly balanced by the losses or gains of the other participants.

1140. Stone Game II

An L5 or L6 engineer at Google doesn't just look for "the answer." They look for the **underlying structure** of the problem. Stone Game II is a classic example of a **Minimax** problem disguised as a Game Theory challenge, solvable via Dynamic Programming (DP).

---

1. Problem Explanation

Imagine two players, Alice and Bob, playing a game with a row of stone piles.

- **The Goal:** Maximize the total number of stones you collect.
- **The Rules:**

1. Alice starts first.
2. There is a variable `M`. Initially, `M = 1`.
3. A player can take `X` piles, where `1 <= X <= 2 * M`.
4. After taking `X` piles, the new `M` for the next player becomes `max(M, X)`.
5. The game ends when no piles are left.

The Non-Trivial Constraint

The "Optimal Play" part is the trickiest. Since both play optimally, Alice isn't just trying to pick the biggest pile now; she's trying to pick a number of piles that leaves Bob in the worst possible position for the rest of the game.

---

2. Solution Explanation

For an L5/L6, the most intuitive way to model this is **Top-Down with Memoization**. Why? Because it maps directly to the decision tree of the game.

The Core Logic: "Total Stones - Best Opponent Score"

Instead of tracking two scores, we track how many stones the **current** player can get from index `i` onwards. If Alice takes some stones, Bob will then try to maximize his own stones from the remaining piles. Alice's best score is: `Total stones remaining - Bob's best possible score`.

Step-by-Step Visual Tree (ASCII)

Let's say `piles = [2, 7, 9, 4, 4]` and `M = 1`.

Step 1: Alice's Turn (i=0, M=1) Alice can take `X = 1` or `X = 2` piles (since 2 * 1 = 2).

```
Decision Level 1 (Alice):
i=0, M=1
|
|-- Choice 1: Take X=1 (pile [2])
|   New state: i=1, M=max(1, 1) = 1
|   Bob's turn starts at index 1 with M=1
|
|-- Choice 2: Take X=2 (piles [2, 7])
    New state: i=2, M=max(1, 2) = 2
    Bob's turn starts at index 2 with M=2
```

Step 2: Bob's Turn (If Alice took X=1) Now Bob is at `i=1, M=1`. He can take `X=1` or `X=2`.

```
Decision Level 2 (Bob):
i=1, M=1
|
|-- Choice 1: Take X=1 (pile [7])
|   New state: i=2, M=1
|   Alice's turn starts at index 2 with M=1
|
|-- Choice 2: Take X=2 (piles [7, 9])
    New state: i=3, M=2
    Alice's turn starts at index 3 with M=2
```

Transition to DP Table

We need to store the results of `(index, M)` so we don't re-calculate them.

```
Suffix Sum Array (Crucial for quick math):
piles:      [ 2,  7,  9,  4,  4 ]
SuffixSum:  [26, 24, 17,  8,  4 ]
(SuffixSum[i] = total stones from index i to the end)

DP Memoization Table (Conceptual):
Rows: index i (0 to N)
Cols: M (1 to N)
+-------+-----+-----+-----+-----+
| i \ M |  1  |  2  |  3  | ... |
+-------+-----+-----+-----+-----+
|   4   |  4  |  4  |  4  |     | <- Base case: only last pile left
|   3   |  8  |  8  |  8  |     |
|   2   | ... | ... | ... |     |
+-------+-----+-----+-----+-----+
```

---

3. Complexity Analysis

Time Complexity (TC)

We explore states defined by `(i, M)`.

- `i` goes from `0` to `N`.
- `M` can technically grow, but it effectively caps at `N`.
- Inside each state, we loop `X` from `1` to `2*M`.

```
TC Derivation:
Total States = N (for i) * N (for M) = N^2
Work per State = 2 * M (which is proportional to N)
Total Complexity = N^2 * N = O(N^3)
```

Space Complexity (SC)

```
SC Derivation:
Memoization Table = N * N states
Recursion Depth = N
Total Complexity = O(N^2)
```

---

4. Solution Code

Top-Down (Recursive + Memoization)

This is usually preferred in interviews for its readability.

Python

```python
def stoneGameII(piles):
    n = len(piles)
    # suffix_sums[i] stores the sum of piles from i to n-1
    suffix_sums = [0] * (n + 1)
    for i in range(n - 1, -1, -1):
        suffix_sums[i] = suffix_sums[i + 1] + piles[i]

    memo = {}

    # solve(i, m) returns the max stones the current player can get
    # starting from index i with parameter m
    def solve(i, m):
        if i >= n: return 0
        # If we can take all remaining piles, do it!
        if i + 2 * m >= n:
            return suffix_sums[i]
```

```
        state = (i, m)
        if state in memo: return memo[state]

        res = 0
        # Try taking X piles from 1 to 2*m
        for x in range(1, 2 * m + 1):
            # Current player's score = Total remaining - Opponent's best score
            opponent_best = solve(i + x, max(m, x))
            res = max(res, suffix_sums[i] - opponent_best)

        memo[state] = res
        return res

    return solve(0, 1)
```

JavaScript

```
var stoneGameII = function(piles) {
    const n = piles.length;
    const suffixSums = new Array(n + 1).fill(0);
    for (let i = n - 1; i >= 0; i--) {
        suffixSums[i] = suffixSums[i + 1] + piles[i];
    }

    const memo = new Map();

    /**
     * @param {number} i - Current starting index
     * @param {number} m - Current M value
     * @returns {number} - Max stones current player can get
     */
    function solve(i, m) {
        if (i >= n) return 0;
        if (i + 2 * m >= n) return suffixSums[i];

        const key = `${i}-${m}`;
        if (memo.has(key)) return memo.get(key);

        let minOpponentScore = Infinity;
        // Optimization: Find the move that leaves the opponent with the LEAST stones
        for (let x = 1; x <= 2 * m; x++) {
            minOpponentScore = Math.min(minOpponentScore, solve(i + x, Math.max(m, x)));
        }

        const res = suffixSums[i] - minOpponentScore;
        memo.set(key, res);
```

```
        return res;
    }

    return solve(0, 1);
};
```

Bottom-Up (Tabulation)

This is the "L6 level" approach, showing deep understanding of state dependencies.

Python

```python
def stoneGameII_bottomUp(piles):
    n = len(piles)
    suffix_sums = [0] * (n + 1)
    for i in range(n - 1, -1, -1):
        suffix_sums[i] = suffix_sums[i + 1] + piles[i]

    # dp[i][m] = max stones current player can get from index i with parameter m
    dp = [[0] * (n + 1) for _ in range(n + 1)]

    for i in range(n - 1, -1, -1):
        for m in range(1, n + 1):
            # If we can take the rest of the piles
            if i + 2 * m >= n:
                dp[i][m] = suffix_sums[i]
            else:
                # Try all possible X and find the one that minimizes opponent's score
                for x in range(1, 2 * m + 1):
                    dp[i][m] = max(dp[i][m], suffix_sums[i] - dp[i + x][max(m, x)])

    return dp[0][1]
```

JavaScript

```javascript
var stoneGameII_bottomUp = function(piles) {
    const n = piles.length;
    const suffixSums = new Array(n + 1).fill(0);
    for (let i = n - 1; i >= 0; i--) {
        suffixSums[i] = suffixSums[i + 1] + piles[i];
    }

    // Initialize DP table with 0s
    const dp = Array.from({ length: n + 1 }, () => new Array(n + 1).fill(0));
```

```
    // Fill the table backwards from the last pile
    for (let i = n - 1; i >= 0; i--) {
        for (let m = 1; m <= n; m++) {
            if (i + 2 * m >= n) {
                dp[i][m] = suffixSums[i];
            } else {
                for (let x = 1; x <= 2 * m; x++) {
                    // Logic: I get everything minus what my opponent gets optimally
                    dp[i][m] = Math.max(dp[i][m], suffixSums[i] - dp[i + x][Math.max(m, x)]);
                }
            }
        }
    }

    return dp[0][1];
};
```

Terminology: Minimax & Suffix Sums

- **Minimax Principle:** In zero-sum games (or games where one's gain is another's loss), you play to maximize your score assuming your opponent is playing to minimize your score.
- **Suffix Sum:** A pre-calculated array where `suffix[i]` is the sum of all elements from `i` to the end. It allows calculating "remaining stones" in `O(1)` time.
- **Memoization:** Storing the results of expensive function calls to avoid redundant "re-solving" of the same game state.

Real World / Indirect Interview Questions

Companies rarely ask "Stone Game II" directly anymore. They use these variations:

1. **Google (Resource Allocation):** "Two teams are competing for server clusters in a row. Each team can take X clusters, which increases the 'cooldown' (M) for the next team..."
2. **Meta (Ad Bidding):** "Two advertisers bid on a sequence of ad slots. Taking more slots now increases the minimum bid for the next round. Maximize total ad value."
3. **Bloomberg (Trading Strategy):** "A sequence of bond payouts. You and a competitor take turns claiming bonds. However, claiming a large chunk of liquid bonds impacts the market volatility (M), affecting how

many the next person can claim."

1575. Count All Possible Routes

An L5 or L6 engineer at Google doesn't just look for a "working" solution; they look for **pattern recognition**, **constraint analysis**, and **clean state management**.

For this specific problem, a senior engineer would immediately identify this as a **Directed Acyclic Graph (DAG) in a 3D-like state space** (Position + Fuel). Since we need to count *all* paths and can revisit cities, this is a classic Dynamic Programming (DP) problem.

---

1. Problem Explanation

You are given:

- `locations`: An array of coordinates on a 1D line.
- `start`: The index you begin at.
- `finish`: The index you want to reach.
- `fuel`: The total fuel you have.

The Rules:

1. Moving from city `i` to city `j` costs `abs(locations[i] - locations[j])` fuel.
2. You can visit any city multiple times (including `start` and `finish`).
3. Every time you land on the `finish` city, it counts as **one valid route**, even if you have fuel left to keep moving.
4. You stop when you run out of fuel and cannot move to another city.

The Goal: Return the total number of ways to reach `finish` from `start` given the fuel. Since the number can be huge, return it **modulo 10^9 + 7**.

---

2. Solution Explanation

The Intuition: Top-Down (Recursive with Memoization)

A senior engineer starts here because it mimics how we naturally think.

State: What defines our "current situation"?

1. Where am I? (`currCity`)
2. How much fuel do I have? (`remainingFuel`)

The Recursive Leap: If I am at `currCity` with `remainingFuel`, the number of ways to reach `finish` is:

- **(1)** if `currCity == finish`, plus…
- **(The sum of)** all ways to reach `finish` from every *other* city I can afford to fly to.

Step-by-Step Visualization

Let's say: `locations = [2, 3, 6, 8]`, `start = 0`, `finish = 3`, `fuel = 5`.

State Tree Visualization

```
Fuel: 5 | Position: City 0 (Loc: 2)
|
|-- Move to City 1 (Loc: 3). Cost = |2-3| = 1. Remaining Fuel: 4
|   |-- Is City 1 the finish? No.
|   |-- Move to City 3 (Loc: 8). Cost = |3-8| = 5. (NOT ENOUGH FUEL, X)
|   |-- Move to City 2 (Loc: 6). Cost = |3-6| = 3. Remaining Fuel: 1
|       |-- Move to City 3... (X)
|
|-- Move to City 2 (Loc: 6). Cost = |2-6| = 4. Remaining Fuel: 1
|   |-- Is City 2 the finish? No.
|   |-- Move to City 1 (Loc: 3). Cost = |6-3| = 3. (NOT ENOUGH FUEL, X)
|
|-- Move to City 3 (Loc: 8). Cost = |2-8| = 6. (NOT ENOUGH FUEL, X)
```

Why we need Memoization (The "Aha!" moment) Without memoization, we recalculate the same state. If we reach **City 2 with 2 fuel** via two different paths, the "number of ways to get to finish from here" is identical. We store this in a `memo[city][fuel]` table.

Converting to Bottom-Up (Tabular)

To be L6-level, you must show how to flip this into an iterative approach. We fill a 2D table where `dp[f][i]` is the number of ways to be at `city i` with `f` fuel remaining.

ASCII DP Table Logic (Fuel vs City):

```
Fuel (f) ->  0    1    2    3    4    5
City (i)
-----------------------------------------
City 0   |  0    0    0    0    0    1  (Start here with 5 fuel)
City 1   |  0    0    0    0    0    0
City 2   |  0    0    0    0    0    0
City 3   |  0    0    0    0    0    0

Processing Fuel from Max to 0:
For each fuel 'f', if dp[f][currCity] > 0:
```

```
Try moving to 'nextCity'.
Cost = abs(loc[currCity] - loc[nextCity])
NewFuel = f - Cost
dp[NewFuel][nextCity] += dp[f][currCity]
```

---

3. Complexity Analysis

Time Complexity (TC)

We have two main variables: `N` (number of cities) and `F` (total fuel).

```
+-------------------------------------------------------------+
| Logic:                                                      |
| 1. We have N * F possible states (City, Fuel).              |
| 2. For each state, we iterate through N possible next cities.|
|                                                             |
| Calculation:                                                |
| Total States = N * F                                        |
| Work per State = N                                          |
| Total TC = N * F * N = (N^2) * F                            |
+-------------------------------------------------------------+
```

Space Complexity (SC)

```
+-------------------------------------------------------------+
| Logic:                                                      |
| 1. We store the results in a 2D array of size N by F.       |
| 2. In Top-Down, we also have the recursion stack.           |
|                                                             |
| Calculation:                                                |
| Memo/DP Table = N * F                                       |
| Recursion Stack (Top-Down) = F (max depth)                 |
| Total SC = O(N * F)                                         |
+-------------------------------------------------------------+
```

---

4. Solution Code

Top-Down (Memoization)

Python

```python
class Solution:
    def countRoutes(self, locations: list[int], start: int, finish: int, fuel: int) -> int:
        MOD = 10**9 + 7
        n = len(locations)
```

```python
        # memo[city][remaining_fuel]
        memo = {}

        # solve(curr, f) calculates ways to reach finish from curr with f fuel
        def solve(curr, f):
            if f < 0: return 0
            if (curr, f) in memo: return memo[(curr, f)]

            # If we are at finish, this is 1 valid route.
            # But we don't stop! We can keep moving if we have fuel.
            res = 1 if curr == finish else 0

            for next_city in range(n):
                if next_city != curr:
                    cost = abs(locations[curr] - locations[next_city])
                    if f >= cost:
                        res = (res + solve(next_city, f - cost)) % MOD

            memo[(curr, f)] = res
            return res

        return solve(start, fuel)
```

JavaScript

```javascript
var countRoutes = function(locations, start, finish, fuel) {
    const MOD = 1000000007;
    const n = locations.length;
    const memo = Array.from({ length: n }, () => Array(fuel + 1).fill(-1));

    /**
     * Recursively calculates possible routes.
     * @param {number} curr - Current city index
     * @param {number} f - Remaining fuel
     */
    function solve(curr, f) {
        if (memo[curr][f] !== -1) return memo[curr][f];

        // Current spot is finish? That's 1 way.
        let res = (curr === finish) ? 1 : 0;

        for (let next = 0; next < n; next++) {
            if (next !== curr) {
                let cost = Math.abs(locations[curr] - locations[next]);
                if (f >= cost) {
                    res = (res + solve(next, f - cost)) % MOD;
```

```
                }
            }
        }

        return memo[curr][f] = res;
    }

    return solve(start, fuel);
};
```

Bottom-Up (Tabulation)

Python

```python
class Solution:
    def countRoutes(self, locations: list[int], start: int, finish: int, fuel: int) -> int:
        MOD = 10**9 + 7
        n = len(locations)
        # dp[f][i] = ways to be at city i with f fuel remaining
        dp = [[0] * n for _ in range(fuel + 1)]

        # Base case: We start at 'start' with total 'fuel'
        dp[fuel][start] = 1

        total_ways = 0
        # We iterate fuel from max down to 0 because moving always consumes fuel
        for f in range(fuel, -1, -1):
            for curr in range(n):
                if dp[f][curr] == 0: continue

                # If we are at the finish, add these ways to total
                if curr == finish:
                    total_ways = (total_ways + dp[f][curr]) % MOD

                # Try moving to every other city
                for next_city in range(n):
                    if curr == next_city: continue
                    cost = abs(locations[curr] - locations[next_city])
                    if f >= cost:
                        dp[f - cost][next_city] = (dp[f - cost][next_city] + dp[f][curr]) %

        return total_ways
```

JavaScript

```javascript
var countRoutes = function(locations, start, finish, fuel) {
    const MOD = 1000000007;
    const n = locations.length;
    const dp = Array.from({ length: fuel + 1 }, () => Array(n).fill(0));

    // Initial state: 1 way to be at start with full fuel
    dp[fuel][start] = 1;
    let totalWays = 0;

    for (let f = fuel; f >= 0; f--) {
        for (let curr = 0; curr < n; curr++) {
            if (dp[f][curr] === 0) continue;

            if (curr === finish) {
                totalWays = (totalWays + dp[f][curr]) % MOD;
            }

            for (let next = 0; next < n; next++) {
                if (curr === next) continue;
                let cost = Math.abs(locations[curr] - locations[next]);
                if (f >= cost) {
                    dp[f - cost][next] = (dp[f - cost][next] + dp[f][curr]) % MOD;
                }
            }
        }
    }

    return totalWays;
};
```

Terminology Checklist

- **Modulo Arithmetic:** We use `(a + b) % MOD` to prevent integer overflow. In interviews, forgetting this is a red flag.
- **State Space:** The combination of `(City, Fuel)` defines where we are. Since fuel only decreases, we don't have infinite loops even though we can visit the same city twice.
- **Overlapping Subproblems:** The reason DP works here. Reaching City A with 10 fuel always has the same number of future paths, regardless of how we got there.

Real-World Interview Variations

1. **Google (Google Maps Team):** "Find the number of ways to travel between two points given a limited battery range for an EV, where some

points have charging stations." (This adds a state change at specific nodes).

2. **Meta (Ads/Feed):** "Given a budget of $X, how many ways can we distribute ads across N placements where each placement has a specific cost and can be reused?"
3. **Bloomberg (Ticker systems):** "Counting paths in a graph where nodes can be revisited but each edge traversal consumes a 'time' resource."