76. Minimum Window Substring

Solved

Hard | Topics | Companies | Hint

Given two strings `s` and `t` of lengths `m` and `n` respectively, return *the* ***minimum window*** ***substring*** *of* `s` *such that every character in* `t` ***(including duplicates)*** *is included in the window*. If there is no such substring, return *the empty string* `""`.

The testcases will be generated such that the answer is **unique**.

Example 1:

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
```

Example 2:

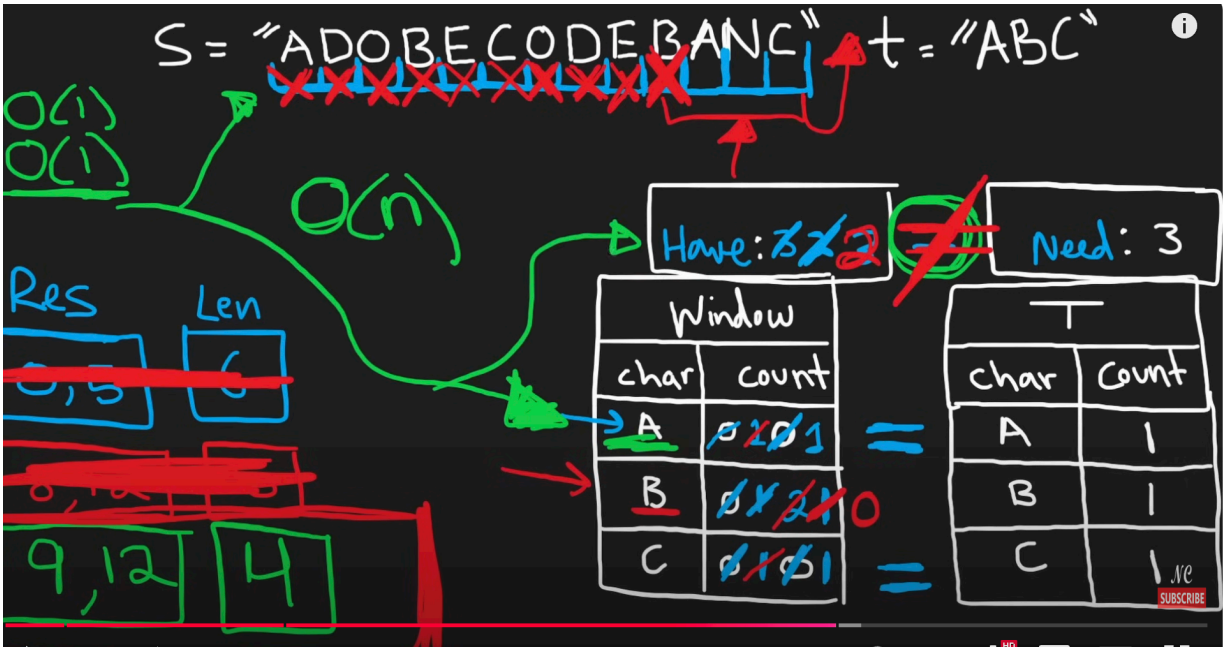
```
Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
```

Example 3:

```
Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
Since the largest window of s only has one 'a', return empty string.
```

Constraints:

- `m == s.length`
- `n == t.length`
- `1 <= m, n <= 10^5`
- `s` and `t` consist of uppercase and lowercase English letters.



In [33]:

```python
from collections import Counter
from math import inf

class Solution:
    def minWindow(self, source: str, target: str) -> str:
        # Create a counter for the target to keep a record of each character's frequency
        target_counter = Counter(target)
        window_counter = Counter()  # This will keep a count of characters in the current window
        valid_char_count = 0        # Number of characters that meet the target criteria
        left = 0                    # Left pointer to shrink the window
        min_left = -1               # Left boundary index of the minimum window
        min_size = inf              # Initialize min_size to positive infinity

        # Iterate over each character in the source string
        for right, char in enumerate(source):
            # Include current character in the window
            window_counter[char] += 1
            #print("window_counter",window_counter, "target_counter",target_counter)

            #print(target_counter[char],window_counter[char])

            # If the current character is needed and the window contains enough of this character
            if target_counter[char] >= window_counter[char]:
                valid_char_count += 1

                #print("char",char,"valid_char_count",valid_char_count)

            # If the window has all the characters needed
            while valid_char_count == len(target):
                # If this window is smaller than the minimum so far, update minimum size and index
                if right - left + 1 < min_size:
                    min_size = right - left + 1
                    min_left = left
```

```
            # If the character at the left pointer is less frequent in the window than in the target,
            # reducing it further would break the window condition
            if target_counter[source[left]] >= window_counter[source[left]]:
                valid_char_count -= 1
                #print("char",source[left],"valid_char_count",valid_char_count)

            # Shrink the window from the left
            window_counter[source[left]] -= 1
            left += 1

        # If no window meets the criteria, return an empty string
        return '' if min_left < 0 else source[min_left:min_left + min_size]
```

In [34]:
```
s=Solution()
s.minWindow("XDOBECODEBANC", "ABCB")
s.minWindow("a", "aa")
```

Out[34]:
```
''
```

---

42. Trapping Rain Water

Solved

Hard | Topics | Companies

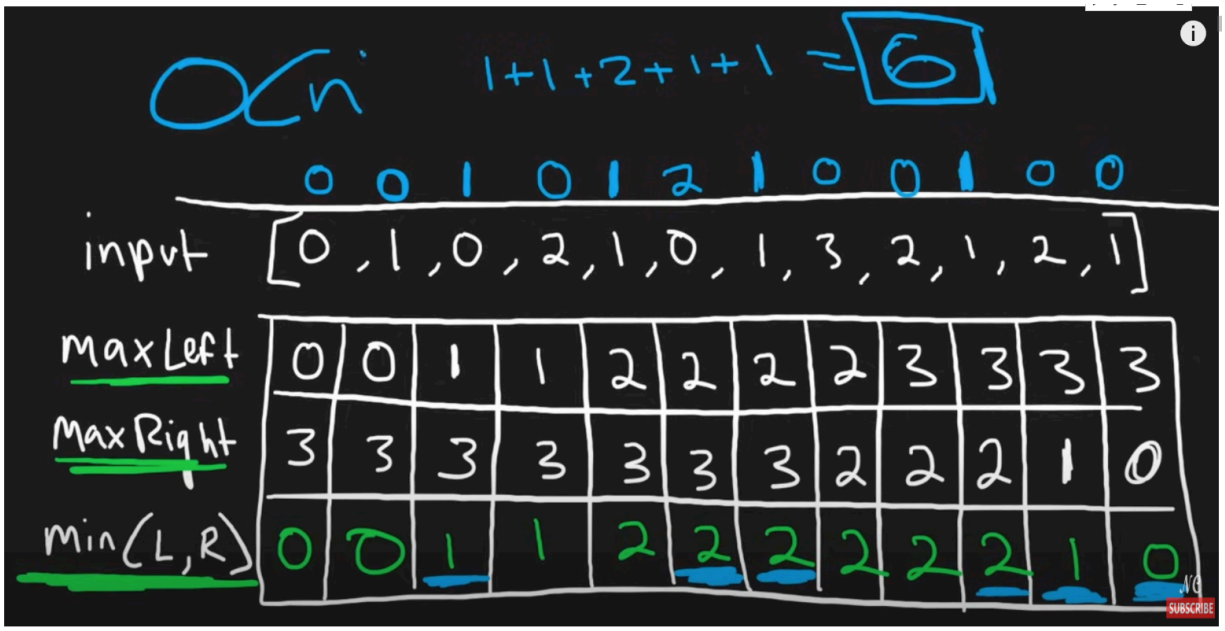
Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain
water (blue section) are being trapped.
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```



At location i will have min(maxleft,maxRight) - h[i] if min(maxleft,maxRight) - h[i]>0

In [56]:
```
from itertools import accumulate
from typing import List
class Solution:
    def trap(self, height: List[int]) -> int:
        maxl = [0] * len(height)
        maxr = [0] * len(height)
        minlr = [0] * len(height)

        maxln = 0
        maxl[0] = 0
        for i in range(1, len(height), 1):
            maxln = max(maxln, height[i - 1])
            maxl[i] = maxln
```

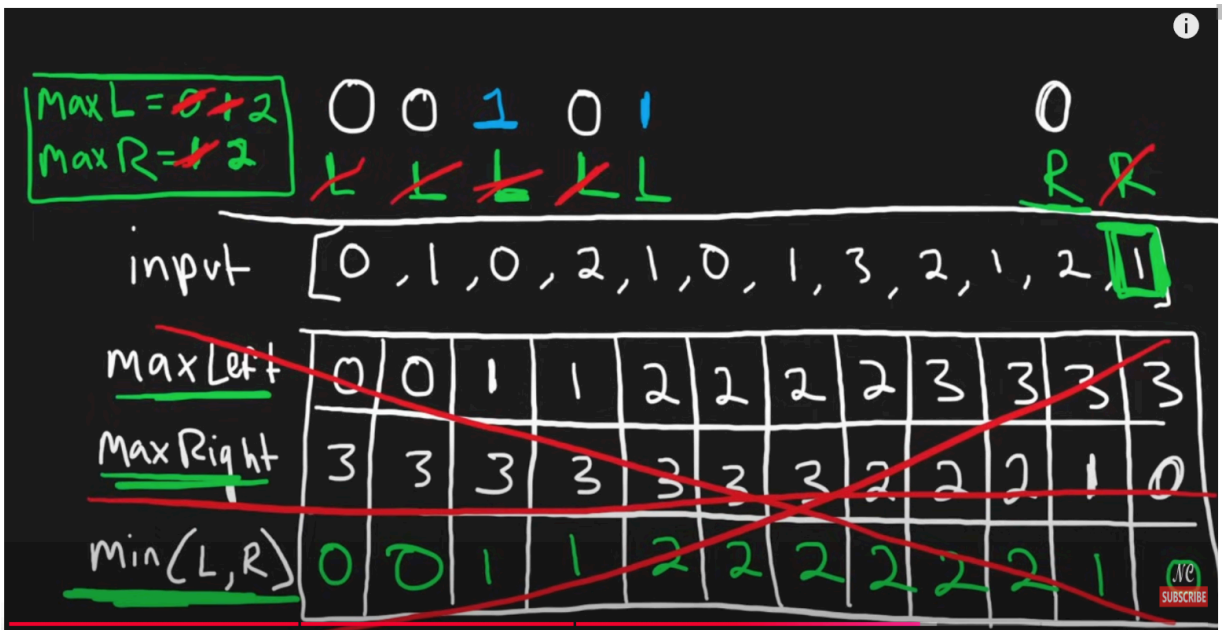
```
        maxrn = 0
        for i in range(len(height) - 2, -1, -1):
            maxrn = max(maxrn, height[i + 1])
            maxr[i] = maxrn

        res = 0
        for i in range(len(height)):
            temp = min(maxl[i], maxr[i]) - height[i]
            if temp > 0:
                res += temp

        return res
```

In [57]:
```
height = [0,1,0,2,1,0,1,3,2,1,2,1]
call=Solution()
call.trap(height)
```

Out[57]: 6

2nd approch with 2 pointer

we can move left pointer without knowing rightMAX bcz it need min(leftMAX,rightMAX) same happen from right side



In [86]:
```
from itertools import accumulate
from typing import List
class Solution:
    def trap(self, height: List[int]) -> int:
        if not height:
            return 0
        maxl = height[0]
        maxr = height[-1]
        left=0
        right=len(height)-1
        result=0

        while left<right:
            if maxl<=maxr:
                left+=1
                maxl=max(maxl,height[left])
                result+= maxl - height[left]
            else:
                right-=1
                maxr=max(maxr,height[right])
                result+= maxr - height[right]
        return result
```

In [87]:
```
height = [0,1,0,2,1,0,1,3,2,1,2,1]
call=Solution()
call.trap(height)
```

Out[87]: 6

---

4. Median of Two Sorted Arrays

Hard | Topics | Companies

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be `O(log (m+n))`.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Constraints:

- `nums1.length == m`
- `nums2.length == n`
- `0 <= m <= 1000`
- `0 <= n <= 1000`
- `1 <= m + n <= 2000`
- `-10^6 <= nums1[i], nums2[i] <= 10^6`

Steps

1. **Partition the Arrays:**
   - Divide `nums1` into two parts: `left1` and `right1` (similarly for `nums2`: `left2` and `right2`).
   - Ensure the partition satisfies the median condition:
     $$\max(\text{left1}, \text{left2}) \leq \min(\text{right1}, \text{right2})$$
   - The size of the left parts must equal half the combined length of both arrays.
2. **Binary Search on the Smaller Array:**
   - Use binary search to find the correct partition on the smaller array (say `nums1`).
   - Adjust the partition index based on comparisons of `max(left1)` and `min(right2)`.
3. **Median Calculation:**
   - If the combined size is even:
     $$\text{median} = \frac{\max(\text{left1}, \text{left2}) + \min(\text{right1}, \text{right2})}{2}$$
   - If odd:
     $$\text{median} = \max(\text{left1}, \text{left2})$$

In [5]:
```python
def findMedianSortedArrays(nums1, nums2):
    # Ensure nums1 is the smaller array
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    total_length = m + n
    half_length = total_length // 2

    left, right = 0, m
    while left <= right:
        partition1 = (left + right) // 2
        partition2 = half_length - partition1

        # Handle edge cases where partitions are out of bounds
        max_left1 = nums1[partition1 - 1] if partition1 > 0 else float('-inf')
        min_right1 = nums1[partition1] if partition1 < m else float('inf')

        max_left2 = nums2[partition2 - 1] if partition2 > 0 else float('-inf')
        min_right2 = nums2[partition2] if partition2 < n else float('inf')

        # Check if we have found the correct partition
```

```
        if max_left1 <= min_right2 and max_left2 <= min_right1:
            if total_length % 2 == 0:
                return (max(max_left1, max_left2) + min(min_right1, min_right2)) / 2
            else:
                return max(max_left1, max_left2)
        elif max_left1 > min_right2:
            # Move partition1 to the left
            right = partition1 - 1
        else:
            # Move partition1 to the right
            left = partition1 + 1

    raise ValueError("Input arrays are not sorted!")
```

Example

Input arrays:

- `nums1 = [1, 3, 8]`
- `nums2 = [7, 9, 10, 11]`

Step 1: Initialization

- Ensure `nums1` is the smaller array. Here, `nums1` is already smaller, so no swap is needed.
- Let:
  - `m = len(nums1) = 3`
  - `n = len(nums2) = 4`
- Total length:

$$\text{total\_length} = m + n = 7$$

  - `half_length = total_length // 2 = 3`

Set binary search bounds:

- `left = 0`, `right = 3` (size of `nums1`)

Step 2: Iteration 1

1. Compute partition indices:

$$\text{partition1} = (left + right)//2 = (0 + 3)//2 = 1$$

$$\text{partition2} = half\_length - partition1 = 3 - 1 = 2$$

2. Compute values at the partitions:
   - For `nums1`:

$$\text{max\_left1} = nums1[partition1 - 1] = nums1[0] = 1$$

$$\text{min\_right1} = nums1[partition1] = nums1[1] = 3$$

   - For `nums2`:

$$\text{max\_left2} = nums2[partition2 - 1] = nums2[1] = 9$$

$$\text{min\_right2} = nums2[partition2] = nums2[2] = 10$$

3. Check partition conditions:
   - Condition:

$$\text{max\_left1} \leq \text{min\_right2} \text{ and } \text{max\_left2} \leq \text{min\_right1}$$

   - Substituting values:

$$1 \leq 10 \, (\text{True}), \; 9 \leq 3 \, (\text{False})$$

Since `max_left2 > min_right1`, **move partition1 to the right**:

$$\text{left} = partition1 + 1 = 2$$

Step 3: Iteration 2

1. Compute partition indices:

$$\text{partition1} = (left + right)//2 = (2 + 3)//2 = 2$$

$$\text{partition2} = half\_length - partition1 = 3 - 2 = 1$$

2. Compute values at the partitions:
   - For `nums1`:

$$\text{max\_left1} = nums1[partition1 - 1] = nums1[1] = 3$$

$$\text{min\_right1} = nums1[partition1] = nums1[2] = 8$$

   - For `nums2`:

$$\text{max\_left2} = nums2[partition2 - 1] = nums2[0] = 7$$

$$\text{min\_right2} = nums2[partition2] = nums2[1] = 9$$

3. Check partition conditions:
   - Condition:

$$\text{max\_left1} \leq \text{min\_right2} \text{ and } \text{max\_left2} \leq \text{min\_right1}$$

   - Substituting values:

$$3 \leq 9 \, (\text{True}), \; 7 \leq 8 \, (\text{True})$$

Since the condition is satisfied, the correct partition is found.

Step 4: Compute the Median

- Total length is odd (`7`), so the median is:

$$\text{median} = \max(\text{max\_left1}, \text{max\_left2}) = \max(3, 7) = 7$$

---

23. Merge k Sorted Lists

Hard | Topics | Companies

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

Constraints:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` is sorted in **ascending order**.

In [20]:
```python
from typing import List, Optional
from queue import PriorityQueue

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    def __lt__(self, other):
        return self.val < other.val

class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        pq = PriorityQueue()

        # Add the head of each list to the priority queue
        for i, head in enumerate(lists):
            if head:
                pq.put((head.val, i, head))

        # Dummy node to simplify the merging process
        dummy_node = current = ListNode()

        # Process the priority queue
        while not pq.empty():
            _, i, node = pq.get()

            # Add the smallest node to the merged list
            current.next = node
            current = current.next

            # If there's a next node, add it to the priority queue
            if node.next:
                pq.put((node.next.val, i, node.next))

        # Return the head of the merged list
        return dummy_node.next
```

In [21]:
```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def create_linked_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for val in arr[1:]:
```

```
        current.next = ListNode(val)
        current = current.next
    return head

def print_linked_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

lists = [
    create_linked_list([1, 4, 5]),
    create_linked_list([1, 3, 4]),
    create_linked_list([2, 6])
]

s=Solution()

merged_head = s.mergeKLists(lists)
```

In [22]:
```
print_linked_list(merged_head)
```

Out[22]:
```
[1, 1, 2, 3, 4, 4, 5, 6]
```

---

239. Sliding Window Maximum

Hard | Topics | Companies | Hint

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Constraints:

- `1 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`
- `1 <= k <= nums.length`

Explanation of Changes

1. **Deque Operations**:
   - You maintain the deque in a decreasing order of values. This ensures that the largest element in the current window is always at the front (`q[0]`).
   - The condition `while (q and nums[right] > nums[q[-1]])` removes all indices from the back of the deque that have values smaller than `nums[right]`.
2. **Out-of-Bounds Check**:
   - The line `if left > q[0]: q.popleft()` ensures that indices no longer in the current window are removed from the deque.
3. **Collecting Maximums**:
   - The condition `(right + 1) >= k` ensures you start appending the maximum to the `output` list only after processing the first `k` elements.
4. **Increment Left**:
   - After adding the maximum to the output, you increment the `left` pointer to move the sliding window forward.

In [23]:
```python
from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        output = []
        q = deque()  # To store indices of elements in the current window
        left, right = 0, 0

        while right < len(nums):
            # Remove indices from the back of the deque if the current number is larger
            while q and nums[right] > nums[q[-1]]:
                q.pop()

            # Add the current index to the deque
            q.append(right)

            # Remove the leftmost index if it is out of the bounds of the sliding window
            if left > q[0]:
                q.popleft()

            # Start recording the maximum value once the first `k` elements have been processed
            if (right + 1) >= k:
                output.append(nums[q[0]])  # The max element in the current window
                left += 1  # Move the left boundary of the window

            # Move the right boundary of the window
            right += 1

        return output
```

In [24]:
```python
# Test case 1
nums = [1,3,-1,-3,5,3,6,7]
k = 3
print(Solution().maxSlidingWindow(nums, k))  # Output: [3, 3, 5, 5, 6, 7]

# Test case 2
nums = [1]
k = 1
print(Solution().maxSlidingWindow(nums, k))  # Output: [1]

# Test case 3
nums = [9,11]
k = 2
print(Solution().maxSlidingWindow(nums, k))  # Output: [11]

# Test case 4
nums = [4,3,2,1]
k = 2
print(Solution().maxSlidingWindow(nums, k))  # Output: [4, 3, 2]
```

```
[3, 3, 5, 5, 6, 7]
[1]
[11]
[4, 3, 2]
```

```python
nums = [1,3,-1,-3,5,3,6,7]
k = 3
```

Execution:

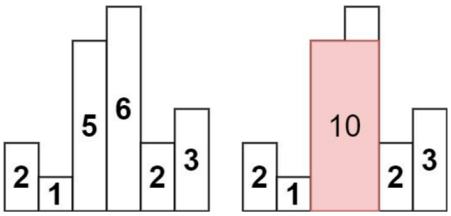
1. **Window** `[1, 3, -1]`:
   - Deque after processing: `[3]`
   - Maximum: `3`
2. **Window** `[3, -1, -3]`:
   - Deque after processing: `[3]`
   - Maximum: `3`
3. **Window** `[-1, -3, 5]`:
   - Deque after processing: `[5]`
   - Maximum: `5`
4. **Window** `[-3, 5, 3]`:
   - Deque after processing: `[5]`
   - Maximum: `5`
5. **Window** `[5, 3, 6]`:
   - Deque after processing: `[6]`
   - Maximum: `6`
6. **Window** `[3, 6, 7]`:
   - Deque after processing: `[7]`
   - Maximum: `7`

84. Largest Rectangle in Histogram

Hard · Topics · Companies

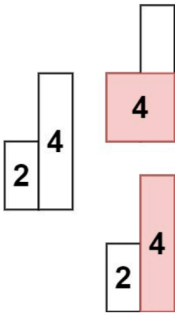
Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram.*

Example 1:

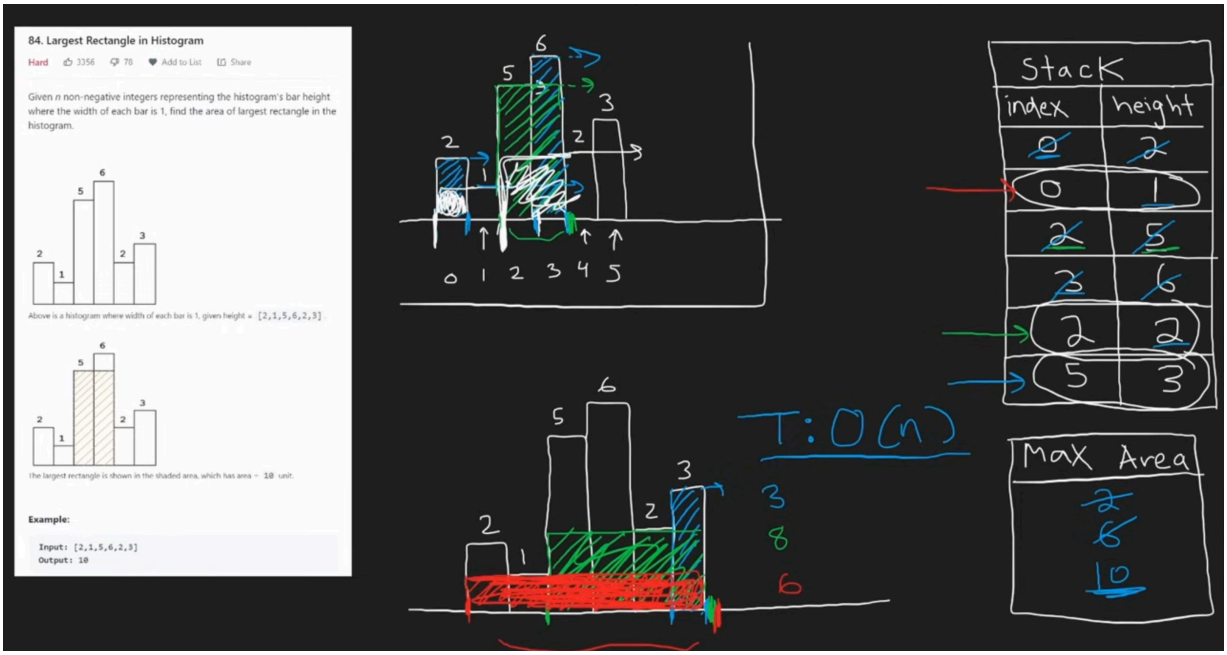



```
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.
```

Example 2:




```
Input: heights = [2,4]
Output: 4
```



In [25]:

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        # Initialize a stack to keep track of (index, height) for histogram bars
        stack = []

        # Variable to keep track of the maximum area found so far
        MaxArea = 0

        # Iterate through the histogram bars by index and height
        for i, h in enumerate(heights):
            start = i  # This variable tracks the leftmost index where the current height could extend

            # If the current height is less than the height of the bar at the top of the stack,
            # pop the stack and calculate the area of the rectangle with the popped height.
            while stack and stack[-1][1] > h:
                index, height = stack.pop()
                # Calculate area with height as the shortest bar and update MaxArea
                MaxArea = max(MaxArea, height * (i - index))
                # Update the start index to the index of the popped bar
                start = index

            # Push the current bar (start index, height) onto the stack
```

```
            stack.append((start, h))

        # Process remaining bars in the stack
        for i, h in stack:
            # Calculate the area for bars extending to the end of the histogram
            MaxArea = max(MaxArea, h * (len(heights) - i))

        # Return the maximum area found
        return MaxArea
```

In [26]:
```
heights = [2, 1, 5, 6, 2, 3]
sol = Solution()
print(sol.largestRectangleArea(heights))
```

```
10
```

---

41. First Missing Positive

Hard | Topics | Companies | Hint

Given an unsorted integer array `nums`. Return the *smallest positive integer* that is *not present* in `nums`.

You must implement an algorithm that runs in `O(n)` time and uses `O(1)` auxiliary space.

Example 1:

```
Input: nums = [1,2,0]
Output: 3
Explanation: The numbers in the range [1,2] are all in the array.
```

Example 2:

```
Input: nums = [3,4,-1,1]
Output: 2
Explanation: 1 is in the array but 2 is missing.
```

Example 3:

```
Input: nums = [7,8,9,11,12]
Output: 1
Explanation: The smallest positive integer 1 is missing.
```

Constraints:

- `1 <= nums.length <= 10^5`
- `-2^31 <= nums[i] <= 2^31 - 1`

Solution Approach

To solve this efficiently:

1. **Focus on Positive Integers**: Ignore negative numbers and numbers greater than the length of the array (`n`).
2. **Use Index as a Hash**: Place each positive integer `x` at its correct index `(x - 1)` in the array.
3. **Find the Missing Positive**: After rearranging, the first index `i` where `nums[i] != i + 1` gives the missing positive number `i + 1`.

Algorithm

1. Iterate through the array and swap numbers to their correct positions.
2. After rearranging, iterate again to find the first index where the condition fails.
3. If all indices satisfy the condition, return `n + 1`.

In [27]:
```python
def first_missing_positive(nums):
    n = len(nums)

    # Place each number in its correct position if possible
    for i in range(n):
        while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            # Swap nums[i] with the number at its correct position
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    # Find the first missing positive
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1

    # If all numbers are in their correct positions
    return n + 1

# Example Usage
print(first_missing_positive([1, 2, 0]))        # Output: 3
print(first_missing_positive([3, 4, -1, 1]))   # Output: 2
print(first_missing_positive([7, 8, 9, 11, 12])) # Output: 1
```

```
3
2
1
```

Explanation

1. For `[3, 4, -1, 1]`:
   - Rearrange: Place `3` at index `2`, `4` at index `3`, `1` at index `0`. Result: `[1, -1, 3, 4]`.
   - Scan: The first missing positive is `2` (index `1` is incorrect).
2. For `[7, 8, 9, 11, 12]`:
   - None of the numbers fit within the range `[1, n]`.
   - Missing positive is `1`.

Time Complexity

- Rearranging: **O(n)** (each number is swapped at most once).
- Scanning: **O(n)**.
- Overall: **O(n)**.

Space Complexity

- Uses the input array for in-place rearrangement: **O(1)**.

---

124. Binary Tree Maximum Path Sum

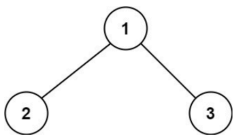
Hard · Topics · Companies

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

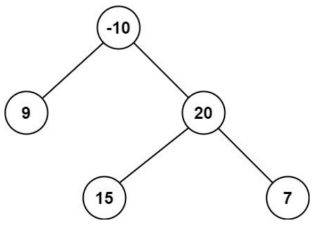
Given the `root` of a binary tree, return *the maximum **path sum** of any **non-empty** path*.

Example 1:




```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:




```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
```

Constraints:

- The number of nodes in the tree is in the range $[1, 3 * 10^4]$.

To solve this:

1. **Understand the Path Sum**:
   - A path can go:
     - Through a subtree.
     - From one child to the parent and into the other child.
   - The maximum path sum is a combination of these possibilities.
2. **Recursive DFS Approach**:
   - Compute the maximum path sum for each subtree.
   - Use a helper function to return the **maximum gain** from a node to its children.
   - Update the global maximum path sum during the traversal.
3. **Key Insight**:
   - At each node, decide whether to:
     - Include the node's value only.
     - Add one or both of its children.

Algorithm

1. Traverse the tree using DFS.
2. For each node:
   - Compute the maximum gain from the left and right subtrees.
   - Update the global maximum with the sum of:
     - The node's value.
     - Maximum gain from the left and right children (if any).
3. Return the maximum path sum.

In [28]:
```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
```

```
        self.right = right

def max_path_sum(root):
    global_max = float('-inf')  # Initialize global maximum

    def dfs(node):
        nonlocal global_max
        if not node:
            return 0

        # Compute the maximum path sum through left and right subtrees
        left_max = max(dfs(node.left), 0)  # Ignore negative paths
        right_max = max(dfs(node.right), 0)

        # Update global maximum path sum
        current_max = node.val + left_max + right_max
        global_max = max(global_max, current_max)

        # Return the maximum gain to parent
        return node.val + max(left_max, right_max)

    dfs(root)
    return global_max

# Example Usage
# Tree:      -10
#           /  \
#          9   20
#             /  \
#            15   7
root = TreeNode(-10)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))

print(max_path_sum(root))  # Output: 42
```

42

![Screenshot 2024-12-23 at 1.02.20 PM.png](<attachment:Screenshot 2024-12-23 at 1.02.20 PM.png>)

---

72. Edit Distance

Medium · Topics · Companies

Given two strings `word1` and `word2`, return *the minimum number of operations required to convert* `word1` *to* `word2`.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

Constraints:

- `0 <= word1.length, word2.length <= 500`
- `word1` and `word2` consist of lowercase English letters.

Explanation of the Code

1. **Initialization**:
   - The table `dp` is initialized with `float('inf')`, which is a way to indicate that the cells are unfilled and need to be computed.
   - The base case for the last row and column is set as follows:
     - For `dp[len(word1)][j]`: The number of operations required to convert `word1` into an empty string (after `i` deletions) is `len(word2) - j`, which is the number of insertions needed to convert the rest of `word1` into `word2`.
     - Similarly, for `dp[i][len(word2)]`: The number of operations to convert the remaining characters of `word1` into an empty string is `len(word1) - i`.
2. **Filling the DP Table**:
   - The table is filled starting from the **bottom-right** corner and working our way up to the **top-left** corner.
   - If the characters `word1[i]` and `word2[j]` match, no operation is needed, so `dp[i][j] = dp[i+1][j+1]` (carry over the previous result).
   - If the characters don't match, the minimum of the three operations (insert, delete, replace) is computed:
     - **Insert**: `dp[i][j+1]`
     - **Delete**: `dp[i+1][j]`
     - **Replace**: `dp[i+1][j+1]`
     - Add `1` to account for the operation.
3. **Returning the Result**:
   - The final result is found at `dp[0][0]`, which gives the minimum number of operations required to convert `word1` to `word2`.

In [29]:
```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        # Initialize the DP table with 'inf' (to simulate uninitialized cells)
        dp = [[float('inf')] * (len(word2) + 1) for _ in range(len(word1) + 1)]

        # Base case for the last row (conversion from word1 to an empty string)
        for j in range(len(word2) + 1):
            dp[len(word1)][j] = len(word2) - j

        # Base case for the last column (conversion from an empty string to word1)
        for i in range(len(word1) + 1):
            dp[i][len(word2)] = len(word1) - i

        # Fill the DP table by iterating from bottom-right to top-left
        for i in range(len(word1) - 1, -1, -1):
            for j in range(len(word2) - 1, -1, -1):
                if word1[i] == word2[j]:
                    dp[i][j] = dp[i + 1][j + 1]  # No operation needed if characters match
                else:
                    dp[i][j] = 1 + min(dp[i + 1][j], dp[i][j + 1], dp[i + 1][j + 1])  # Min of insert, delete, replace

        return dp[0][0]  # The final result is at the top-left corner
```

Step-by-Step Explanation

- **Initialization of the DP Table**:
  - The table `dp` is initialized to `inf`, which represents uncomputed values.
  - The base cases are computed by filling the last row and column:
    - The last row represents the conversion of `word1` into an empty string.
    - The last column represents the conversion of an empty string into `word2`.
- **Filling the DP Table**:
  - Start from the bottom-right corner of the DP table and work your way to the top-left.
  - For each pair of characters `word1[i]` and `word2[j]`:
    - If they match, no operation is needed (`dp[i][j] = dp[i + 1][j + 1]`).
    - If they don't match, compute the minimum of the three possible operations (insert, delete, replace) and add 1 to account for the operation performed.
- **Final Answer**:
  - The minimum number of operations required to convert `word1` to `word2` is stored in `dp[0][0]`.

Time and Space Complexity

- **Time Complexity**: **O(m * n)**
  Where `m` and `n` are the lengths of `word1` and `word2`, respectively. We fill the DP table of size `(m+1) x (n+1)`, and each cell computation takes constant time.
- **Space Complexity**: **O(m * n)**
  The space complexity is determined by the size of the DP table, which is `(m+1) x (n+1)`.

---

10. Regular Expression Matching

Hard | Topics | Companies

Given an input string `s` and a pattern `p`, implement regular expression matching with support for `'.'` and `'*'` where:

- `'.'` Matches any single character.
- `'*'` Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input: s = "aa", p = "a*"
Output: true
Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".
```

Example 3:

```
Input: s = "ab", p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".
```

Constraints:

- `1 <= s.length <= 20`
- `1 <= p.length <= 20`
- `s` contains only lowercase English letters.
- `p` contains only lowercase English letters, `'.'`, and `'*'`.
- It is guaranteed for each appearance of the character `'*'`, there will be a previous valid character to match.

How It Works

1. **Base Case**:
   - If the pattern `p` is exhausted (`j == len(p)`), return `True` only if the string `s` is also exhausted (`i == len(s)`).
2. **First Match**:
   - Check if the first character of the remaining `s` and `p` match. This is `True` if:
     - `i < len(s)` (there are characters left in `s`), and
     - Either the current character in `p` is a dot (`p[j] == '.'`) or it matches the current character in `s` (`s[i] == p[j]`).
3. **Handling `*`**:
   - If the next character in the pattern (`p[j+1]`) is `*`, there are two cases:
     - **Zero occurrences**: Skip the `"char*"` part of the pattern (`match(i, j + 2)`).
     - **One or more occurrences**: Match the current character in `s` with the pattern and continue matching (`first_match and match(i + 1, j)`).
4. **No `*`**:
   - If there is no `*`, simply proceed to the next characters in both `s` and `p` if `first_match` is `True`.

In [30]:
```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        def match(i: int, j: int) -> bool:
            # Base case: If we've processed the entire pattern
            if j == len(p):
                return i == len(s)

            # Check if the first character of s and p match
            first_match = i < len(s) and (s[i] == p[j] or p[j] == '.')

            # Handle '*' in the pattern
            if j + 1 < len(p) and p[j + 1] == '*':
                # Two options:
                # 1. Skip the "char*" in the pattern (match zero occurrences)
                # 2. Use the '*' to match one or more of the current char in s
                return (match(i, j + 2) or
                        (first_match and match(i + 1, j)))
            else:
                # If there's no '*', move both pointers if the characters match
                return first_match and match(i + 1, j + 1)

        # Start the recursion from the beginning of both strings
        return match(0, 0)
```

In [31]:
```
s = "aab"
p = "c*a*b"
solution = Solution()
print(solution.isMatch(s, p))  # Output: True
```

```
True
```

Optimizing with Memoization To optimize this recursive approach, we can use memoization to store results for overlapping subproblems.

In [33]:
```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        memo = {}

        def match(i: int, j: int) -> bool:
            if (i, j) in memo:
                return memo[(i, j)]

            if j == len(p):
                result = i == len(s)
            else:
                first_match = i < len(s) and (s[i] == p[j] or p[j] == '.')
                if j + 1 < len(p) and p[j + 1] == '*':
                    result = (match(i, j + 2) or
                              (first_match and match(i + 1, j)))
                else:
                    result = first_match and match(i + 1, j + 1)

            memo[(i, j)] = result
            return result
```

```
        return match(0, 0)
```

In [34]:
```
s = "aab"
p = "c*a*b"
solution = Solution()
print(solution.isMatch(s, p))  # Output: True
```

```
True
```

Dynamic Programming Approach

We define a 2D DP table, where `dp[i][j]` is `True` if the first `i` characters of `s` match the first `j` characters of `p`.

Algorithm

1. **DP Table Initialization**:
   - Create a DP table `dp` of size `(len(s) + 1) x (len(p) + 1)` and initialize all values to `False`.
   - `dp[0][0]` is `True` because an empty string matches an empty pattern.
2. **Base Case for Patterns with `*`**:
   - If the pattern contains `*`, it can match zero occurrences of the preceding character. This means `dp[0][j]` is `True` if `p[j-1]` is `*` and `dp[0][j-2]` is `True`.
3. **Fill the DP Table**:
   - For each character in `s` and `p`:
     - If the current characters match (`p[j-1] == s[i-1]` or `p[j-1] == '.'`), then `dp[i][j] = dp[i-1][j-1]`.
     - If the current character in `p` is `*`, there are two possibilities:
       - The `*` represents zero occurrences: `dp[i][j] = dp[i][j-2]`.
       - The `*` represents one or more occurrences: `dp[i][j] = dp[i-1][j]` if the preceding character in `p` matches the current character in `s`.
4. **Result**:
   - The final result is stored in `dp[len(s)][len(p)]`.

In [36]:
```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)

        # Initialize the DP table
        dp = [[False] * (n + 1) for _ in range(m + 1)]

        # Base case: empty string matches empty pattern
        dp[0][0] = True

        # Handle patterns with '*' that match an empty string
        for j in range(2, n + 1):
            if p[j - 1] == '*':
                dp[0][j] = dp[0][j - 2]

        # Fill the DP table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if p[j - 1] == s[i - 1] or p[j - 1] == '.':
                    # Characters match or '.' matches any character
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == '*':
                    # '*' matches zero or more of the preceding character
                    dp[i][j] = dp[i][j - 2]  # Zero occurrences
                    if p[j - 2] == s[i - 1] or p[j - 2] == '.':
                        dp[i][j] = dp[i][j] or dp[i - 1][j]  # One or more occurrences

        # Final result
        return dp[m][n]
```

In [37]:
```
s = "aab"
p = "c*a*b"
solution = Solution()
print(solution.isMatch(s, p))  # Output: True
```

```
True
```

---

32. Longest Valid Parentheses

Hard | Topics | Companies

Given a string containing just the characters `'('` and `')'`, return *the length of the longest valid (well-formed) parentheses substring*.

Example 1:

```
Input: s = "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()".
```

Example 2:

```
Input: s = ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()".
```

Example 3:

```
Input: s = ""
Output: 0
```

Constraints:

- `0 <= s.length <= 3 * 10^4`
- `s[i]` is `'('`, or `')'`.

Approaches

Approach 1: Using a Stack

We can use a stack to keep track of indices of unmatched parentheses. This helps us compute the lengths of valid substrings.

Algorithm:

1. Use a stack to store indices.
2. Push `-1` into the stack as a base for valid substrings.
3. For each character:
   - If it's `'('`, push its index onto the stack.
   - If it's `')'`:
     - Pop the top of the stack.
     - If the stack is empty, push the current index as a base.
     - Otherwise, calculate the length of the valid substring using the current index and the top of the stack.
4. Return the maximum length found.

In [38]:
```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        stack = [-1]  # Initialize stack with a base index
        max_length = 0

        for i, char in enumerate(s):
            if char == '(':
                stack.append(i)  # Push the index of '(' onto the stack
            else:
                stack.pop()  # Pop the top of the stack
                if not stack:
                    stack.append(i)  # If the stack is empty, push the current index
```

```
            else:
                # Calculate the length of the valid substring
                max_length = max(max_length, i - stack[-1])

    return max_length
```

Input String: `")()())"`

Initialization:

- `stack = [-1]` (base index).
- `max_length = 0`.

Iteration:

1. **Index 0, Character** `')'`:
    - Pop `-1` from the stack → `stack = []`.
    - Stack is empty, push the current index `0` → `stack = [0]`.
2. **Index 1, Character** `'('`:
    - Push the index `1` → `stack = [0, 1]`.
3. **Index 2, Character** `')'`:
    - Pop the top of the stack (`1`) → `stack = [0]`.
    - Calculate length: `2 - stack[-1] = 2 - 0 = 2`.
    - Update `max_length = max(0, 2) = 2`.
4. **Index 3, Character** `'('`:
    - Push the index `3` → `stack = [0, 3]`.
5. **Index 4, Character** `')'`:
    - Pop the top of the stack (`3`) → `stack = [0]`.
    - Calculate length: `4 - stack[-1] = 4 - 0 = 4`.
    - Update `max_length = max(2, 4) = 4`.
6. **Index 5, Character** `')'`:
    - Pop the top of the stack (`0`) → `stack = []`.
    - Stack is empty, push the current index `5` → `stack = [5]`.

Approach 2: Two Pointers

We can use two pointers to traverse the string twice (once from left to right and once from right to left) to keep track of open and close parentheses.

Algorithm:

1. **Left to Right**:
   - Count `open` and `close` parentheses.
   - Whenever `open == close`, update the maximum length.
   - If `close > open`, reset the counters.
2. **Right to Left**:
   - Count `open` and `close` parentheses.
   - Whenever `open == close`, update the maximum length.
   - If `open > close`, reset the counters.

In [39]:
```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        max_length = 0
        open_count, close_count = 0, 0

        # Left to right
        for char in s:
            if char == '(':
                open_count += 1
            else:
                close_count += 1
            if open_count == close_count:
                max_length = max(max_length, 2 * close_count)
            elif close_count > open_count:
                open_count = close_count = 0

        # Reset counters for right to left
        open_count, close_count = 0, 0

        # Right to left
        for char in reversed(s):
            if char == '(':
                open_count += 1
            else:
                close_count += 1
            if open_count == close_count:
                max_length = max(max_length, 2 * open_count)
            elif open_count > close_count:
                open_count = close_count = 0

        return max_length
```

In [42]:
```python
s = "(()))())("
solution = Solution()
print(solution.longestValidParentheses(s))  # Output: 4
```

```
8
```

---

127. Word Ladder

Hard | Topics | Companies

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.
```

Constraints:

- `1 <= beginWord.length <= 10`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 5000`
- `wordList[i].length == beginWord.length`

Algorithm Steps

1. **Preprocessing**:
   - Create a dictionary where the key is a "generic pattern" of words (e.g., `h*t` for `hot`), and the value is a list of words matching the pattern.
2. **BFS Initialization**:
   - Start from `beginWord`, initialize a queue with the word and the level (distance from the start).
   - Use a set to mark visited words.
3. **BFS Traversal**:
   - For the current word, generate all possible generic patterns.
   - For each generic pattern, look up matching words in the preprocessed dictionary.
   - Add each matching word to the queue if it hasn't been visited.
4. **End Condition**:
   - If `endWord` is reached, return the current level.
5. **No Path**:
   - If the queue is empty and `endWord` was not reached, return `0`.

In [43]:

```python
from collections import deque, defaultdict

class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: list[str]) -> int:
        if endWord not in wordList:
            return 0

        # Preprocess wordList into a generic pattern dictionary
        wordList = set(wordList)  # Convert to set for faster lookup
        pattern_dict = defaultdict(list)

        for word in wordList:
            for i in range(len(word)):
                pattern = word[:i] + '*' + word[i+1:]
                pattern_dict[pattern].append(word)
```

```
        # BFS initialization
        queue = deque([(beginWord, 1)])  # (current_word, level)
        visited = set()  # Keep track of visited words
        visited.add(beginWord)

        while queue:
            current_word, level = queue.popleft()

            # Generate generic patterns for the current word
            for i in range(len(current_word)):
                pattern = current_word[:i] + '*' + current_word[i+1:]

                # Process all neighbors
                for neighbor in pattern_dict[pattern]:
                    if neighbor == endWord:
                        return level + 1

                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, level + 1))

                # Clear the processed pattern to reduce future lookups
                pattern_dict[pattern] = []

        return 0
```

In [44]:
```
beginWord = "hit"
endWord = "cog"
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

solution = Solution()
print(solution.ladderLength(beginWord, endWord, wordList))  # Output: 5
```

```
5
```

Execution Steps:

1. **Initialization**:
   - `pattern_dict`:

     ```arduino
     *ot: ["hot", "dot", "lot"]
     h*t: ["hot"]
     ho*: ["hot"]
     d*t: ["dot"]
     do*: ["dot", "dog"]
     *og: ["dog", "log", "cog"]
     l*t: ["lot"]
     lo*: ["lot", "log"]
     c*g: ["cog"]
     co*: ["cog"]
     ```

   - `queue`: `deque([("hit", 1)])`
   - `visited`: `{"hit"}`.
2. **BFS Iteration**:
   - Process `"hit"`, level `1`:
     - Patterns: `*it`, `h*t`, `hi*`.
     - Neighbor: `"hot"`.
     - Add `"hot"` to the queue: `deque([("hot", 2)])`.
   - Process `"hot"`, level `2`:
     - Patterns: `*ot`, `h*t`, `ho*`.
     - Neighbors: `"dot"`, `"lot"`.
     - Add `"dot"` and `"lot"` to the queue: `deque([("dot", 3), ("lot", 3)])`.
   - Process `"dot"`, level `3`:
     - Patterns: `*ot`, `d*t`, `do*`.
     - Neighbor: `"dog"`.
     - Add `"dog"` to the queue: `deque([("lot", 3), ("dog", 4)])`.
   - Process `"lot"`, level `3`:
     - Patterns: `*ot`, `l*t`, `lo*`.
     - Neighbor: `"log"`.
     - Add `"log"` to the queue: `deque([("dog", 4), ("log", 4)])`.
   - Process `"dog"`, level `4`:
     - Patterns: `*og`, `d*g`, `do*`.
     - Neighbor: `"cog"`.
     - Found `endWord`, return `5`.

---

295. Find Median from Data Stream

Hard | Topics | Companies

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is `3`.
- For example, for `arr = [2,3]`, the median is `(2 + 3) / 2 = 2.5`.

Implement the MedianFinder class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:

```
Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Constraints:

- -10^5 <= num <= 10^5
- There will be at least one element in the data structure before calling `findMedian`.
- At most $5 * 10^4$ calls will be made to `addNum` and `findMedian`.

Approach

We use two heaps:

1. **Max Heap** (`lower`): Stores the smaller half of the numbers.
2. **Min Heap** (`upper`): Stores the larger half of the numbers.

The key idea is to balance the heaps so that:

- The size difference between the two heaps is at most 1.
- The median can be easily computed:
  - If the heaps are of equal size, the median is the average of the max of the `lower` heap and the min of the `upper` heap.
  - If one heap has more elements, the median is the root of that heap.

Algorithm

1. **Adding a Number**:
   - Add the number to one of the heaps (ensuring the max heap always has the smaller half of the numbers).
   - Balance the heaps if needed (move elements between heaps to maintain the size property).
2. **Finding the Median**:
   - If the heaps have the same size, return the average of the max of `lower` and the min of `upper`.
   - If one heap is larger, return the root of that heap.

In [1]:
```python
import heapq

class MedianFinder:
```

```
    def __init__(self):
        # Max heap for the lower half (invert sign for max heap behavior)
        self.lower = []
        # Min heap for the upper half
        self.upper = []

    def addNum(self, num: int) -> None:
        # Add to max heap (invert sign to simulate max heap)
        heapq.heappush(self.lower, -num)

        # Ensure every number in `lower` is <= every number in `upper`
        if self.lower and self.upper and (-self.lower[0] > self.upper[0]):
            heapq.heappush(self.upper, -heapq.heappop(self.lower))

        # Balance the heaps so that the size difference is at most 1
        if len(self.lower) > len(self.upper) + 1:
            heapq.heappush(self.upper, -heapq.heappop(self.lower))
        elif len(self.upper) > len(self.lower):
            heapq.heappush(self.lower, -heapq.heappop(self.upper))

    def findMedian(self) -> float:
        # If heaps are of equal size, median is the average of the roots
        if len(self.lower) == len(self.upper):
            return (-self.lower[0] + self.upper[0]) / 2.0
        # Otherwise, median is the root of the larger heap
        return -self.lower[0] if len(self.lower) > len(self.upper) else self.upper[0]
```

In [2]:
```
medianFinder = MedianFinder()
medianFinder.addNum(1)
medianFinder.addNum(2)
print(medianFinder.findMedian())  # Output: 1.5
medianFinder.addNum(3)
print(medianFinder.findMedian())  # Output: 2
```

```
1.5
2
```

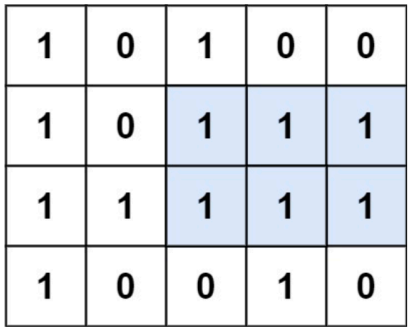
---

85. Maximal Rectangle

Hard · Topics · Companies

Given a `rows x cols` binary `matrix` filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return *its area*.

Example 1:



1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
```

Example 2:

```
Input: matrix = [["0"]]
Output: 0
```

Example 3:

```
Input: matrix = [["1"]]
Output: 1
```

Steps:

1. **Iterate through rows**:
   - Build a height array where each element represents the height of the histogram for the current column up to the current row.
2. **Apply Largest Rectangle in Histogram**:
   - Use the `largestRectangleArea` function from the previous problem for each row's height array.

In [4]:
```
from typing import List
class Solution:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        if not matrix or not matrix[0]:
            return 0  # Handle empty matrix case
```

```
        # Initialize variables
        rows, cols = len(matrix), len(matrix[0])
        heights = [0] * cols
        max_area = 0

        # Iterate through each row in the matrix
        for row in matrix:
            # Update the heights array based on the current row
            for col in range(cols):
                if row[col] == "1":
                    heights[col] += 1  # Increment height
                else:
                    heights[col] = 0  # Reset height

            # Calculate the largest rectangle area for the current histogram
            max_area = max(max_area, self.largestRectangleArea(heights))

        return max_area

    def largestRectangleArea(self, heights: List[int]) -> int:
        # Monotonic stack approach for Largest Rectangle in Histogram
        stack = []
        max_area = 0

        for i, h in enumerate(heights):
            start = i
            while stack and stack[-1][1] > h:
                index, height = stack.pop()
                max_area = max(max_area, height * (i - index))
                start = index
            stack.append((start, h))

        for i, h in stack:
            max_area = max(max_area, h * (len(heights) - i))

        return max_area
```

In [8]:
```
matrix = [
    ["1", "0", "1", "0", "0"],
    ["1", "0", "1", "1", "1"],
    ["1", "1", "1", "1", "1"],
    ["1", "0", "0", "1", "0"]
]
Solution().maximalRectangle(matrix)
```

Out[8]: 6

---

25. Reverse Nodes in k-Group

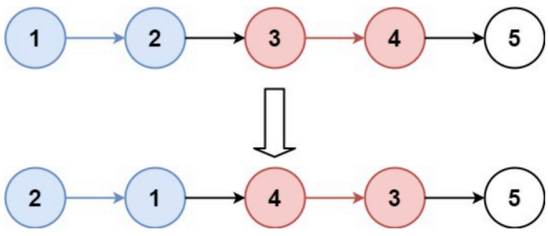
Hard · Topics · Companies

Given the `head` of a linked list, reverse the nodes of the list `k` at a time, and return *the modified list*.

`k` is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of `k` then left-out nodes, in the end, should remain as it is.

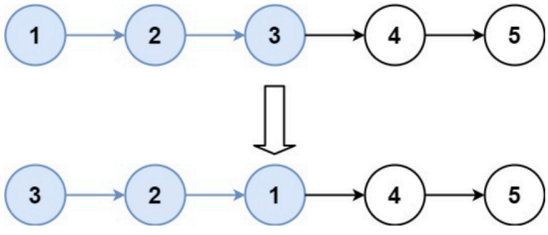
You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Example 2:




```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
```

In [12]:
```
from typing import Optional

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

```python
class Solution:
    def reverseKGroup(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        dummyNode = ListNode(0, head)  # Dummy node pointing to the head of the list
        groupPrev = dummyNode

        while True:
            # Find the kth node from the current position
            kth = self.getkth(groupPrev, k)
            if not kth:  # If there are fewer than k nodes left, stop
                break

            groupNext = kth.next  # Node after the kth node

            # Reverse the group of k nodes
            prev, curr = groupNext, groupPrev.next
            while curr != groupNext:
                temp = curr.next
                curr.next = prev
                prev = curr
                curr = temp

            # Update pointers to connect reversed group
            temp = groupPrev.next
            groupPrev.next = kth
            groupPrev = temp  # Move groupPrev to the end of the reversed group

        return dummyNode.next

    def getkth(self, curr, k):
        # Find the kth node from the current position
        while curr and k > 0:
            curr = curr.next
            k -= 1
        return curr
```

In [20]:

```python
# Helper function to create a linked list from a list of values
def create_linked_list(values):
    dummy = ListNode()
    curr = dummy
    for val in values:
        curr.next = ListNode(val)
        curr = curr.next
    return dummy.next

# Helper function to print a linked list
def print_linked_list(head):
    vals = []
    while head:
        vals.append(head.val)
        head = head.next
    print(" -> ".join(map(str, vals)))

# Example
solution = Solution()
head = create_linked_list([1, 2, 3, 4, 5,8,9,10])
k = 3
new_head = solution.reverseKGroup(head, k)
print_linked_list(new_head)
```

```
3 -> 2 -> 1 -> 8 -> 5 -> 4 -> 9 -> 10
```

---

312. Burst Balloons

Hard · Topics · Companies

You are given `n` balloons, indexed from `0` to `n - 1`. Each balloon is painted with a number on it represented by an array `nums`. You are asked to burst all the balloons.

If you burst the `i^th` balloon, you will get `nums[i - 1] * nums[i] * nums[i + 1]` coins. If `i - 1` or `i + 1` goes out of bounds of the array, then treat it as if there is a balloon with a `1` painted on it.

Return *the maximum coins you can collect by bursting the balloons wisely*.

Example 1:

```
Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8   +  1*3*8  + 1*8*1 = 167
```

Example 2:

```
Input: nums = [1,5]
Output: 10
```

Constraints:

- `n == nums.length`
- `1 <= n <= 300`
- `0 <= nums[i] <= 100`

Code Breakdown

1. Array Transformation

```python
nums = [1] + nums + [1]
```

- Adds two dummy balloons with value `1` to the start and end of the array `nums`.
- These dummy balloons ensure the boundaries of the array are handled uniformly when calculating coins for the bursting of inner balloons.

For example:

- Input: `nums = [3, 1, 5, 8]`
- Transformed: `nums = [1, 3, 1, 5, 8, 1]`

---

2. Recursive Function with Memoization

The main logic is implemented in the `dfs(left, right)` function.

Parameters

- `left`: The index of the left boundary of the current subarray being considered.
- `right`: The index of the right boundary of the current subarray being considered.

Base Case

```python
if left + 1 == right:
    return 0
```

- If the `left` and `right` indices are adjacent (`left + 1 == right`), there are no balloons to burst in this range, so the maximum coins for this subarray is `0`.

3. Memoization

```python
if (left, right) in memo:
    return memo[(left, right)]
```

- To avoid recomputing results for the same subarray `[left, right]`, the results are stored in a dictionary `memo`.
- If the result for `(left, right)` is already computed, return it directly.

---

4. Recursive Logic

```python
for k in range(left + 1, right):
```

- Iterate over all possible balloons `k` in the range `[left + 1, right - 1]`.
- The balloon at index `k` is assumed to be the **last balloon to burst** in this range. This splits the problem into:
  1. Bursting balloons in the subarray `[left, k]` (left of `k`).
  2. Bursting balloons in the subarray `[k, right]` (right of `k`).

---

5. Calculate Coins

```python
coins = nums[left] * nums[k] * nums[right]
coins += dfs(left, k) + dfs(k, right)
max_coins = max(max_coins, coins)
```

- The coins gained from bursting balloon `k` last are `nums[left] * nums[k] * nums[right]`.
- Add the results of recursively bursting balloons in the subarrays `[left, k]` and `[k, right]`.
- Keep track of the maximum coins collected for this range by updating `max_coins`.

---

6. Store in Memo

```python
memo[(left, right)] = max_coins
return max_coins
```

- Store the computed result for the range `(left, right)` in `memo` to avoid redundant computations in the future.

---

7. Final Call

```python
return dfs(0, n - 1)
```

- The result for the entire array is obtained by considering the range `[0, n - 1]`, which includes all balloons (including the dummy ones).

Example Walkthrough

Input:

plaintext Copy code

```
nums = [3, 1, 5, 8]
```

Transformed:

plaintext Copy code

```
nums = [1, 3, 1, 5, 8, 1]
```

Execution:

1. Start with the entire range `[0, 5]`.
2. Consider all possible `k` values for the last balloon to burst in this range (`k = 1, 2, 3, 4`).
   - For each `k`, recursively solve the left `[0, k]` and right `[k, 5]` subproblems.
   - Accumulate coins from `nums[0] * nums[k] * nums[5]` and add results from the subproblems.
3. Store results for each subrange in `memo` and return the maximum coins for `[0, 5]`.

Output:

plaintext Copy code

```
167
```

In [4]:
```python
from typing import List
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        nums = [1] + nums + [1]
        n = len(nums)
        memo = {}

        def dfs(left, right):
            if left + 1 == right:
                return 0
            if (left, right) in memo:
                return memo[(left, right)]

            max_coins = 0
            for k in range(left + 1, right):
                coins = nums[left] * nums[k] * nums[right]
                coins += dfs(left, k) + dfs(k, right)
                max_coins = max(max_coins, coins)

            memo[(left, right)] = max_coins
            return max_coins

        return dfs(0, n - 1)
```

In [5]:
```python
nums = [3,1,5,8]
Solution().maxCoins(nums)
```

Out[5]: 167

---

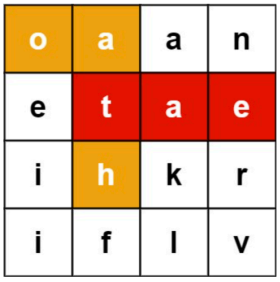
212. Word Search II

Hard Topics Companies Hint

Given an `m x n board` of characters and a list of strings `words`, return *all words on the board*.

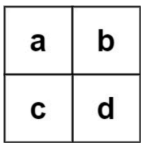
Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

Example 1:




```
Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]
```

Example 2:




```
Input: board = [["a","b"],["c","d"]], words = ["abcb"]
Output: []
```

Explanation of the Algorithm

The algorithm uses a **Trie (prefix tree)** and **DFS (Depth-First Search)** to efficiently find all words from the given `words` list that can be formed on the `board`. Here's the step-by-step breakdown:

Step 1: Building the Trie

The **Trie** is used to store all the words from the `words` list. This allows for efficient prefix matching, ensuring we don't waste time searching for words that don't exist.

Key Steps:

1. **Initialize the Trie**: Create a `TrieNode` class to represent each node in the Trie. Each node stores:
   - A dictionary `children` to store child nodes (next characters in the prefix).
   - A boolean `isWord` to mark the end of a word.
2. **Add Words to the Trie**: For each word in the `words` list:
   - Traverse each character in the word.
   - Add the character to the Trie if it doesn't exist.
   - At the end of the word, mark the `isWord` flag as `True` for the last node.

Step 2: Searching the Board with DFS

The board is a grid of characters where we need to search for words. We perform DFS starting from every cell to find all possible words that match prefixes in the Trie.

Key Steps:

1. **DFS Function**:
   - **Base Cases**:
     - Check if the current cell is out of bounds or has been visited (`(r, c) in visits`).
     - Check if the character in the current cell exists in the Trie node's children.
   - **Recursive Case**:
     - Add the current cell to the visited set.
     - Move deeper into the Trie using the character in the current cell.
     - If the current Trie node marks the end of a word (`isWord = True`), add the word to the result set.
     - Recursively explore the four neighboring cells (up, down, left, right).
2. **Backtracking**:
   - After exploring all neighbors, remove the current cell from the visited set to allow other paths to use it.

Step 3: Iterate Over All Cells

Start DFS from every cell on the board. For each cell:

- Check if the character exists as a prefix in the Trie.
- Explore all possible paths starting from that cell.

In [6]:
```python
from typing import List

class TrieNode:
    def __init__(self):
        self.children = {}
        self.isWord = False

    def addNode(self, word):
        curr = self
        for c in word:
            if c not in curr.children:
                curr.children[c] = TrieNode()
            curr = curr.children[c]
        curr.isWord = True  # Mark the end of the word


class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        # Build the Trie
        root = TrieNode()
        for w in words:
            root.addNode(w)

        ROW, COL = len(board), len(board[0])
        res, visits = set(), set()

        def dfs(r, c, node, word):
            # Base case: Check boundaries, visited cells, and character in Trie
            if (r < 0 or c < 0 or r >= ROW or c >= COL or
                    (r, c) in visits or board[r][c] not in node.children):
                return

            # Add the current cell to the visited set
            visits.add((r, c))
            # Move to the next Trie node
            node = node.children[board[r][c]]
            # Append the character to the current word
            word += board[r][c]

            # If this node represents the end of a word, add it to results
            if node.isWord:
                res.add(word)

            # Explore all four directions
            dfs(r - 1, c, node, word)
            dfs(r + 1, c, node, word)
            dfs(r, c - 1, node, word)
            dfs(r, c + 1, node, word)

            # Backtrack
            visits.remove((r, c))

        # Start DFS from each cell in the board
        for r in range(ROW):
            for c in range(COL):
                dfs(r, c, root, "")
```

```
        return list(res)
```

In [8]:
```
board = [
    ['o', 'a', 'a', 'n'],
    ['e', 't', 'a', 'e'],
    ['i', 'h', 'k', 'r'],
    ['i', 'f', 'l', 'v']
]
words = ["oath", "pea", "eat", "rain"]

solution = Solution()
print(solution.findWords(board, words))
```

```
['oath', 'eat']
```

Complexity Analysis

1. **Time Complexity**:
   - **Building the Trie**: $O(M)$, where M is the total number of characters in all words.
   - **DFS Search**:
     - In the worst case, all cells in the board are visited, and for each cell, we explore up to 4 neighbors.
     - Let the board size be $n \times m$, and let L be the maximum length of words. The worst-case complexity is $O(n \times m \times 4^L)$.
2. **Space Complexity**:
   - **Trie Storage**: $O(M)$.
   - **Visited Set**: $O(n \times m)$ in the worst case.

---

123. Best Time to Buy and Sell Stock III

Hard · Topics · Companies

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

Find the maximum profit you can achieve. You may complete **at most two transactions**.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.
```

Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Constraints:

- `1 <= prices.length <= 10^5`
- `0 <= prices[i] <= 10^5`

Explanation of the Code

1. Forward Pass (`left_profit`):

- **Goal**: Calculate the maximum profit you can make with a single transaction up to each day i.
- **Logic**:
  - Keep track of the minimum price seen so far (`min_price`).
  - For each day, calculate the profit if you sell on that day (`prices[i] - min_price`).
  - Update the `left_profit` array with the maximum profit so far.

2. Backward Pass (`right_profit`):

- **Goal**: Calculate the maximum profit you can make with a single transaction starting from each day i.
- **Logic**:
  - Keep track of the maximum price seen so far (`max_price`).
  - For each day, calculate the profit if you buy on that day (`max_price - prices[i]`).
  - Update the `right_profit` array with the maximum profit so far.

3. Combine Results:

- For each day i, the maximum profit is the sum of:
  - The best profit you can make with a single transaction up to day i (`left_profit[i]`).
  - The best profit you can make with a single transaction starting from day i (`right_profit[i]`).

In [9]:

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        n = len(prices)

        # Forward pass: max profit with one transaction up to day i
        left_profit = [0] * n
        min_price = prices[0]
        for i in range(1, n):
            min_price = min(min_price, prices[i])  # Update the minimum price
            left_profit[i] = max(left_profit[i - 1], prices[i] - min_price)  # Max profit up to day i

        # Backward pass: max profit with one transaction from day i onward
        right_profit = [0] * n
        max_price = prices[-1]
        for i in range(n - 2, -1, -1):
            max_price = max(max_price, prices[i])  # Update the maximum price
            right_profit[i] = max(right_profit[i + 1], max_price - prices[i])  # Max profit from day i onward

        # Combine the results of the two passes
        max_profit = 0
        for i in range(n):
            max_profit = max(max_profit, left_profit[i] + right_profit[i])

        return max_profit
```

Example Walkthrough

Input:

```python
prices = [3,3,5,0,0,3,1,4]
```

Forward Pass (`left_profit`):

```plaintext
Day:       0  1  2  3  4  5  6  7
Prices:    3  3  5  0  0  3  1  4
Min Price: 3  3  3  0  0  0  0  0
Left Prof: 0  0  2  2  2  3  3  4
```

Backward Pass (`right_profit`):

```plaintext
Day:       0  1  2  3  4  5  6  7
Prices:    3  3  5  0  0  3  1  4
Max Price: 4  4  4  4  4  4  4  4
Right Prof:4  4  4  4  4  3  3  0
```

Combine Results:

```plaintext
Day:          0  1  2  3  4  5  6  7
Left Profit:  0  0  2  2  2  3  3  4
Right Profit: 4  4  4  4  4  3  3  0
Combined:     4  4  6  6  6  6  6  4
```

Output:

```plaintext
6
```

315. Count of Smaller Numbers After Self

Hard | Topics | Companies

Given an integer array `nums`, return *an integer array* `counts` *where* `counts[i]` *is the number of smaller elements to the right of* `nums[i]`.

Example 1:

```
Input: nums = [5,2,6,1]
Output: [2,1,1,0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

Example 2:

```
Input: nums = [-1]
Output: [0]
```

Example 3:

```
Input: nums = [-1,-1]
Output: [0,0]
```

Constraints:

- `1 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`

In [1]:

```python
from typing import List

class Solution:
    def countSmaller(self, nums: List[int]) -> List[int]:
        # Step 1: Discretize the nums array
        sorted_nums = sorted(set(nums))
        rank_map = {num: i + 1 for i, num in enumerate(sorted_nums)}  # Rank starts from 1

        # Step 2: Initialize the Binary Indexed Tree (BIT)
        fenwick_tree = [0] * (len(sorted_nums) + 1)

        # Function to update the BIT
        def update(index, delta):
            while index < len(fenwick_tree):
                fenwick_tree[index] += delta
                index += index & -index

        # Function to query the BIT
        def query(index):
            result = 0
            while index > 0:
                result += fenwick_tree[index]
                index -= index & -index
            return result

        # Step 3: Process nums from right to left
        result = []
        for num in reversed(nums):
            rank = rank_map[num]  # Get the rank of the current number
            result.append(query(rank - 1))  # Count of smaller elements
            update(rank, 1)  # Add the current number's rank to the BIT

        return result[::-1]  # Reverse the result to match the original order
```

Step 1: BIT Operations

1. **Initialization**:
   - Start with a BIT array initialized to zero. For the discretized ranks, the size of the BIT is `len(sorted(nums)) + 1` (extra space for 1-based indexing).
   - BIT at the beginning:

```plaintext
BIT = [0, 0, 0, 0, 0]  (size = 5)
```

2. **BIT Operations**:
   - **Query Operation**: To count the sum of elements in the range `[1, index]`.
     - Algorithm: Keep adding `BIT[index]` and move `index -= index & -index` until `index = 0`.
   - **Update Operation**: To increment the value at a specific `index` by `delta`.
     - Algorithm: Increment `BIT[index]` by `delta` and move `index += index & -index` until `index > size`.

Step 2: Step-by-Step Execution

1. Process `1` (rank = 1):

- **Query**: Count of numbers smaller than `1` (rank `0`):
  - Start at `index = 0` → No elements smaller than `1`, so sum = `0`.
- **Update**: Add `1` to BIT at `index = 1`:
  - Update `BIT[1] += 1` → `BIT = [0, 1, 0, 0, 0]`
  - Propagate: `index = 2` → `BIT[2] += 1` → `BIT = [0, 1, 1, 0, 0]`
  - Propagate: `index = 4` → `BIT[4] += 1` → `BIT = [0, 1, 1, 0, 1]`

2. Process `6` (rank = 4):

- **Query**: Count of numbers smaller than `6` (rank `3`):
  - Start at `index = 3`:
    - Add `BIT[3] = 0` → `sum = 0`
    - Move to `index = 2`: Add `BIT[2] = 1` → `sum = 1`
    - Move to `index = 0` → Stop.
- **Update**: Add `1` to BIT at `index = 4`:
  - Update `BIT[4] += 1` → `BIT = [0, 1, 1, 0, 2]`

3. Process `2` (rank = 2):

- **Query**: Count of numbers smaller than `2` (rank `1`):
  - Start at `index = 1`:
    - Add `BIT[1] = 1` → `sum = 1`
    - Move to `index = 0` → Stop.
- **Update**: Add `1` to BIT at `index = 2`:
  - Update `BIT[2] += 1` → `BIT = [0, 1, 2, 0, 2]`
  - Propagate: `index = 4` → `BIT[4] += 1` → `BIT = [0, 1, 2, 0, 3]`

4. Process `5` (rank = 3):

- **Query**: Count of numbers smaller than `5` (rank `2`):
  - Start at `index = 2`:
    - Add `BIT[2] = 2` → `sum = 2`
    - Move to `index = 0` → Stop.
- **Update**: Add `1` to BIT at `index = 3`:
  - Update `BIT[3] += 1` → `BIT = [0, 1, 2, 1, 3]`
  - Propagate: `index = 4` → `BIT[4] += 1` → `BIT = [0, 1, 2, 1, 4]`

Step 3: Final BIT and Results

- Final BIT state:

```plaintext
BIT = [0, 1, 2, 1, 4]
```

- Reverse results collected during the process:

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

Trace of BIT Operations

Element	Rank	Query Index	Query Result	Update Index	Updated BIT
1	1	0	0	1	`[0, 1, 1, 0, 1]`
6	4	3	1	4	`[0, 1, 1, 0, 2]`
2	2	1	1	2	`[0, 1, 2, 0, 3]`
5	3	2	2	3	`[0, 1, 2, 1, 4]`

This gives the final answer `[2, 1, 1, 0]`.

---

51. N-Queens

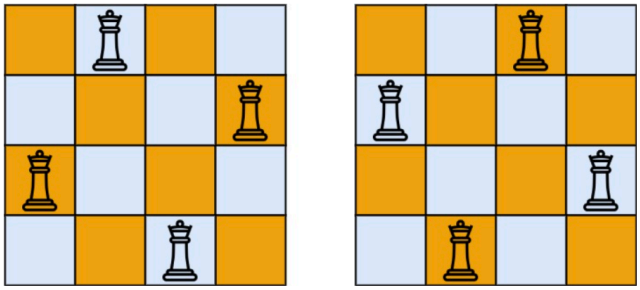
Hard | Topics | Companies

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

Constraints:

- `1 <= n <= 9`

In [2]:
```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        if n <= 0:
            return []

        # Sets to track column and diagonal conflicts
        col = set()
        posDig = set()  # Positive diagonals (r + c)
        negDig = set()  # Negative diagonals (r - c)

        res = []  # List to store the final solutions
        board = [["."] * n for _ in range(n)]  # Initialize the chessboard

        def backtracking(r):
            # Base case: all queens are placed
            if r == n:
                # Create a copy of the current board and add to results
                copy = ["".join(row) for row in board]
                res.append(copy)
                return

            # Try placing a queen in each column of row `r`
            for c in range(n):
                if c in col or (r - c) in negDig or (r + c) in posDig:
                    continue

                # Place queen and mark conflicts
                col.add(c)
                posDig.add(r + c)
                negDig.add(r - c)
                board[r][c] = "Q"

                # Recur to place queens in the next row
                backtracking(r + 1)

                # Backtrack: remove the queen and unmark conflicts
                col.remove(c)
                posDig.remove(r + c)
                negDig.remove(r - c)
                board[r][c] = "."

        backtracking(0)
        return res
```

In [3]:
```python
n = 4
solution = Solution()
print(solution.solveNQueens(n))
```

```
[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']]
```

Explanation of Code

Initialization

- A `board` is created as a 2D list filled with `"."` to represent empty spaces.
- `col`, `posDig`, and `negDig` are sets used to track conflicts in columns, positive diagonals (`r + c`), and negative diagonals (`r - c`).

Backtracking Function

1. **Base Case**: If `r == n`, all queens have been successfully placed. A copy of the board is created and added to `res`.
2. **Placement**:
   - Iterate through all columns `c` in the current row `r`.
   - Skip columns that have conflicts based on `col`, `posDig`, and `negDig`.
3. **Recursive Call**:
   - Place a queen at `(r, c)`, mark conflicts, and call `backtracking(r + 1)`.
4. **Backtracking**:
   - Remove the queen and unmark conflicts to explore other possibilities.

---

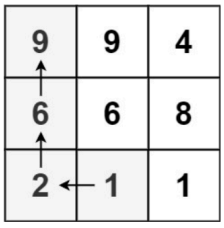
329. Longest Increasing Path in a Matrix

Hard Topics Companies

Given an `m x n` integers `matrix`, return *the length of the longest increasing path in* `matrix`.

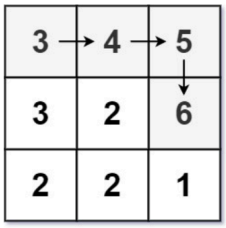
From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

Example 1:




```
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
```

Example 2:




```
Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]
Output: 4
Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.
```

Example 3:

```
Input: matrix = [[1]]
Output: 1
```

In [4]:
```python
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        # Step 1: Initialize necessary variables
```

```
        row = len(matrix)  # Number of rows in the matrix
        col = len(matrix[0])  # Number of columns in the matrix
        dp = {}  # Dictionary for memoization (to store previously calculated results)

        # Step 2: Define the DFS function for depth-first search with memoization
        def dfs(r, c, preVal):
            # Base Case: If the current cell is out of bounds or the value is not greater than preVal
            # we return 0, which means no valid path is possible from this cell
            if r < 0 or r == row or c < 0 or c == col or matrix[r][c] <= preVal:
                return 0  # Return 0 since we cannot move to this cell in an increasing path

            # If the current position (r, c) has already been visited and stored in dp, return the stored result
            if (r, c) in dp:
                return dp[(r, c)]  # Return the memoized result for this cell

            # Step 3: Initialize the result for the current cell
            res = 1  # At least the current cell itself is part of the path, so start with length 1

            # Step 4: Explore all four directions (down, up, left, right) to find the longest increasing path
            # For each direction, recursively call dfs, adding 1 to the result
            res = max(res, 1 + dfs(r + 1, c, matrix[r][c]))  # Move down
            res = max(res, 1 + dfs(r - 1, c, matrix[r][c]))  # Move up
            res = max(res, 1 + dfs(r, c - 1, matrix[r][c]))  # Move left
            res = max(res, 1 + dfs(r, c + 1, matrix[r][c]))  # Move right

            # Step 5: Memoize the result for the current cell to avoid redundant calculations
            dp[(r, c)] = res  # Store the computed result in the dp dictionary

            return res  # Return the longest increasing path length from this cell

        # Step 6: Iterate over all cells in the matrix and compute the longest increasing path starting from each cell
        for r in range(row):  # Loop over each row
            for c in range(col):  # Loop over each column
                dfs(r, c, -1)  # Call dfs for the current cell (start with -1 as preVal)

        # Step 7: The final result will be the maximum of all the computed paths in the dp dictionary
        return max(dp.values())  # Return the maximum value found in the dp dictionary (longest path)
```

In [6]:
```
matrix = [
    [9, 9, 4],
    [6, 6, 8],
    [2, 1, 0]
]

solution = Solution()
print(solution.longestIncreasingPath(matrix))  # Output: 4
```

5

---

37. Sudoku Solver

Hard · Topics · Companies

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]
Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

Explanation: The input board is shown above and the only valid solution is shown below:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

2. Data Structures

The algorithm uses three data structures to keep track of the numbers already present:

1. `row`: A list of sets where `row[i]` tracks the numbers present in the `i-th` row.
2. `col`: A list of sets where `col[j]` tracks the numbers present in the `j-th` column.
3. `boxes`: A list of sets where `boxes[box_id]` tracks the numbers present in the `box_id-th` 3x3 sub-box.

Formula for Box ID:

$$\text{box\_id} = (\text{row}//3) \times 3 + (\text{col}//3)$$

---

3. Initialization

- The algorithm iterates through the board to populate `row`, `col`, and `boxes` with the numbers already present.
- If a cell is not `"."`, the number is added to the corresponding row, column, and box.

---

4. Recursive Backtracking

The algorithm uses a recursive **backtracking** approach to try and fill the board:

1. **Base Case:**
   - If the current row index (`r`) equals 9, it means we've processed all rows, and the board is solved. Set `solved = True` and return.
2. **Determine Next Cell:**
   - For the current cell at `(r, c)`, calculate the next cell:

$$\text{new\_r} = r + \frac{c+1}{9}, \quad \text{new\_c} = (c+1) \mod 9$$

3. **Skip Pre-Filled Cells:**
   - If the current cell already contains a number (`board[r][c] != "."`), move to the next cell using recursion.
4. **Try Numbers 1–9:**
   - For each number from 1 to 9:
     - Convert it to a string (`num_str = str(num)`) because the board stores numbers as strings.
     - Check if `num_str` can be placed in the current cell:
       - It should not be in `row[r]`, `col[c]`, or `boxes[box_id]`.
     - If valid, add `num_str` to the row, column, and box sets, and place it in `board[r][c]`.
5. **Recurse:**
   - Call `backtracking` on the next cell.
6. **Backtrack:**
   - If placing `num_str` doesn't lead to a solution, remove it from the row, column, and box sets, and reset `board[r][c]` to `"."`.

In [7]:
```python
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        # Track used numbers for rows, columns, and boxes
        row = [set() for _ in range(9)]
        col = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # Initialize sets with existing numbers
        for i in range(9):
            for j in range(9):
                if board[i][j] != ".":
                    num = board[i][j]
                    row[i].add(num)
                    col[j].add(num)
                    box_id = (i // 3) * 3 + (j // 3)
                    boxes[box_id].add(num)

        def backtracking(r, c):
            nonlocal solved

            # If we reach beyond the last row, the board is solved
            if r == 9:
                solved = True
                return

            # Calculate the next cell to visit
            new_r = r + (c + 1) // 9
            new_c = (c + 1) % 9

            if board[r][c] != ".":
                # Skip filled cells
                backtracking(new_r, new_c)
            else:
                # Try placing numbers 1-9 in the current cell
                for num in range(1, 10):
                    num_str = str(num)  # Convert to string for consistency
                    box_id = (r // 3) * 3 + (c // 3)
                    if (num_str not in row[r]) and (num_str not in col[c]) and (num_str not in boxes[box_id]):
                        # Place the number
                        row[r].add(num_str)
                        col[c].add(num_str)
                        boxes[box_id].add(num_str)
                        board[r][c] = num_str

                        # Recur to the next cell
                        backtracking(new_r, new_c)

                        # Backtrack if not solved
                        if not solved:
                            row[r].remove(num_str)
                            col[c].remove(num_str)
                            boxes[box_id].remove(num_str)
                            board[r][c] = "."

        solved = False
        backtracking(0, 0)
```

In [9]:
```python
board = [
    ["5", "3", ".", ".", "7", ".", ".", ".", "."],
    ["6", ".", ".", "1", "9", "5", ".", ".", "."],
    [".", "9", "8", ".", ".", ".", ".", "6", "."],
    ["8", ".", ".", ".", "6", ".", ".", ".", "3"],
    ["4", ".", ".", "8", ".", "3", ".", ".", "1"],
    ["7", ".", ".", ".", "2", ".", ".", ".", "6"],
    [".", "6", ".", ".", ".", ".", "2", "8", "."],
    [".", ".", ".", "4", "1", "9", ".", ".", "5"],
    [".", ".", ".", ".", "8", ".", ".", "7", "9"],
]
Solution().solveSudoku(board)

board
```

Out[9]:
```
[['5', '3', '4', '6', '7', '8', '9', '1', '2'],
 ['6', '7', '2', '1', '9', '5', '3', '4', '8'],
 ['1', '9', '8', '3', '4', '2', '5', '6', '7'],
 ['8', '5', '9', '7', '6', '1', '4', '2', '3'],
 ['4', '2', '6', '8', '5', '3', '7', '9', '1'],
 ['7', '1', '3', '9', '2', '4', '8', '5', '6'],
 ['9', '6', '1', '5', '3', '7', '2', '8', '4'],
 ['2', '8', '7', '4', '1', '9', '6', '3', '5'],
 ['3', '4', '5', '2', '8', '6', '1', '7', '9']]
```

Example:

Input Board (Partial):

```plaintext
[
 ["5", "3", ".", ".", "7", ".", ".", ".", "."],
 ["6", ".", ".", "1", "9", "5", ".", ".", "."],
 [".", "9", "8", ".", ".", ".", ".", "6", "."],
 # Rest of the rows omitted for simplicity
]
```

Starting Position:

Let's say we are at cell `(0, 1)` (row 0, column 1), which contains `"3"`.

Next Cell Calculation:

1. $r = 0$, $c = 1$ (current position).
2. Calculate new_r and new_c:
   - $\text{new\_r} = r + \frac{c+1}{9} = 0 + \frac{2}{9} = 0$ (no row increment since $c + 1$ is less than 9).
   - $\text{new\_c} = (c + 1) \mod 9 = (1 + 1) \mod 9 = 2$.
3. **Next Cell:** `(0, 2)` (row 0, column 2).

Continuing to the Next Cell:

Now at `(0, 2)` (which is `"."`), move to `(0, 3)`:

1. $r = 0$, $c = 2$.
2. $\text{new\_r} = 0 + \frac{3}{9} = 0$, $\text{new\_c} = (2 + 1) \mod 9 = 3$.
3. **Next Cell:** `(0, 3)`.

Example of Row Transition:

At the last column of the row, e.g., `(0, 8)`:

1. $r = 0$, $c = 8$.
2. $\text{new\_r} = 0 + \frac{8+1}{9} = 0 + 1 = 1$ (move to the next row).
3. $\text{new\_c} = (8 + 1) \mod 9 = 0$ (reset column to 0).
4. **Next Cell:** `(1, 0)` (first column of the next row).

Formula for Box ID:

For a cell at position (r, c) (row r, column c):

$$\text{box\_id} = \left(\frac{r}{3}\right) \times 3 + \left(\frac{c}{3}\right)$$

- $r//3$: Determines which group of 3 rows the cell is in (0, 1, or 2).
- $c//3$: Determines which group of 3 columns the cell is in (0, 1, or 2).

Explanation:

- The grid is divided into 9 boxes, each represented by an ID from 0 to 8.
- These IDs are assigned as follows:

```plaintext
Box IDs:
0 | 1 | 2
---------
3 | 4 | 5
---------
6 | 7 | 8
```

Example Calculations:

1. For cell $(0, 0)$ (row 0, column 0):

   $$\text{box\_id} = (0//3) \times 3 + (0//3) = 0 \times 3 + 0 = 0$$

   The cell belongs to **Box 0**.

2. For cell $(4, 7)$ (row 4, column 7):

   $$\text{box\_id} = (4//3) \times 3 + (7//3) = 1 \times 3 + 2 = 5$$

   The cell belongs to **Box 5**.

3. For cell $(8, 8)$ (row 8, column 8):

   $$\text{box\_id} = (8//3) \times 3 + (8//3) = 2 \times 3 + 2 = 8$$

   The cell belongs to **Box 8**.

---

301. Remove Invalid Parentheses

Hard | Topics | Companies | Hint

Given a string `s` that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return *a list of **unique strings** that are valid with the minimum number of removals*. You may return the answer in **any order**.

Example 1:

```
Input: s = "()())()"
Output: ["(())()","()()()"]
```

Example 2:

```
Input: s = "(a())()"
Output: ["(a())()","(a)()()"]
```

Example 3:

```
Input: s = ")("
Output: [""]
```

Constraints:

- `1 <= s.length <= 25`
- `s` consists of lowercase English letters and parentheses `'('` and `')'`.
- There will be at most `20` parentheses in `s`.

In [ ]:

In [10]:
```python
from collections import deque

class Solution:
    def removeInvalidParentheses(self, s: str) -> List[str]:
        def is_valid(string):
            count = 0
            for char in string:
                if char == '(':
                    count += 1
                elif char == ')':
                    count -= 1
                if count < 0:  # More ')' than '('
                    return False
            return count == 0  # Ensure balanced

        queue = deque([s])
        visited = set([s])  # To avoid duplicates
        found = False
        result = []

        while queue:
            current = queue.popleft()

            if is_valid(current):
                result.append(current)
                found = True

            if found:
                continue  # No need to generate more levels

            for i in range(len(current)):
                if current[i] not in "()":
                    continue
                next_string = current[:i] + current[i+1:]  # Remove one parenthesis
                if next_string not in visited:
                    visited.add(next_string)
                    queue.append(next_string)

        return result
```

In [18]:
```python
s = ")("
Solution().removeInvalidParentheses(s)
```

Out[18]:
```
['']
```

Step-by-Step Execution:

Initial Setup:

- Start with `queue = deque(["()())()"])`
- `visited = {"()())()"}`
- `result = []`

Level 0: Process the string `()())()`

1. Check if `()())()` is valid:
   - Count `(` and `)` balance: `()())()` is **not valid** because of an extra `)`.
2. Generate new strings by removing one parenthesis:
   - Removing at index 0 → `)())()`
   - Removing at index 1 → `(())()`
   - Removing at index 2 → `()))()`
   - Removing at index 3 → `()()()`
   - Removing at index 4 → `()())(`
   - Removing at index 5 → `()()))`
   - Removing at index 6 → `()())(`
3. Add these new strings to the `queue` and `visited`:
   - `queue = deque([")())()", "(())()", "()))()", "()()()", "()())(", "()()))"])`
   - `visited = {"()())()", ")())()", "(())()", "()))()", "()()()", "()())(", "()()))"}`

Level 1: Process each string in the `queue`

---

1. Process `")())()"`:
   - Check validity → **Not valid**.
   - Generate new strings by removing one parenthesis:
     - Removing at index 0 → `())()`
     - Removing at index 1 → `))()`
     - Removing at index 2 → `)()()`
     - Removing at index 3 → `)())`
     - Removing at index 4 → `)()(`
   - Add to `queue` and `visited` if not already visited.

---

2. Process `"(())()"`:
   - Check validity → **Valid**! Add to `result`.
   - Result now: `result = ["(())()"]`.

---

3. Process `"))))()"`, `())()` → Both invalid, process similarly by removing one character.
4. Process `"()()()"`:
   - Check validity → **Valid**! Add to `result`.
   - Result now: `result = ["(())()", "()()()"]`.

Stop Condition:

Once a valid string is found, stop processing deeper levels (since all valid strings of the same length are at this level).

Final Output:

```plaintext
result = ["(())()", "()()()"]
```

Visualization of BFS Levels:

Level 0:

- Start with `()())()`

Level 1:

- Generated: `")())()", "(())()", "()))()", "()()()", "()())(", "()()))"`

Level 2:

- Valid strings found: `"()()()"` and `"(())()"`

This ensures all valid strings are found with the minimum number of removals.

---

140. Word Break II

Hard | Topics | Companies

Given a string `s` and a dictionary of strings `wordDict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

> **Input:** s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]
> **Output:** ["cats and dog","cat sand dog"]

Example 2:

> **Input:** s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]
> **Output:** ["pine apple pen apple","pineapple pen apple","pine applepen apple"]
> **Explanation:** Note that you are allowed to reuse a dictionary word.

Example 3:

> **Input:** s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
> **Output:** []

In [19]:
```python
from typing import List

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> List[str]:
        wordSet = set(wordDict)  # Use a set for faster lookups
        memo = {}  # Memoization dictionary

        def dfs(substring):
            if substring in memo:  # Check if the result is already computed
                return memo[substring]
            if not substring:  # Base case: if the string is empty
                return [""]

            result = []
            for i in range(1, len(substring) + 1):
                prefix = substring[:i]
                if prefix in wordSet:  # Check if prefix is valid
                    # Recursively solve for the remaining string
                    suffix_results = dfs(substring[i:])
                    for sentence in suffix_results:
                        if sentence:
                            result.append(prefix + " " + sentence)
                        else:
                            result.append(prefix)

            memo[substring] = result  # Cache the result
            return result

        return dfs(s)
```

In [20]:
```python
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
Solution().wordBreak(s,wordDict)
```

Out[20]:
```
['cat sand dog', 'cats and dog']
```

Explanation of Code:

1. `wordSet`:
   - Converts `wordDict` into a set for O(1) lookup.
2. `dfs(substring)`:
   - Finds all possible sentences starting from the given substring.
   - If the substring is in the memo, return the cached result.
3. **Base Case**:
   - If the substring is empty, return a list with an empty string.
4. **Recursive Step**:
   - Iterate through possible prefixes of the string.
   - If the prefix is valid (exists in `wordSet`), recursively solve for the suffix.
   - Combine the prefix with each result from the recursive call.
5. **Result Combination**:
   - Append the valid prefix and the suffix (if non-empty) to the result.

Example Walkthrough:

Input:

```plaintext
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
```

Execution:

1. `dfs("catsanddog")`:
   - Valid prefixes: `"cat"`, `"cats"`
   - Recursively solve for `"sanddog"` and `"anddog"`.
2. `dfs("sanddog")`:
   - Valid prefix: `"sand"`
   - Recursively solve for `"dog"`.
3. `dfs("dog")`:
   - Valid prefix: `"dog"`
   - Base case: return `["dog"]`.
4. Combine results:
   - `"sand dog"`
   - `"and dog"`

Output:

```plaintext
["cat sand dog", "cats and dog"]
```