Array, String and Hash table

1. Gas Station

Pattern: Greedy

Problem Statement

There are `n` gas stations along a circular route, where the amount of gas at the `i`th station is `gas[i]`.
You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from the `i`th station to its next `(i + 1)`th station. You begin the journey with an empty tank at one of the gas stations.
Given two integer arrays `gas` and `cost`, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1*. If there exists a solution, it is **guaranteed to be unique**.

Sample Input & Output

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation: Start at station 3 (index 3).
Fill 4 units → go to 4 (cost 1, left 3).
Add 5 → total 8 → go to 0 (cost 2, left 6).
Add 1 → 7 → go to 1 (cost 3, left 4).
Add 2 → 6 → go to 2 (cost 4, left 2).
Add 3 → 5 → go to 3 (cost 5, left 0). Done.
```

```
Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation: Total gas = 9, total cost = 10 → impossible.
```

```
Input: gas = [5], cost = [4]
Output: 0
Explanation: Single station: 5 gas, 4 cost → completes loop.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def canCompleteCircuit(self, gas: List[int],
                           cost: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - total_tank tracks net gas for entire circuit
        #   - curr_tank tracks gas from current start candidate
        #   - start is our candidate starting index
        total_tank = 0
        curr_tank = 0
        start = 0

        # STEP 2: Main loop / recursion
        #   - Traverse all stations once
        #   - If curr_tank drops below 0, reset start to next
        #     station and reset curr_tank
        for i in range(len(gas)):
            total_tank += gas[i] - cost[i]
            curr_tank += gas[i] - cost[i]

            # STEP 3: Update state / bookkeeping
            #   - If we can't reach next station from 'start',
            #     no point trying any station between 'start'
            #     and 'i' - skip to i+1 as new candidate
            if curr_tank < 0:
                start = i + 1
```

```
                curr_tank = 0

        # STEP 4: Return result
        #   - If total gas >= total cost, solution exists
        #   - Else, impossible → return -1
        return start if total_tank >= 0 else -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.canCompleteCircuit(
        [1,2,3,4,5], [3,4,5,1,2]
    ) == 3, "Normal case failed"

    #  Test 2: Edge case - impossible
    assert sol.canCompleteCircuit(
        [2,3,4], [3,4,3]
    ) == -1, "Impossible case failed"

    #  Test 3: Tricky/negative - single station
    assert sol.canCompleteCircuit(
        [5], [4]
    ) == 0, "Single station failed"

    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`gas = [1,2,3,4,5]`, `cost = [3,4,5,1,2]`

Initial state:
- `total_tank = 0`
- `curr_tank = 0`
- `start = 0`

Step 1: i = 0
- Compute net: `gas[0] - cost[0] = 1 - 3 = -2`
- Update:
- `total_tank = 0 + (-2) = -2`
- `curr_tank = 0 + (-2) = -2`
- Since `curr_tank < 0`:
- Set `start = 0 + 1 = 1`
- Reset `curr_tank = 0`

State: `total=-2`, `curr=0`, `start=1`

Step 2: i = 1
- Net: `2 - 4 = -2`
- Update:
- `total_tank = -2 + (-2) = -4`
- `curr_tank = 0 + (-2) = -2`
- `curr_tank < 0` →
- `start = 1 + 1 = 2`
- `curr_tank = 0`

State: `total=-4`, `curr=0`, `start=2`

Step 3: i = 2
- Net: `3 - 5 = -2`
- Update:
- `total = -4 + (-2) = -6`
- `curr = 0 + (-2) = -2`
- `curr < 0` →
- `start = 3`
- `curr = 0`

State: `total=-6`, `curr=0`, `start=3`

Step 4: i = 3
- Net: `4 - 1 = 3`
- Update:
- `total = -6 + 3 = -3`
- `curr = 0 + 3 = 3`
- `curr >= 0` → no change to `start`

State: `total=-3`, `curr=3`, `start=3`

Step 5: i = 4
- Net: `5 - 2 = 3`
- Update:
- `total = -3 + 3 = 0`
- `curr = 3 + 3 = 6`
- Still non-negative → keep `start=3`

Final state: `total=0`, `curr=6`, `start=3`

After loop:
- Check `total_tank >= 0` → `0 >= 0` → **True**
- Return `start = 3`

Output: `3` — matches expected!

Key Insight:
The greedy choice works because if you can't reach station `j` from `i`, then **no station between `i` and `j` can be a valid start** — they'd have even less gas. So we skip directly to `j+1`.

Complexity Analysis

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

  Single pass through the `gas` and `cost` arrays. Each station visited exactly once.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`total_tank`, `curr_tank`, `start`) used — constant extra space.

2. Largest Number

Pattern: Greedy

Problem Statement

Given a list of non-negative integers `nums`, arrange them such that they form the largest possible number and return it as a string.
Since the result may be very large, return it as a string instead of an integer.
Note: The result should not have leading zeros unless the result is exactly `"0"`.

Sample Input & Output

```
Input: [10, 2]
Output: "210"
Explanation: "2" + "10" = "210" > "102"
```

```
Input: [3, 30, 34, 5, 9]
Output: "9534330"
Explanation: Among all permutations, "9534330" is the largest.
```

```
Input: [0, 0]
Output: "0"
Explanation: All numbers are zero → return single "0", not "00".
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from functools import cmp_to_key

class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        # STEP 1: Convert all numbers to strings for custom comparison
        #   - We need to decide order based on concatenation result,
        #     not numeric value (e.g., "3" vs "30" → "330" > "303")
        str_nums = [str(num) for num in nums]

        # STEP 2: Define custom comparator
        #   - For two strings a and b, if a+b > b+a, a should come first
        #   - This greedy choice ensures locally optimal ordering
        def compare(a: str, b: str) -> int:
            if a + b > b + a:
                return -1  # a comes before b
            elif a + b < b + a:
                return 1   # b comes before a
            else:
                return 0   # equal

        # STEP 3: Sort using custom comparator
        #   - Python's sort is stable; we use cmp_to_key to adapt old-style
        #     comparator to key function
        str_nums.sort(key=cmp_to_key(compare))

        # STEP 4: Join and handle edge case of all zeros
        #   - If largest number is "0", entire result is "0"
        #   - Prevents output like "000"
        result = ''.join(str_nums)
        return "0" if result[0] == "0" else result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.largestNumber([10, 2]) == "210"

    #  Test 2: Edge case - all zeros
    assert sol.largestNumber([0, 0]) == "0"
```

```
#  Test 3: Tricky/negative - mix with leading zeros in numbers
assert sol.largestNumber([3, 30, 34, 5, 9]) == "9534330"

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through `[3, 30, 34, 5, 9]` step by step.

Step 1: Convert to strings
- Input: `nums = [3, 30, 34, 5, 9]`
- After `str_nums = [str(num) for num in nums]`:
→ `str_nums = ["3", "30", "34", "5", "9"]`

Step 2: Define `compare(a, b)`
This function decides order by checking which concatenation is bigger:
- `compare("3", "30")` → compare `"330"` vs `"303"` → `"330" > "303"` → return `-1` → `"3"` comes before `"30"`
- `compare("5", "9")` → `"59"` vs `"95"` → `"59" < "95"` → return `1` → `"9"` comes before `"5"`

Step 3: Sort using custom comparator
Python's sort uses our `compare` logic via `cmp_to_key`.
It repeatedly asks: *"Should A come before B?"* using our rule.

Sorting proceeds (conceptually): - Compare all pairs to find best order. - Final sorted order: `["9", "5", "34", "3", "30"]`
Why? - `"9"` beats everyone (`"95" > "59"`, `"934" > "349"`, etc.) - `"5"` next - `"34"` vs `"3"` → `"343" > "334"` → so `"34"` before `"3"` - `"3"` vs `"30"` → `"330" > "303"` → `"3"` before `"30"`

So: `str_nums` becomes `["9", "5", "34", "3", "30"]`

Step 4: Join and handle zeros
- `result = ''.join(str_nums)` → `"9534330"`
- Check first char: `result[0] == "9"  "0"` → return `"9534330"`

Output: `"9534330"`

Edge Case: `[0, 0]`
- `str_nums = ["0", "0"]`
- Sorting: `compare("0","0")` → `"00" == "00"` → order unchanged
- `result = "00"`
- But `result[0] == "0"` → return `"0"` (not `"00"`)

This prevents invalid output like `"00"` or `"000"`.

Complexity Analysis

- **Time Complexity**: `O(n log n * k)`

  Sorting takes `O(n log n)` comparisons. Each comparison concatenates two strings of length up to `k` (max digits in a number), so each comparison is `O(k)`.
  Total: `O(n log n * k)`.
  In practice, `k` `10` (for 32-bit ints), so often treated as `O(n log n)`.

- **Space Complexity**: `O(n * k)`

  We store `n` strings, each up to `k` characters long. Sorting may use `O(n)` extra space (Timsort). Total: `O(n * k)`.

3. Combination Sum

Pattern: Backtracking

Problem Statement

Given an array of **distinct** integers `candidates` and a target integer `target`, return a list of all **unique combinations** of `candidates` where the chosen numbers sum to `target`.
You may return the combinations in **any order**.
The **same number** may be chosen from `candidates` an **unlimited number of times**.
Two combinations are unique if the frequency of at least one of the chosen numbers is different.

Sample Input & Output

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation: 2+2+3 = 7 and 7 = 7. No other combinations work.
```

```
Input: candidates = [2], target = 1
Output: []
Explanation: Cannot reach odd target with only even number.
```

```
Input: candidates = [1], target = 1
Output: [[1]]
Explanation: Single-element match.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def combinationSum(
        self, candidates: List[int], target: int
    ) -> List[List[int]]:
```

```python
        # STEP 1: Initialize structures
        #   - result: collects valid combinations
        #   - current: tracks current path (mutable state)
        result = []
        current = []

        # STEP 2: Main loop / recursion
        #   - Use DFS with backtracking
        #   - Start index avoids duplicate permutations
        #   - Prune when current sum exceeds target
        def dfs(start: int, remaining: int):
            if remaining == 0:
                # Found valid combination
                result.append(current.copy())
                return
            if remaining < 0:
                # Prune invalid path
                return

            for i in range(start, len(candidates)):
                num = candidates[i]
                # STEP 3: Update state / bookkeeping
                current.append(num)
                # Recurse with same index (reuse allowed)
                dfs(i, remaining - num)
                # Backtrack: remove last choice
                current.pop()

        dfs(0, target)
        # STEP 4: Return result
        #   - All paths explored; result may be empty
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.combinationSum([2,3,6,7], 7) == [[2,2,3],[7]]

    #  Test 2: Edge case - no solution
    assert sol.combinationSum([2], 1) == []
```

```python
#  Test 3: Tricky/negative - single element match
assert sol.combinationSum([1], 1) == [[1]]

print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `combinationSum([2,3,6,7], 7)` step by step.

Initial State: - `result = []` - `current = []` - Call `dfs(0, 7)`

Step 1: `dfs(0, 7)`
- `remaining = 7 > 0`, so enter loop `i = 0 to 3` - `i = 0`: `num = 2` - `current.append(2)` → `current = [2]` - Call `dfs(0, 5)` (7 - 2)

Step 2: `dfs(0, 5)`
- `i = 0`: `num = 2` - `current = [2,2]` - Call `dfs(0, 3)`

Step 3: `dfs(0, 3)`
- `i = 0`: `num = 2` - `current = [2,2,2]` - Call `dfs(0, 1)`

Step 4: `dfs(0, 1)`
- `i = 0`: `num = 2` → `remaining = 1 - 2 = -1` → prune (return) - Loop ends → backtrack: `current.pop()` → `current = [2,2]`

Back in Step 3: `dfs(0, 3)`
- `i = 1`: `num = 3` - `current = [2,2,3]` - Call `dfs(1, 0)` → **remaining = 0!** - Append copy: `result = [[2,2,3]]` - Return - Backtrack: `current = [2,2]` - `i = 2`: `num = 6` → 3 - 6 < 0 → skip - Done → backtrack to `current = [2]`

Back in Step 2: `dfs(0, 5)`
- `i = 1`: `num = 3` - `current = [2,3]` - Call `dfs(1, 2)` → all nums 3 → 2-3<0 → no path - Continue… eventually backtrack to root

Back in Step 1: `dfs(0, 7)`
- `i = 3: num = 7 - current = [7] -` Call `dfs(3, 0)` → append → `result = [[2,2,3], [7]]`

Final Result: `[[2,2,3], [7]]`

Key insight: **`start` index prevents duplicates** like `[2,3,2]` by only allowing choices from current index onward.

Complexity Analysis

- **Time Complexity**: `O(2^(T/M))`

  > Where `T = target`, `M = min(candidates)`.
  > In worst case, we explore all combinations of smallest number (e.g., `M=1` → `T` levels, 2 choices per level exponential).
  > Pruning reduces this, but worst-case remains exponential.

- **Space Complexity**: `O(T/M)`

  > Recursion depth is at most `target / min(candidates)` (e.g., all 1s → depth = T).
  > `current` list and call stack scale with depth.
  > Output space not counted per LeetCode convention.

4. Palindrome Pairs

Pattern: Hashing + String Manipulation + Prefix/Suffix Decomposition

---

Problem Statement

> Given a list of **unique** strings `words`, return all the **palindrome pairs** of indices `(i, j)` such that `i != j` and `words[i] + words[j]` is a palindrome.

You may return the answer in **any order**.

A **palindrome** is a string that reads the same backward as forward.

Sample Input & Output

```
Input: words = ["abcd","dcba","lls","s","sssll"]
Output: [[0,1],[1,0],[3,2],[2,4]]
Explanation:
- "abcd" + "dcba" = "abcddcba" → palindrome
- "dcba" + "abcd" = "dcbaabcd" → palindrome
- "s" + "lls" = "slls" → palindrome
- "lls" + "sssll" = "llssssll" → palindrome
```

```
Input: words = ["bat","tab","cat"]
Output: [[0,1],[1,0]]
Explanation: "battab" and "tabbat" are palindromes.
```

```
Input: words = ["a",""]
Output: [[0,1],[1,0]]
Explanation: "a" + "" = "a", and "" + "a" = "a" → both palindromes.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def palindromePairs(self, words: List[str]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Build a word-to-index map for O(1) lookup
        #   - Store reversed words to check complements
        word_to_index = {word: i for i, word in enumerate(words)}
        result = []

        # STEP 2: Main loop / recursion
        #   - For each word, consider all possible splits
        #   - Check if prefix/suffix is palindrome and
        #     the reverse of the other part exists in map
        for i, word in enumerate(words):
            n = len(word)
```

```python
            # Consider all split points: from 0 to n (inclusive)
            for j in range(n + 1):
                prefix = word[:j]
                suffix = word[j:]

                # Case 1: prefix is palindrome → look for reversed suffix
                if self._is_palindrome(prefix):
                    reversed_suffix = suffix[::-1]
                    if (reversed_suffix in word_to_index and
                        word_to_index[reversed_suffix] != i):
                        result.append([word_to_index[reversed_suffix], i])

                # Case 2: suffix is palindrome → look for reversed prefix
                # Avoid double-counting when j == 0 (empty prefix)
                if j != n and self._is_palindrome(suffix):
                    reversed_prefix = prefix[::-1]
                    if (reversed_prefix in word_to_index and
                        word_to_index[reversed_prefix] != i):
                        result.append([i, word_to_index[reversed_prefix]])

        # STEP 3: Update state / bookkeeping
        #   - All valid pairs are appended during iteration
        #   - No duplicates due to unique words and careful case split

        # STEP 4: Return result
        #   - Result list contains all valid [i, j] pairs
        return result

    def _is_palindrome(self, s: str) -> bool:
        # Helper: check if string is palindrome using two pointers
        left, right = 0, len(s) - 1
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True
        # return s == s[::-1] also works with much faster runtime.

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()
```

```python
#   Test 1: Normal case
words1 = ["abcd", "dcba", "lls", "s", "sssll"]
output1 = sol.palindromePairs(words1)
expected1 = [[0,1],[1,0],[3,2],[2,4]]
assert sorted(output1) == sorted(expected1), f"Test 1 failed: {output1}"
print("  Test 1 passed")

#   Test 2: Edge case - empty string
words2 = ["a", ""]
output2 = sol.palindromePairs(words2)
expected2 = [[0,1],[1,0]]
assert sorted(output2) == sorted(expected2), f"Test 2 failed: {output2}"
print("  Test 2 passed")

#   Test 3: Tricky/negative - no pairs
words3 = ["abc", "def", "ghi"]
output3 = sol.palindromePairs(words3)
expected3 = []
assert output3 == expected3, f"Test 3 failed: {output3}"
print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 2**: `words = ["a", ""]`

Goal: Find all `[i, j]` such that `words[i] + words[j]` is a palindrome.

Initialization

- `word_to_index = {"a": 0, "": 1}`
- `result = []`

Process `i = 0`, `word = "a"` (length = 1)

We consider split points `j = 0, 1, 2` → but `range(1 + 1)` = `j = 0, 1`

j = 0:

- `prefix = word[:0] = ""`
- `suffix = word[0:] = "a"`
- Check **Case 1**: Is `prefix` (`""`) a palindrome? Yes.
  - `reversed_suffix = "a"[::-1] = "a"`
  - `"a"` is in map → index 0
  - But `word_to_index["a"] == i (0)` → skip (no self-pair)
- Check **Case 2**: `j != n` → `0 != 1` → true
  - Is `suffix` (`"a"`) a palindrome? Yes.
  - `reversed_prefix = ""[::-1] = ""`
  - `""` is in map → index 1
  - `1 != 0` → valid!
  - Append `[i, index] = [0, 1]` → `result = [[0,1]]`

j = 1:

- `prefix = word[:1] = "a"`
- `suffix = word[1:] = ""`
- **Case 1**: Is `"a"` palindrome? Yes.
  - `reversed_suffix = ""[::-1] = ""`
  - `""` in map → index 1
  - `1 != 0` → valid!
  - Append `[1, 0]` → `result = [[0,1], [1,0]]`
- **Case 2**: `j == n (1)` → skip to avoid double-counting empty suffix

---

Process `i = 1`, `word = ""` (length = 0)

- `j` in `range(0 + 1)` → only `j = 0`
- `prefix = ""`, `suffix = ""`
- **Case 1**: `""` is palindrome → `reversed_suffix = ""`
  - Index = 1, but `i = 1` → skip (self)
- **Case 2**: `j == n (0)` → skip

Final Result

- `result = [[0,1], [1,0]]`
- Matches expected output!

Key Insight:

The empty string acts as a "neutral" element — `"a" + "" = "a"` and `"" + "a" = "a"`, both palindromes.

Complexity Analysis

- **Time Complexity**: `O(n * k²)`
  - `n` = number of words
  - `k` = average word length
  - For each word (`n`), we try `k+1` splits
  - For each split, we check if prefix/suffix is palindrome → `O(k)`
  - Reversing substring and dict lookup → `O(k)`
  - Total: `O(n * k * k) = O(n k²)`
- **Space Complexity**: `O(n * k)`
  - Hash map stores all words → `O(n k)`
  - Result list can be up to `O(n²)` in worst case, but typically much less
  - We count dominant input-dependent storage → `O(n k)`

5. Longest Palindromic Substring

Pattern: Expand Around Centers

Problem Statement

Given a string `s`, return the longest palindromic substring in `s`.
A string is palindromic if it reads the same forward and backward.
You may assume that the maximum length of `s` is 1000.

Sample Input & Output

```
Input: "babad"
Output: "bab"
Explanation: "aba" is also valid; both are length 3.
```

```
Input: "cbbd"
Output: "bb"
Explanation: The longest palindrome is "bb".
```

```
Input: "a"
Output: "a"
Explanation: Single character is trivially a palindrome.
```

LeetCode Editorial Solution + Inline Tests

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        # STEP 1: Initialize structures
        #   - Track the start index and max length of the best palindrome
        if not s:
            return ""
        start = 0
        max_len = 1

        # Helper to expand around center
        def expand(left: int, right: int):
            # Expand while characters match and indices valid
```

```python
            while (left >= 0 and right < len(s) and s[left] == s[right]):
                left -= 1
                right += 1
            # Return length of palindrome centered here
            return right - left - 1

        # STEP 2: Main loop / recursion
        #   - Try every possible center (odd and even length)
        for i in range(len(s)):
            # Odd-length: center at i
            len1 = expand(i, i)
            # Even-length: center between i and i+1
            len2 = expand(i, i + 1)
            current_max = max(len1, len2)

            # STEP 3: Update state / bookkeeping
            #   - If we found a longer palindrome, update start & max_len
            if current_max > max_len:
                max_len = current_max
                # Recalculate start index based on center and length
                start = i - (current_max - 1) // 2

        # STEP 4: Return result
        #   - Slice from start to start + max_len
        return s[start:start + max_len]

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.longestPalindrome("babad") in {"bab", "aba"}

    #  Test 2: Edge case
    assert sol.longestPalindrome("a") == "a"

    #  Test 3: Tricky/negative
    assert sol.longestPalindrome("cbbd") == "bb"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly →

instant feedback.

Example Walkthrough

We'll trace `longestPalindrome("babad")` step by step.

Initial state:
- `s = "babad"`
- `start = 0`, `max_len = 1`

Iteration i = 0 (`s[0] = 'b'`):
- Call `expand(0, 0)` → odd-length center at index 0
- `left=0, right=0`: `'b' == 'b'` → OK → `left=-1, right=1`
- Now `left < 0` → stop
- Returns `1 - (-1) - 1 = 1` → `len1 = 1`
- Call `expand(0, 1)` → even-length between 0 and 1
- `s[0]='b'`, `s[1]='a'` → not equal → returns `1 - 0 - 1 = 0` → `len2 = 0`
- `current_max = max(1, 0) = 1`
- Not > `max_len` (1), so no update

State: `start=0`, `max_len=1`

Iteration i = 1 (`s[1] = 'a'`):
- `expand(1,1)`:
- `'a'=='a'` → expand → `left=0, right=2` → `'b'=='b'` → OK
- Expand → `left=-1, right=3` → stop
- Returns `3 - (-1) - 1 = 3` → `len1 = 3`
- `expand(1,2)`: `'a' vs 'b'` → no match → `len2 = 0`
- `current_max = 3` > `max_len=1` → **update!**
- `max_len = 3`
- `start = 1 - (3 - 1)//2 = 1 - 1 = 0`

State: `start=0`, `max_len=3` → current best: `s[0:3] = "bab"`

Iteration i = 2 (`s[2] = 'b'`):
- `expand(2,2)`:
- `'b'` → expand to `left=1, right=3`: `'a'=='a'` → OK
- Then `left=0, right=4`: `'b'=='d'`? No → stop
- Returns `4 - 0 - 1 = 3` → `len1 = 3`
- `expand(2,3)`: `'b' vs 'a'` → no → `len2 = 0`
- `current_max = 3`, not > 3 → no update

State unchanged

Iteration i = 3 (`s[3] = 'a'`):
- `expand(3,3)`: `'a'` → expand to `left=2, right=4`: `'b' vs 'd'` → no → `len1 = 1`
- `expand(3,4)`: `'a' vs 'd'` → no → `len2 = 0`
- No update

Iteration i = 4: similar → `len1=1`, no update

Final return: `s[0:0+3] = "bab"`

Output: `"bab"` (or `"aba"` would also be acceptable per problem)

Complexity Analysis

- **Time Complexity**: $O(n^2)$

  We iterate through `n` centers. For each, `expand()` may scan up to `n` characters in worst case (e.g., all same char like "aaaa"). So total ~ $n * n = O(n^2)$.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`start`, `max_len`, loop indices). The helper function uses constant extra space (no recursion stack beyond current call). Input string not counted.

6. Longest Common Prefix

Pattern: Strings & Prefix Matching

Problem Statement

Write a function to find the longest common prefix string amongst an array of strings.
If there is no common prefix, return an empty string `""`.

Sample Input & Output

```
Input: ["flower","flow","flight"]
Output: "fl"
Explanation: The common prefix across all strings is "fl".
```

```
Input: ["dog","racecar","car"]
Output: ""
Explanation: No common prefix exists among the strings.
```

```
Input: [""]
Output: ""
Explanation: Single empty string has no prefix.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        # STEP 1: Initialize structures
        #   - Use first string as reference for prefix
        #   - Early exit if empty input or first string is empty

        if not strs or not strs[0]:
            return ""

        prefix = strs[0]

        # STEP 2: Main loop / recursion
        #   - Compare prefix with each subsequent string
        #   - Shorten prefix until it matches start of current string

        for i in range(1, len(strs)):
            # Reduce prefix until it matches beginning of strs[i]
            while not strs[i].startswith(prefix):
                prefix = prefix[:-1]
                # If prefix becomes empty, no common prefix exists
                if not prefix:
                    return ""

        # STEP 3: Update state / bookkeeping
        #   - Prefix is updated in-place during loop
        #   - No extra bookkeeping needed beyond prefix trimming

        # STEP 4: Return result
        #   - Return final prefix after all comparisons
        return prefix

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.longestCommonPrefix(
        ["flower", "flow", "flight"]
    )
    print(f"Test 1: '{result1}'")  # Expected: 'fl'
```

```python
#   Test 2: Edge case
result2 = sol.longestCommonPrefix([""])
print(f"Test 2: '{result2}'")  # Expected: ''

#   Test 3: Tricky/negative
result3 = sol.longestCommonPrefix(
    ["dog", "racecar", "car"]
)
print(f"Test 3: '{result3}'")  # Expected: ''
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `["flower", "flow", "flight"]`.

Step 1:
- **Code**: `if not strs or not strs[0]: return ""`
- **Check**: `strs` is not empty, and `strs[0] = "flower"` is not empty.
- **Action**: Skip return, continue.
- **State**: `prefix = "flower"`

Step 2:
- **Loop starts**: `i = 1`, current string = `"flow"`
- **Check**: Does `"flow".startswith("flower")`? → **No**
- **Enter while loop**: - **Iteration 1**: `prefix = "flower"[:-1] = "flowe"`
→ `"flow".startswith("flowe")`? No
- **Iteration 2**: `prefix = "flowe"[:-1] = "flow"`
→ `"flow".startswith("flow")`? **Yes** → exit while
- **State**: `prefix = "flow"`

Step 3:
- **Loop continues**: `i = 2`, current string = `"flight"`
- **Check**: Does `"flight".startswith("flow")`? → **No**
- **Enter while loop**: - **Iteration 1**: `prefix = "flow"[:-1] = "flo"`
→ `"flight".startswith("flo")`? No
- **Iteration 2**: `prefix = "flo"[:-1] = "fl"`
→ `"flight".startswith("fl")`? **Yes** → exit while
- **State**: `prefix = "fl"`

Step 4:
- Loop ends (all strings processed).
- **Return**: `"fl"`

Final Output: `'fl'` — matches expected.

Complexity Analysis

- **Time Complexity**: `O(S)`

  Where `S` is the sum of all characters in all strings.
  In worst case (all strings identical), we compare every character of the first string with every other string → total character comparisons = `S`.

- **Space Complexity**: `O(1)`

  Only using a single `prefix` variable that references substrings of the first string.
  No additional data structures scale with input size.
  (Note: String slicing creates new strings in Python, but total extra space is bounded by length of first string → still `O(m)` where `m = len(strs[0])`, but often considered `O(1)` auxiliary space in editorial contexts.)

7. String to Integer (atoi)

Pattern: String Manipulation

Problem Statement

Implement the `myAtoi(string s)` function, which converts a string to a 32-bit signed integer.

The algorithm for `myAtoi(string s)` is as follows:

1. Read in and ignore any leading whitespace.
2. Check if the next character (if not already at the end of the string) is `'-'` or `'+'`. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. `"123" -> 123`, `"0032" -> 32`). If no digits were read, then the integer is `0`. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range $[-2^{31}, 2^{31} - 1]$, then clamp the integer so that it remains in the range. Specifically, integers less than -2^{31} should be clamped to -2^{31}, and integers greater than $2^{31} - 1$ should be clamped to $2^{31} - 1$.
6. Return the integer as the final result.

Note: - Only the space character `' '` is considered a whitespace character. - Do not ignore any characters other than the leading whitespace or the rest of the string after the digits.

Sample Input & Output

```
Input: "42"
Output: 42
Explanation: No whitespace, no sign, just digits → 42.
```

```
Input: "   -42"
Output: -42
Explanation: Leading spaces ignored, '-' sets sign, digits "42" → -42.
```

```
Input: "4193 with words"
Output: 4193
Explanation: Digits stop at space; rest ignored.
```

```
Input: "words and 987"
Output: 0
Explanation: No valid digits after optional sign → 0.
```

```
Input: "-91283472332"
Output: -2147483648
Explanation: Value < -2³¹ → clamped to -2³¹.
```

LeetCode Editorial Solution + Inline Tests

```python
class Solution:
    def myAtoi(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - index tracks current position in string
        #   - sign starts positive; flipped if '-' seen
        #   - result accumulates the numeric value
        index = 0
        sign = 1
        result = 0
        n = len(s)

        # Skip leading whitespace
        while index < n and s[index] == ' ':
            index += 1

        # Check for sign
        if index < n and (s[index] == '+' or s[index] == '-'):
            sign = -1 if s[index] == '-' else 1
            index += 1

        # STEP 2: Main loop / recursion
        #   - Process digits until non-digit or end
        #   - Break on non-digit per problem rules
        while index < n and s[index].isdigit():
            digit = int(s[index])

            # STEP 3: Update state / bookkeeping
            #   - Clamp before overflow using bounds check
```

```
            #   - Prevents Python int overflow (not needed in C++/Java)
            if result > (2**31 - 1 - digit) // 10:
                return -2**31 if sign == -1 else 2**31 - 1

            result = result * 10 + digit
            index += 1

        # STEP 4: Return result
        #   - Apply sign and clamp if needed (though clamping
        #     already handled during digit processing)
        final = sign * result
        INT_MIN, INT_MAX = -2**31, 2**31 - 1
        if final < INT_MIN:
            return INT_MIN
        if final > INT_MAX:
            return INT_MAX
        return final

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.myAtoi("42") == 42

    #  Test 2: Edge case (leading spaces + sign)
    assert sol.myAtoi("   -42") == -42

    #  Test 3: Tricky/negative (clamp overflow)
    assert sol.myAtoi("-91283472332") == -2147483648

    # Additional robustness checks
    assert sol.myAtoi("4193 with words") == 4193
    assert sol.myAtoi("words and 987") == 0
    assert sol.myAtoi("") == 0
    assert sol.myAtoi("+1") == 1
    assert sol.myAtoi("+-12") == 0  # invalid after sign
    assert sol.myAtoi("   +0 123") == 0  # stops at space after 0

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly →

instant feedback.

Example Walkthrough

We'll walk through `myAtoi("   -42")` step by step.

Initial state:
`s = "   -42"`, `index = 0`, `sign = 1`, `result = 0`, `n = 6`

Step 1: Skip leading whitespace
- `index=0`: `s[0] = ' '` → skip → `index = 1`
- `index=1`: `s[1] = ' '` → skip → `index = 2`
- `index=2`: `s[2] = ' '` → skip → `index = 3`
- `index=3`: `s[3] = '-'` → not space → exit loop

State: `index=3`, `sign=1`, `result=0`

Step 2: Check for sign
- `index=3 < 6` and `s[3] == '-'` → set `sign = -1`
- Increment `index` → `index = 4`

State: `index=4`, `sign=-1`, `result=0`

Step 3: Process digits
- `index=4`: `s[4] = '4'` → digit = 4
- Check overflow: `result=0`, `(2³¹-1 - 4)//10   214748364` → 0 that → OK
- `result = 0*10 + 4 = 4`
- `index = 5`
- `index=5`: `s[5] = '2'` → digit = 2
- Check overflow: `result=4`, `(2147483647 - 2)//10 = 214748364` → 4 that → OK
- `result = 4*10 + 2 = 42`
- `index = 6`
- `index=6 == n` → exit loop

State: `result=42`, `sign=-1`

Step 4: Apply sign and clamp
- `final = -1 * 42 = -42`
- `-42` is between `-2147483648` and `2147483647` → return `-42`

Final Output: `-42`

Complexity Analysis

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

  We scan the string at most once: skipping whitespace, checking sign, and reading digits. Each character visited once → linear in input length.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`index`, `sign`, `result`, etc.) are used. No extra data structures scale with input size.

8. Encode and Decode Strings

Pattern: String Manipulation

Problem Statement

Design an algorithm to encode a list of strings to a single string, and decode the single string back into the original list of strings.

The encoded string must be decodable without any additional information (e.g., delimiters like commas may appear in input strings).

Note: The string may contain any possible characters, including non-printable ones. Do **not** use class variables or global state.

Sample Input & Output

```
Input: ["hello", "world"]
Output: ["hello", "world"]
Explanation: Encoded string might be "5:hello5:world". Decoding splits on length prefixes.
```

```
Input: [""]
Output: [""]
Explanation: Empty string must be handled - encoded as "0:".
```

```
Input: ["", "a", "bc", ""]
Output: ["", "a", "bc", ""]
Explanation: Mixed empty and non-empty strings; lengths vary.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def encode(self, strs: List[str]) -> str:
        # STEP 1: Initialize encoded string
        #   - Use length + ':' + string format to avoid delimiter
        #     collision (since ':' is not used in length part)
        encoded = []

        # STEP 2: Main loop over each string
        #   - Prepend each string with its length and a colon
        #   - This creates unambiguous segments
        for s in strs:
            encoded.append(str(len(s)) + ':' + s)

        # STEP 3: Join all segments into one string
        #   - No extra separator needed; length tells us where
        #     each string ends
        return ''.join(encoded)
```

```
    def decode(self, s: str) -> List[str]:
        # STEP 1: Initialize result list and pointer
        #   - Use index i to traverse the encoded string
        decoded = []
        i = 0

        # STEP 2: Main loop while not at end
        #   - Find next colon to extract length prefix
        #   - Convert prefix to integer → tells us how many
        #     chars to read next
        while i < len(s):
            # Locate colon after current length digits
            colon_idx = s.find(':', i)
            if colon_idx == -1:
                break  # Should not happen in valid input

            # Parse length from digits before colon
            length = int(s[i:colon_idx])

            # STEP 3: Extract actual string using length
            #   - Start after colon, take 'length' chars
            start = colon_idx + 1
            decoded.append(s[start:start + length])

            # Update pointer to next segment start
            i = start + length

        # STEP 4: Return decoded list
        #   - Handles empty strings (length=0) naturally
        return decoded

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    input1 = ["hello", "world"]
    encoded1 = sol.encode(input1)
    decoded1 = sol.decode(encoded1)
    assert decoded1 == input1, f"Test 1 failed: {decoded1}"
    print(" Test 1 passed:", decoded1)
```

```python
#   Test 2: Edge case - single empty string
input2 = [""]
encoded2 = sol.encode(input2)
decoded2 = sol.decode(encoded2)
assert decoded2 == input2, f"Test 2 failed: {decoded2}"
print("  Test 2 passed:", decoded2)

#   Test 3: Tricky/negative - mixed empty & non-empty
input3 = ["", "a", "bc", ""]
encoded3 = sol.encode(input3)
decoded3 = sol.decode(encoded3)
assert decoded3 == input3, f"Test 3 failed: {decoded3}"
print("  Test 3 passed:", decoded3)
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3**: `["", "a", "bc", ""]` step by step.

Step 1: `encode(["", "a", "bc", ""])`

- Initialize `encoded = []`
- Loop over each string:

  String 1: `""` (empty)

  - `len("") = 0`
  - Append `"0:" + ""` → `"0:"`
  - `encoded = ["0:"]`

  String 2: `"a"`

  - `len("a") = 1`

– Append `"1:" + "a"` → `"1:a"`

– `encoded = ["0:", "1:a"]`

String 3: `"bc"`

– `len("bc") = 2`

– Append `"2:" + "bc"` → `"2:bc"`

– `encoded = ["0:", "1:a", "2:bc"]`

String 4: `""`

– Append `"0:"` again

– `encoded = ["0:", "1:a", "2:bc", "0:"]`

- Return `''.join(encoded)` → `"0:1:a2:bc0:"`

Encoded string: `"0:1:a2:bc0:"`

Step 2: `decode("0:1:a2:bc0:")`

- Initialize `decoded = []`, `i = 0`

Iteration 1: `i = 0`
- `s.find(':', 0)` → index 1 (`"0:"`)
- `length = int(s[0:1]) = int("0") = 0`
- `start = 1 + 1 = 2`
- Extract `s[2:2+0] = s[2:2] = ""`
- Append `""` → `decoded = [""]`
- Update `i = 2 + 0 = 2`

Iteration 2: `i = 2`
- `s.find(':', 2)` → index 3 (`"1:"` starts at 2)
- `length = int(s[2:3]) = 1`
- `start = 3 + 1 = 4`
- Extract `s[4:4+1] = "a"`
- `decoded = ["", "a"]`
- `i = 4 + 1 = 5`

Iteration 3: `i = 5`
- `s.find(':', 5)` → index 6 (`"2:"`)
- `length = int("2") = 2`
- `start = 7`
- Extract `s[7:9] = "bc"`
- `decoded = ["", "a", "bc"]`
- `i = 7 + 2 = 9`

Iteration 4: `i = 9`
- `s.find(':', 9)` → index 10 (`"0:"`)
- `length = 0`
- `start = 11`
- Extract `s[11:11] = ""`
- `decoded = ["", "a", "bc", ""]`
- `i = 11 + 0 = 11` → end of string

Decoded: `["", "a", "bc", ""]` — matches input!

Complexity Analysis

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

  Where `n` is the total number of characters across all input strings.
  Both `encode` and `decode` scan each character exactly once.
  `find(':')` runs in linear time but over disjoint segments — total work remains `O(n)`.

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

  The encoded string and decoded list both store all input characters.
  Intermediate lists (`encoded`) also scale with total input size.
  No recursion or deep nesting — only linear extra space.

9. Longest Palindrome

Pattern: String Manipulation

Problem Statement

Given a string **s** which consists of lowercase or uppercase letters, return the length of the **longest palindrome** that can be built with those letters.

Letters are **case sensitive**, so **"Aa"** is not considered a palindrome.

Sample Input & Output

```
Input: "abccccdd"
Output: 7
Explanation: One longest palindrome is "dccaccd", which uses 7 letters.
```

```
Input: "a"
Output: 1
Explanation: Only one character - it forms a palindrome of length 1.
```

```
Input: "Aa"
Output: 1
Explanation: 'A' and 'a' are different; only one can be used as center.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import Counter

class Solution:
    def longestPalindrome(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - Count frequency of each character.
        #   - Palindromes use pairs + optional single center.
        char_counts = Counter(s)
        length = 0
        has_odd = False
```

```python
        # STEP 2: Main loop / recursion
        #   - For each character count:
        #       * Use all even parts (count // 2 * 2)
        #       * Track if any odd exists for center
        for count in char_counts.values():
            # Add even pairs to length
            length += (count // 2) * 2

            # STEP 3: Update state / bookkeeping
            #   - If any char has odd count, we can place
            #     one in the center (only once).
            if count % 2 == 1:
                has_odd = True

        # STEP 4: Return result
        #   - If there's an unused odd char, add 1 for center
        if has_odd:
            length += 1

        return length

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.longestPalindrome("abccccdd") == 7

    #  Test 2: Edge case
    assert sol.longestPalindrome("a") == 1

    #  Test 3: Tricky/negative
    assert sol.longestPalindrome("Aa") == 1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `longestPalindrome("abccccdd")` step by step.

Initial state:
- Input string: `"abccccdd"`
- `char_counts = Counter({'c': 4, 'd': 2, 'a': 1, 'b': 1})`
- `length = 0`
- `has_odd = False`

Step 1: Process `'c'` → count = 4
- `4 // 2 = 2` → `2 * 2 = 4`
- `length += 4` → `length = 4`
- `4 % 2 == 0` → `has_odd` remains `False`

State: `length=4`, `has_odd=False`

Step 2: Process `'d'` → count = 2
- `2 // 2 = 1` → `1 * 2 = 2`
- `length += 2` → `length = 6`
- `2 % 2 == 0` → `has_odd` still `False`

State: `length=6`, `has_odd=False`

Step 3: Process `'a'` → count = 1
- `1 // 2 = 0` → `0 * 2 = 0`
- `length += 0` → `length = 6`
- `1 % 2 == 1` → set `has_odd = True`

State: `length=6`, `has_odd=True`

Step 4: Process `'b'` → count = 1
- `1 // 2 = 0` → add 0 → `length = 6`
- `1 % 2 == 1` → `has_odd` already `True`, stays `True`

State: `length=6`, `has_odd=True`

Final Step: After loop
- Since `has_odd` is `True`, add 1 → `length = 7`
- Return `7`

Final Output: `7`
Key Insight: We use **all even pairs** from every character, and **at most one odd character** as the center.

Complexity Analysis

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

  We iterate over the string once to build the counter (`O(n)`) and then over at most 52 keys (letters a–z, A–Z), which is constant. So total is `O(n)`.

- **Space Complexity**: `O(1)`

  The counter stores at most 52 distinct characters (26 lowercase + 26 uppercase). This is bounded by a constant, so space is `O(1)`.

10. Product of Array Except Self

Pattern: Arrays & Hashing (Prefix/Suffix Products)

Problem Statement

Given an integer array `nums`, return an array `answer` such that `answer[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer. You must write an algorithm that runs in `O(n)` time and without using division.

Sample Input & Output

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
Explanation: 2*3*4=24, 1*3*4=12, 1*2*4=8, 1*2*3=6
```

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
Explanation: Only at index 2 is there no zero; product = (-1)*1*(-3)*3 = 9
```

```
Input: nums = [0,0]
Output: [0,0]
Explanation: Every product includes at least one zero
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        # STEP 1: Initialize result array with 1s
        #   - We'll use this to store prefix products first,
        #     then multiply by suffix products in-place.
        answer = [1] * n
```

```
        # STEP 2: Compute prefix products (left to right)
        #   - answer[i] = product of all elements before i
        #   - This captures "everything to the left"
        for i in range(1, n):
            answer[i] = answer[i - 1] * nums[i - 1]

        # STEP 3: Compute suffix products (right to left)
        #   - Use a single variable to track running suffix
        #   - Multiply it into answer[i] to combine left & right
        suffix = 1
        for i in range(n - 1, -1, -1):
            answer[i] *= suffix
            suffix *= nums[i]

        # STEP 4: Return result
        #   - No edge case handling needed: works for n=1, zeros, negatives
        return answer

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.productExceptSelf([1,2,3,4]) == [24,12,8,6], \
        "Normal case failed"

    #   Test 2: Edge case - contains zero
    assert sol.productExceptSelf([-1,1,0,-3,3]) == [0,0,9,0,0], \
        "Zero case failed"

    #   Test 3: Tricky/negative - all zeros or two zeros
    assert sol.productExceptSelf([0,0]) == [0,0], \
        "Double zero case failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `nums = [1, 2, 3, 4]` step by step.

Initial state:
- `nums = [1, 2, 3, 4]`
- `n = 4`
- `answer = [1, 1, 1, 1]`

Step 1: Prefix pass (left to right)
Loop from `i = 1` to `3`:

- **i = 1**:
  `answer[1] = answer[0] * nums[0] = 1 * 1 = 1`
  → `answer = [1, 1, 1, 1]`
- **i = 2**:
  `answer[2] = answer[1] * nums[1] = 1 * 2 = 2`
  → `answer = [1, 1, 2, 1]`
- **i = 3**:
  `answer[3] = answer[2] * nums[2] = 2 * 3 = 6`
  → `answer = [1, 1, 2, 6]`

Now `answer[i]` = product of all elements **before** index `i`.

Step 2: Suffix pass (right to left)
Initialize `suffix = 1`
Loop from `i = 3` down to `0`:

- **i = 3**:
  `answer[3] *= suffix → 6 * 1 = 6`
  Then update `suffix = suffix * nums[3] = 1 * 4 = 4`
  → `answer = [1, 1, 2, 6]`
- **i = 2**:
  `answer[2] *= suffix → 2 * 4 = 8`
  `suffix = 4 * nums[2] = 4 * 3 = 12`
  → `answer = [1, 1, 8, 6]`

- **i = 1**:
  `answer[1] *= suffix → 1 * 12 = 12`
  `suffix = 12 * nums[1] = 12 * 2 = 24`
  → `answer = [1, 12, 8, 6]`
- **i = 0**:
  `answer[0] *= suffix → 1 * 24 = 24`
  `suffix = 24 * nums[0] = 24 * 1 = 24`
  → `answer = [24, 12, 8, 6]`

Final output: `[24, 12, 8, 6]`

Key insight:
We never use division. Instead, we split the product into **left (prefix)** and **right (suffix)** parts, then combine them in two passes.

Complexity Analysis

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

  Two separate linear passes over the array (each `O(n)`), so total `O(2n) = O(n)`.

- **Space Complexity**: `O(1)`

  Only using the output array (`answer`) and one extra variable (`suffix`).
  The problem states output space doesn't count toward space complexity, so auxiliary space is `O(1)`.

11. Best Time to Buy and Sell Stock

Pattern: Greedy / One-Pass Tracking

Problem Statement

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return `0`.

Sample Input & Output

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6 - 1 = 5.
```

```
Input: [7,6,4,3,1]
Output: 0
Explanation: Prices only decrease. No profitable transaction possible.
```

```
Input: [1]
Output: 0
Explanation: Only one day - can't buy and sell on same day.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - min_price tracks the lowest price seen so far (best buy day)
        #   - max_profit tracks the highest profit achievable up to current day
```

```python
        min_price = float('inf')
        max_profit = 0

        # STEP 2: Main loop / recursion
        #   - Iterate through each day's price
        #   - Invariant: min_price is always the cheapest day before current day
        for price in prices:
            # STEP 3: Update state / bookkeeping
            #   - Update min_price if we find a cheaper day
            if price < min_price:
                min_price = price
            #   - Calculate profit if sold today, update max_profit if better
            elif price - min_price > max_profit:
                max_profit = price - min_price

        # STEP 4: Return result
        #   - max_profit is 0 if no profitable trade exists (handles edge cases)
        return max_profit

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.maxProfit([7,1,5,3,6,4]) == 5, "Normal case failed"

    #  Test 2: Edge case - decreasing prices
    assert sol.maxProfit([7,6,4,3,1]) == 0, "Decreasing prices failed"

    #  Test 3: Tricky/negative - single day
    assert sol.maxProfit([1]) == 0, "Single day failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `maxProfit([7,1,5,3,6,4])` step by step.

Initial State:
- `min_price = inf`
- `max_profit = 0`

Step 1: `price = 7`
- Is `7 < inf`? → **Yes**
- Update `min_price = 7`
- Skip profit check (since we just updated min)
- **State**: `min_price=7`, `max_profit=0`

Step 2: `price = 1`
- Is `1 < 7`? → **Yes**
- Update `min_price = 1`
- **State**: `min_price=1`, `max_profit=0`

Step 3: `price = 5`
- Is `5 < 1`? → **No**
- Check profit: `5 - 1 = 4`
- Is `4 > 0`? → **Yes** → `max_profit = 4`
- **State**: `min_price=1`, `max_profit=4`

Step 4: `price = 3`
- Is `3 < 1`? → **No**
- Profit: `3 - 1 = 2`
- Is `2 > 4`? → **No** → no change
- **State**: `min_price=1`, `max_profit=4`

Step 5: `price = 6`
- Is `6 < 1`? → **No**
- Profit: `6 - 1 = 5`
- Is `5 > 4`? → **Yes** → `max_profit = 5`
- **State**: `min_price=1`, `max_profit=5`

Step 6: `price = 4`
- Is `4 < 1`? → **No**
- Profit: `4 - 1 = 3`
- Is `3 > 5`? → **No** → no change
- **State**: `min_price=1`, `max_profit=5`

Final Return: `5`

> **Key Insight**: We never need to remember past days — just the cheapest price so far and best profit. This is the **greedy** pattern: make the locally optimal choice (track min price) to reach global optimum.

Complexity Analysis

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

  We iterate through the `prices` list exactly once. Each operation inside the loop is `O(1)`.

- **Space Complexity**: `O(1)`

  Only two extra variables (`min_price`, `max_profit`) are used, regardless of input size. No scaling data structures.

12. Ransom Note

Pattern: Arrays & Hashing (Frequency Counting)

Problem Statement

Given two strings `ransomNote` and `magazine`, return `true` if `ransomNote` can be constructed by using the letters from `magazine` and `false` otherwise.
Each letter in `magazine` can only be used once in `ransomNote`.

Sample Input & Output

```
Input: ransomNote = "a", magazine = "b"
Output: false
Explanation: 'a' is not present in magazine.
```

```
Input: ransomNote = "aa", magazine = "ab"
Output: false
Explanation: Only one 'a' in magazine, but need two.
```

```
Input: ransomNote = "aa", magazine = "aab"
Output: true
Explanation: Two 'a's and one 'b' in magazine → enough for "aa".
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import Counter

class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        # STEP 1: Initialize structures
        #   - Use Counter to count frequency of each char in magazine.
        #   - This lets us track available letters efficiently.
        mag_count = Counter(magazine)

        # STEP 2: Main loop / recursion
        #   - Iterate over each character needed in ransomNote.
        #   - Invariant: mag_count[c] reflects remaining uses of char c.
        for char in ransomNote:
            # STEP 3: Update state / bookkeeping
            #   - If char not in mag_count or count is 0, fail.
            #   - Decrement count to "use" one occurrence.
            if mag_count[char] <= 0:
                return False
            mag_count[char] -= 1

        # STEP 4: Return result
        #   - If loop completes, all chars were available → success.
        return True

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.canConstruct("aa", "aab") == True, "Normal case failed"
```

```python
#  Test 2: Edge case - empty ransom note
assert sol.canConstruct("", "abc") == True, "Empty note should pass"

#  Test 3: Tricky/negative - insufficient letters
assert sol.canConstruct("aab", "aa") == False, "Insufficient letters"

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **`canConstruct("aa", "aab")`** step by step.

Step 1: Initialize `mag_count`
- Code: `mag_count = Counter(magazine)`
- `magazine = "aab"`
- `Counter("aab")` → counts: `{'a': 2, 'b': 1}`
- **State**: `mag_count = {'a': 2, 'b': 1}`

Step 2: Start loop over `ransomNote = "aa"`
- First character: `char = 'a'`

Step 2a: Check availability
- Code: `if mag_count[char] <= 0` → `mag_count['a'] = 2` → condition is `False`
- Proceed to decrement

Step 2b: Use one 'a'
- Code: `mag_count['a'] -= 1` → `2 - 1 = 1`
- **State**: `mag_count = {'a': 1, 'b': 1}`

Step 3: Next character in loop
- Second character: `char = 'a'`

Step 3a: Check availability
- `mag_count['a'] = 1` → still > 0 → condition `False`
- Proceed

Step 3b: Use another 'a'
- `mag_count['a'] = 1 - 1 = 0`
- **State**: `mag_count = {'a': 0, 'b': 1}`

Step 4: Loop ends
- All characters in `"aa"` processed successfully
- Code reaches `return True`
- **Final Output**: `True`

Key Takeaway:
We never "build" the note — we just **verify availability** by counting down from what's in the magazine.
This avoids modifying strings (which is slow) and uses **hash map frequency tracking**, a core technique in the *Arrays & Hashing* pattern.

Complexity Analysis

- **Time Complexity**: `O(m + n)`

  Where `m = len(magazine)` and `n = len(ransomNote)`.
  Building `Counter(magazine)` takes `O(m)`.
  Looping through `ransomNote` takes `O(n)`.
  Each hash map lookup/update is `O(1)` average.

- **Space Complexity**: `O(k)`

  Where `k` is the number of unique characters in `magazine`.
  In worst case (all chars unique), `k = m`, so `O(m)`.
  But for English letters, `k  26` → effectively `O(1)` in practice.

13. Insert Delete GetRandom O(1)

Pattern: Arrays & Hashing (Index Mapping)

Problem Statement

Implement the `RandomizedSet` class: - `RandomizedSet()` Initializes the `RandomizedSet` object. - `bool insert(int val)` Inserts an item `val` into the set if not present. Returns `true` if the item was not present, `false` otherwise. - `bool remove(int val)` Removes an item `val` from the set if present. Returns `true` if the item was present, `false` otherwise. - `int getRandom()` Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average** `O(1)` time complexity.

Sample Input & Output

```
Input: ["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getl
       [[], [1], [2], [2], [], [1], [2], []]
Output: [null, true, false, true, 2, true, false, 2]
Explanation:
- insert(1): 1 not present → add → return true
- remove(2): 2 not present → return false
- insert(2): 2 not present → add → return true
- getRandom(): set = {1,2} → return 1 or 2 (e.g., 2)
- remove(1): 1 present → remove → return true
- insert(2): 2 already present → return false
- getRandom(): set = {2} → return 2
```

```
Input: ["RandomizedSet", "insert", "insert", "getRandom"]
       [[], [0], [1], []]
Output: [null, true, true, 0 or 1]
```

```
Input: ["RandomizedSet", "insert", "remove", "remove", "insert", "getRandom"]
       [[], [1], [1], [1], [1], []]
Output: [null, true, true, false, true, 1]
```

LeetCode Editorial Solution + Inline Tests

```python
import random
from typing import Dict, List

class RandomizedSet:
    def __init__(self):
        # STEP 1: Initialize structures
        #   - nums: dynamic array to store values for O(1) random access
        #   - val_to_index: hashmap to map value → its index in nums
        self.nums: List[int] = []
        self.val_to_index: Dict[int, int] = {}

    def insert(self, val: int) -> bool:
        # STEP 2: Check existence via hashmap (O(1))
        if val in self.val_to_index:
            return False

        # STEP 3: Append to end of list and record index
        self.val_to_index[val] = len(self.nums)
        self.nums.append(val)
        return True

    def remove(self, val: int) -> bool:
        # STEP 2: Check if value exists
        if val not in self.val_to_index:
            return False

        # STEP 3: Swap target with last element for O(1) removal
        last_val = self.nums[-1]
        idx = self.val_to_index[val]

        # Move last element to target's position
        self.nums[idx] = last_val
```

```
        self.val_to_index[last_val] = idx

        # STEP 4: Remove last element and clean up map
        self.nums.pop()
        del self.val_to_index[val]
        return True

    def getRandom(self) -> int:
        # STEP 4: Return random element from list
        #   - random.choice is O(1) for list access
        return random.choice(self.nums)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    #  Test 1: Normal case
    rs = RandomizedSet()
    assert rs.insert(1) == True
    assert rs.remove(2) == False
    assert rs.insert(2) == True
    rand = rs.getRandom()
    assert rand in {1, 2}
    assert rs.remove(1) == True
    assert rs.insert(2) == False
    assert rs.getRandom() == 2
    print("  Test 1 passed")

    #  Test 2: Edge case - single element
    rs2 = RandomizedSet()
    rs2.insert(0)
    assert rs2.getRandom() == 0
    rs2.remove(0)
    rs2.insert(5)
    assert rs2.getRandom() == 5
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - repeated ops
    rs3 = RandomizedSet()
    assert rs3.insert(1) == True
    assert rs3.remove(1) == True
    assert rs3.remove(1) == False  # already removed
    assert rs3.insert(1) == True
    assert rs3.insert(1) == False  # already present
```

```
assert rs3.getRandom() == 1
print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through **Test 1** step by step:

1. **`rs = RandomizedSet()`**
   - `self.nums = []`
   - `self.val_to_index = {}`
     → Empty set initialized.
2. **`rs.insert(1)`**
   - Check: `1 in {}`? → No.
   - Set `val_to_index[1] = len([]) = 0`
   - Append `1` → `nums = [1]`
     → Returns `True`.
3. **`rs.remove(2)`**
   - Check: `2 in {1:0}`? → No.
     → Returns `False`. State unchanged.
4. **`rs.insert(2)`**
   - Check: `2 in {1:0}`? → No.
   - Set `val_to_index[2] = len([1]) = 1`
   - Append `2` → `nums = [1, 2]`
     → Returns `True`.
5. **`rs.getRandom()`**
   - `random.choice([1, 2])` → suppose it returns `2`.
     → Output: `2`.

6. **`rs.remove(1)`**
    - `1` is in map at index `0`.
    - `last_val = 2`, `idx = 0`
    - Set `nums[0] = 2` → `nums = [2, 2]`
    - Update `val_to_index[2] = 0`
    - Pop last → `nums = [2]`
    - Delete `val_to_index[1]` → map = `{2: 0}`
      → Returns `True`.
7. **`rs.insert(2)`**
    - `2 in {2:0}` → Yes.
      → Returns `False`.
8. **`rs.getRandom()`**
    - Only element is `2` → returns `2`.

Final state: `nums = [2]`, `val_to_index = {2: 0}`

Key Insight: By swapping the element to remove with the last one, we avoid shifting the entire array—enabling **O(1) removal** while keeping the list compact for random access.

Complexity Analysis

- **Time Complexity**: `O(1)` average for all operations
    - `insert`: Hashmap lookup + list append = O(1)
    - `remove`: Hashmap lookup + swap + pop + hashmap update = O(1)
    - `getRandom`: `random.choice` on list = O(1)
- **Space Complexity**: `O(N)`
    - We store each value once in `nums` and once in `val_to_index`
    - Total space scales linearly with number of elements `N`

14. First Missing Positive

Pattern: Arrays & Hashing (Index as Hash Key)

Problem Statement

> Given an unsorted integer array `nums`, return the smallest missing positive integer. You must implement an algorithm that runs in `O(n)` time and uses constant extra space.

Sample Input & Output

```
Input: [1,2,0]
Output: 3
Explanation: 1 and 2 are present; 3 is the first missing positive.
```

```
Input: [3,4,-1,1]
Output: 2
Explanation: 1 and 3 and 4 are present; 2 is missing.
```

```
Input: [7,8,9,11,12]
Output: 1
Explanation: No positive integers from 1 onward are present.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - We'll use the input array itself as a hash table.
        #   - Valid answers must be in [1, n+1], so ignore
        #     numbers outside this range.
        n = len(nums)

        # STEP 2: Main loop / recursion
        #   - Place each number x in position x-1 if 1 <= x <= n.
        #   - Use cyclic sort: keep swapping until current pos
        #     has correct value or invalid number.
        for i in range(n):
            while (1 <= nums[i] <= n and
                   nums[nums[i] - 1] != nums[i]):
                # Swap nums[i] to its correct position
                correct_idx = nums[i] - 1
                nums[i], nums[correct_idx] = \
                    nums[correct_idx], nums[i]

        # STEP 3: Update state / bookkeeping
        #   - Now scan for first index i where nums[i] != i+1.
        #   - That means i+1 is missing.
        for i in range(n):
            if nums[i] != i + 1:
                return i + 1

        # STEP 4: Return result
        #   - If all positions 0..n-1 have 1..n, then n+1 is missing.
        return n + 1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.firstMissingPositive([1, 2, 0]) == 3

    #  Test 2: Edge case - all out of range
    assert sol.firstMissingPositive([7, 8, 9, 11, 12]) == 1
```

```python
#  Test 3: Tricky/negative - duplicates and negatives
assert sol.firstMissingPositive([3, 4, -1, 1]) == 2
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**: `nums = [3, 4, -1, 1]`

Goal: Find the smallest missing positive integer.

Initial State

- `nums = [3, 4, -1, 1]`
- `n = 4`
- Valid positive integers we care about: `1, 2, 3, 4`
  (because answer must be in `[1, n+1] = [1,5]`)

Step 1: Cyclic Sort Pass (`i = 0`)

- `i = 0`, `nums[0] = 3`
- Is `1 <= 3 <= 4`? Yes.
- Is `nums[3 - 1] = nums[2] = -1` equal to 3? No → swap!
- Swap `nums[0]` and `nums[2]`:
  - `nums` becomes `[-1, 4, 3, 1]`

Now `i = 0`, `nums[0] = -1` - `-1` is not in `[1,4]` → skip. Move to next `i`.

Step 2: `i = 1`

- `nums[1] = 4`
- `1 <= 4 <= 4`?
- `nums[4 - 1] = nums[3] = 1  4` → swap!
- Swap `nums[1]` and `nums[3]`:
  - `nums` becomes `[-1, 1, 3, 4]`

Now `i = 1`, `nums[1] = 1 - 1` is valid. - `nums[1 - 1] = nums[0] = -1  1` → swap! - Swap `nums[1]` and `nums[0]`: - `nums` becomes `[1, -1, 3, 4]`

Now `i = 1`, `nums[1] = -1` → invalid → move on.

Step 3: `i = 2`

- `nums[2] = 3`
- Valid?
- `nums[3 - 1] = nums[2] = 3` → already correct!
- No swap. Move on.

Step 4: `i = 3`

- `nums[3] = 4`
- Valid?
- `nums[4 - 1] = nums[3] = 4` → correct!

Array after cyclic sort: `[1, -1, 3, 4]`

Step 5: Scan for first mismatch

- `i = 0`: `nums[0] = 1` → should be `1`
- `i = 1`: `nums[1] = -1` → should be `2` → **return 2**

Final output: `2`

Complexity Analysis

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

  Each element is swapped at most once into its correct place.
  The while loop may look nested, but total swaps `n`.
  Final scan is `O(n)`. So overall linear.

- **Space Complexity**: `O(1)`

  We modify the input array in place.
  Only use a few extra variables (`n`, `i`, `correct_idx`).
  No additional data structures that scale with input.