20. Valid Parentheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "(]"

Output: false

Example 4:

Input: s = "([])"

Output: true

Constraints:

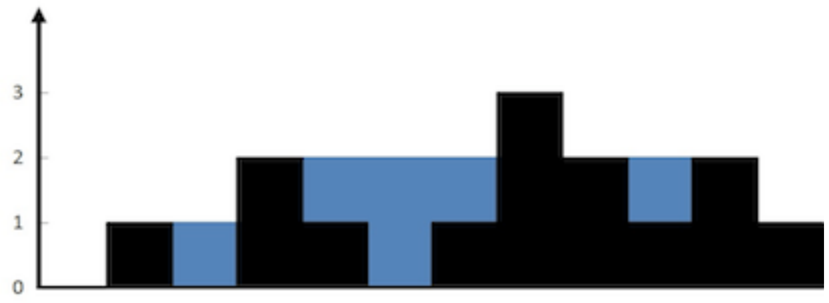
- `1 <= s.length <=` 10^4
- `s` consists of parentheses only `'()[]{}'`.

```
class Solution:
    def isValid(self, s: str) -> bool:
        if not s :return True
        stack = []
        for i in s:
            if not stack :
                stack.append(i)
            elif stack[-1] == "(" and i == ")":
                stack.pop()
            elif stack[-1] == "{" and i == "}":
                stack.pop()
            elif stack[-1] == "[" and i == "]":
                stack.pop()
            else:
                stack.append(i)
        return True if not stack else False
```

42. Trapping Rain Water

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
```

Example 2:

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

Constraints:

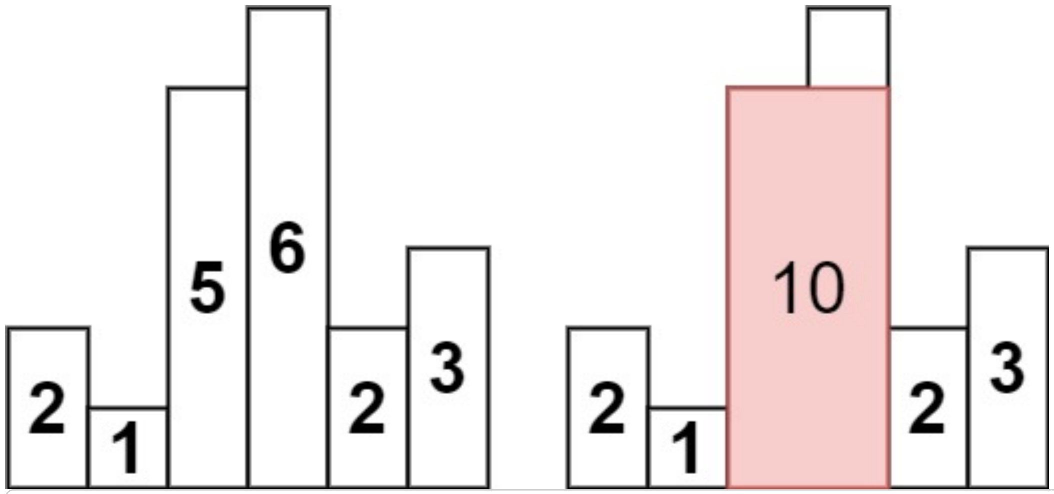
- `n == height.length`
- `1 <= n <= 2 * 10`^{4}
- `0 <= height[i] <= 10`^{5}

```
from typing import List

class Solution:
    def trap(self, height: List[int]) -> int:
        n = len(height)
        if n == 0:
            return 0

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

        # Fill left_max
        left_max[0] = height[0]
        for i in range(1, n):
            left_max[i] = max(left_max[i - 1], height[i])

        # Fill right_max
        right_max[n - 1] = height[n - 1]
        for i in range(n - 2, -1, -1):
            right_max[i] = max(right_max[i + 1], height[i])

        # Calculate trapped water
        res = 0
        for i in range(n):
            res += min(left_max[i], right_max[i]) - height[i]

        return res
```

84. Largest Rectangle in Histogram

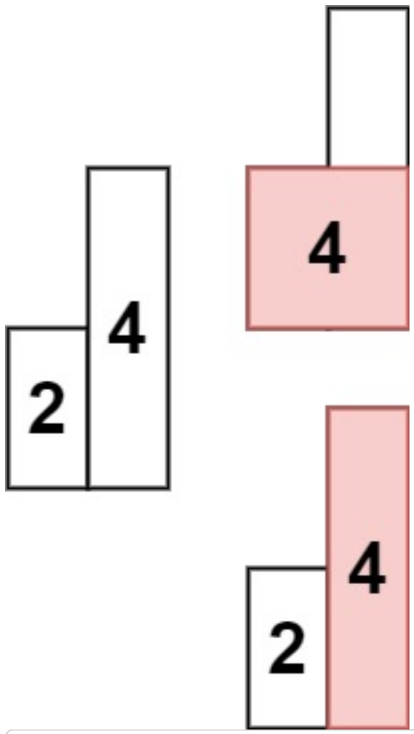
Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram.*

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
```

Constraints:

- `1 <= heights.length <= 10`^{5}

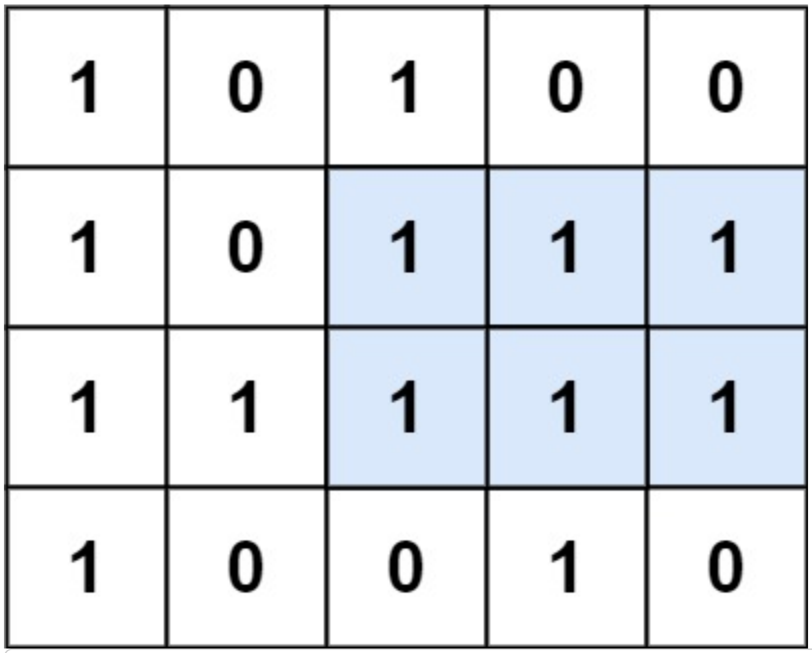
- `0 <= heights[i] <= 10`^{4}

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = []
        area = 0
        n = len(heights)
        for i in range(n):
            while stack and heights[stack[-1]] > heights[i]:
                bar = stack.pop()
                nse = i
                pse = stack[-1] if stack else -1
                area = max(area,heights[bar] * (nse - pse -1))
            stack.append(i)

        while stack:
            bar = stack.pop()
            nse = n
            pse = stack[-1] if stack else -1
            area = max(area,heights[bar] * (nse - pse -1))
        return area
```

85. Maximal Rectangle

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"],['
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
```

Constraints:

- `rows == matrix.length`
- `cols == matrix[i].length`
- `1 <= row, cols <= 200`
- `matrix[i][j]` is `'0'` or `'1'`.

```
class Solution:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        n = len(matrix)
        m = len(matrix[0])
        for j in range(m):
            for i in range(n):
                if i > 0 and matrix[i][j] != "0":
                    matrix[i][j] = int(matrix[i-1][j]) + int(matrix[i][j])
                else:
                    matrix[i][j]  = int(matrix[i][j])

        area = 0
        def larget_Histogram(heights):
            nonlocal area,m
            stack = []
            pse = 0
            nse = 0
            for i in range(m):
                while stack and heights[stack[-1]] > heights[i]:
                    bar = stack.pop()
                    pse = stack[-1] if stack else -1
                    nse = i
                    area = max(area, heights[bar] *(nse - pse -1) )
                stack.append(i)

            while stack:
                bar = stack.pop()
                pse = stack[-1] if stack else -1
                nse = m
                area = max(area, heights[bar] *(nse - pse -1) )


        for ind in range(n):
            larget_Histogram(matrix[ind])
        return area
```

146. LRU Cache

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache (https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU)**.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity`.

- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Example 1:

```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);    // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);    // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);    // return -1 (not found)
lRUCache.get(3);    // return 3
lRUCache.get(4);    // return 4
```

Constraints:

- `1 <= capacity <= 3000`
- `0 <= key <=` 10^4
- `0 <= value <=` 10^5
- At most $2 * 10^5$ calls will be made to `get` and `put`.

```python
from heapq import heappush,heappop
class Node:
    def __init__(self,node,keyy):
        self.val = node
        self.keyy = keyy
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.head = Node("head",-1)
        self.tail = Node("tail",-1)
        self.head.next = self.tail
        self.tail.prev = self.head
        self.map = {}
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key not in self.map:return -1
        x = self.head
        node = self.map[key]
        v = node.val
        temp = node.prev
        temp.next = node.next
        temp.next.prev = temp
        temp = self.head.next
        node.next  = temp
        temp.prev = node
        self.head.next = node
        node.prev = self.head
        return v

    def put(self, key: int, value: int) -> None:
        if key in self.map:
            self.map[key].val = value
            self.get(key)
        else:
            node = Node(value,key)
            temp = self.head.next
            node.next  = temp
            temp.prev = node
            self.head.next = node
            node.prev = self.head
            if self.capacity > 0:
                self.capacity-=1
            else:
                keyy = self.tail.prev.keyy
                if keyy in self.map:
                    del self.map[keyy]
                temp = self.tail.prev.prev
                temp.next  = self.tail
                self.tail.prev = temp
            self.map[key] = node
```

155. Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with `O(1)` time complexity for each function.

Example 1:

```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Constraints:

- `-2`^{31} `<= val <= 2`^{31} `- 1`
- Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.
- At most `3 * 10`^{4} calls will be made to `push`, `pop`, `top`, and `getMin`.

```
class MinStack:
    def __init__(self):
        self.stack = []

    def push(self, val: int) -> None:
        minn = self.getMin()
        if minn == None or val < minn :
            minn = val
        self.stack.append([val,minn])

    def pop(self) -> None:
        return self.stack.pop()[0]

    def top(self) -> int:
        return self.stack[-1][0]

    def getMin(self) -> int:
        return self.stack[-1][1] if self.stack else None
```

225. Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element x to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a queue, which means that only `push to back`, `peek/pop from front`, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

```
Input
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 2, 2, false]

Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False
```

Constraints:

- `1 <= x <= 9`
- At most `100` calls will be made to `push`, `pop`, `top`, and `empty`.
- All the calls to `pop` and `top` are valid.

Follow-up: Can you implement the stack using only one queue?

```python
class MyStack:

    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x: int) -> None:
        self.q1.append(x)

    def pop(self) -> int:
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())

        popped_element = self.q1.popleft()

        # Swap q1 and q2
        self.q1, self.q2 = self.q2, self.q1

        return popped_element

    def top(self) -> int:
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())

        top_element = self.q1[0]

        self.q2.append(self.q1.popleft())

        # Swap q1 and q2
        self.q1, self.q2 = self.q2, self.q1

        return top_element

    def empty(self) -> bool:
        return len(self.q1) == 0
```

232. Implement Queue using Stacks

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push` , `peek` , `pop` , and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element x to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

```
Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 1, 1, false]

Explanation
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

Constraints:

- `1 <= x <= 9`
- At most `100` calls will be made to `push`, `pop`, `peek`, and `empty`.
- All the calls to `pop` and `peek` are valid.

Follow-up: Can you implement the queue such that each operation is **amortized (https://en.wikipedia.org/wiki/Amortized_analysis)** `O(1)` time complexity? In other words, performing `n` operations will take overall `O(n)` time even if one of those operations may take longer.

```
class MyQueue:

    def __init__(self):
        self.stack = []
        self.length = 0

    def push(self, x: int) -> None:
        self.length +=1
        temp = [x]
        for i in self.stack:
            temp.append(i)
        self.stack = temp.copy()

    def pop(self) -> int:
        self.length -=1
        return self.stack.pop()

    def peek(self) -> int:
        return self.stack[-1]


    def empty(self) -> bool:
        return True if self.length==0 else False
```

239. Sliding Window Maximum

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`

```
from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        n = len(nums)
        if not nums or k == 0:
            return []

        result = []
        que = deque()  # Will store indexes of elements

        for i in range(n):
            # Remove indexes of elements not in the window
            while que and que[0] <= i - k:
                que.popleft()

            # Remove elements smaller than the current from the back of the deque
            while que and nums[que[-1]] < nums[i]:
                que.pop()

            que.append(i)

            # Append the current max to the result once the first window is fully t
raversed
            if i >= k - 1:
                result.append(nums[que[0]])

        return result
```

402. Remove K Digits

Given string num representing a non-negative integer `num`, and an integer `k`, return *the smallest possible integer after removing* `k` *digits from* `num`.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must no
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing which
```

Constraints:

- `1 <= k <= num.length <= 10`^{5}
- `num` consists of only digits.
- `num` does not have any leading zeros except for the zero itself.

```
import sys

sys.set_int_max_str_digits(1000000)
class Solution:
    def removeKdigits(self, num: str, k: int) -> str:
        if len(num) == k :return "0"
        stack = []
        for i in num:
            while stack and stack[-1] > i and k>0:
                stack.pop()
                k-=1
            stack.append(i)

        stack = stack[:-k] if k > 0 else stack
        result = "".join(stack).lstrip('0') #Remove leasding zero
        return result if result else "0"
```

460. LFU Cache

Design and implement a data structure for a Least Frequently Used (LFU) (https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the `LFUCache` class:

- `LFUCache(int capacity)` Initializes the object with the `capacity` of the data structure.
- `int get(int key)` Gets the value of the `key` if the `key` exists in the cache. Otherwise, returns `-1`.
- `void put(int key, int value)` Update the value of the `key` if present, or inserts the `key` if not already present. When the cache reaches its `capacity`, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** `key` would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to `1` (due to the `put` operation). The **use counter** for a key in the cache is incremented either a `get` or `put` operation is called on it.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Example 1:

```
Input
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is  most
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1);   // cache=[1,_], cnt(1)=1
lfu.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1);      // return 1
                 // cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3);   // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
                 // cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4);   // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
                 // cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4);      // return 4
                 // cache=[4,3], cnt(4)=2, cnt(3)=3
```

Constraints:

- `1 <= capacity <= 10`^{4}
- `0 <= key <= 10`^{5}
- `0 <= value <= 10`^{9}
- At most `2 * 10`^{5} calls will be made to `get` and `put`.

```
class Node:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.freq = 1
        self.prev = self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = Node(None, None)  # Dummy head
        self.tail = Node(None, None)  # Dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def insert_at_head(self, node):
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def remove_last(self):
        if self.tail.prev == self.head:
            return None
        last = self.tail.prev
        self.remove(last)
        return last

    def is_empty(self):
        return self.head.next == self.tail

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.size = 0
        self.min_freq = 0
        self.node_map = {}  # key -> node
        self.freq_map = {}  # freq -> DoublyLinkedList

    def _update(self, node):
        freq = node.freq
        self.freq_map[freq].remove(node)

        if self.freq_map[freq].is_empty():
            del self.freq_map[freq]
            if self.min_freq == freq:
                self.min_freq += 1

        node.freq += 1
        new_freq = node.freq
        if new_freq not in self.freq_map:
            self.freq_map[new_freq] = DoublyLinkedList()
        self.freq_map[new_freq].insert_at_head(node)

    def get(self, key):
        if key not in self.node_map:
            return -1
        node = self.node_map[key]
        self._update(node)
        return node.val

    def put(self, key, value):
        if self.capacity == 0:
            return

        if key in self.node_map:
            node = self.node_map[key]
            node.val = value
            self._update(node)
        else:
            if self.size >= self.capacity:
                # Evict least frequently used node
                lfu_list = self.freq_map[self.min_freq]
                evicted = lfu_list.remove_last()
                if evicted:
                    del self.node_map[evicted.key]
                    self.size -= 1

            # Insert new node
            new_node = Node(key, value)
            self.node_map[key] = new_node
            if 1 not in self.freq_map:
                self.freq_map[1] = DoublyLinkedList()
            self.freq_map[1].insert_at_head(new_node)
            self.min_freq = 1
            self.size += 1
```

496. Next Greater Element I

The **next greater element** of some element `x` in an array is the **first greater** element that is **to the right** of `x` in the same array.

You are given two **distinct 0-indexed** integer arrays `nums1` and `nums2`, where `nums1` is a subset of `nums2`.

For each `0 <= i < nums1.length`, find the index `j` such that `nums1[i] == nums2[j]` and determine the **next greater element** of `nums2[j]` in `nums2`. If there is no next greater element, then the answer for this query is `-1`.

Return *an array* `ans` *of length* `nums1.length` *such that* `ans[i]` *is the* ***next greater element*** *as described above.*

Example 1:

```
Input: nums1 = [4,1,2], nums2 = [1,3,4,2]
Output: [-1,3,-1]
Explanation: The next greater element for each value of nums1 is as follows:
- 4 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answ
- 1 is underlined in nums2 = [1,3,4,2]. The next greater element is 3.
- 2 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answ
```

Example 2:

```
Input: nums1 = [2,4], nums2 = [1,2,3,4]
Output: [3,-1]
Explanation: The next greater element for each value of nums1 is as follows:
- 2 is underlined in nums2 = [1,2,3,4]. The next greater element is 3.
- 4 is underlined in nums2 = [1,2,3,4]. There is no next greater element, so the answ
```

Constraints:

- `1 <= nums1.length <= nums2.length <= 1000`
- `0 <= nums1[i], nums2[i] <=` 10^4
- All integers in `nums1` and `nums2` are **unique**.
- All the integers of `nums1` also appear in `nums2`.

Follow up: Could you find an `O(nums1.length + nums2.length)` solution?

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        n2 = len(nums2)
        stack = []
        ans = [-1] *n2
        for i in range(n2-1,-1,-1):
            while stack and nums2[i] > stack[-1]:
                stack.pop()
            if stack:
                ans[i] = stack[-1]
            stack.append(nums2[i])

        res = []
        for num in nums1:
            for j in range(n2):
                if nums2[j] == num:
                    res.append(ans[j])
                    break
        return res
```

503. Next Greater Element II

Given a circular integer array `nums` (i.e., the next element of `nums[nums.length - 1]` is `nums[0]`), return *the **next greater number** for every element in* `nums`.

The **next greater number** of a number `x` is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return `-1` for this number.

Example 1:

```
Input: nums = [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number.
The second 1's next greater number needs to search circularly, which is also 2.
```

Example 2:

```
Input: nums = [1,2,3,4,3]
Output: [2,3,4,-1,4]
```

Constraints:

- `1 <= nums.length <= 10`^{4}
- `-10`^{9} `<= nums[i] <= 10`^{9}

```
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        n = len(nums)
        stack = []
        ans = [-1]*n

        for i in range((2*n)-1,-1,-1):
            while stack and stack[-1] <= nums[i%n]:
                stack.pop()

            if stack:
                ans[i%n] = stack[-1]
            stack.append(nums[i%n])
        return ans
```

735. Asteroid Collision

We are given an array `asteroids` of integers representing asteroids in a row. The indices of the asteriod in the array represent their relative position in space.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

```
Input: asteroids = [5,10,-5]
Output: [5,10]
Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.
```

Example 2:

```
Input: asteroids = [8,-8]
Output: []
Explanation: The 8 and -8 collide exploding each other.
```

Example 3:

```
Input: asteroids = [10,2,-5]
Output: [10]
Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting ir
```

Constraints:

- `2 <= asteroids.length <= 10`^{4}
- `-1000 <= asteroids[i] <= 1000`
- `asteroids[i] != 0`

```
class Solution:
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        stack = []
        n = len(asteroids)
        for i in range(n):
            while stack and  asteroids[i] < 0 < stack[-1]:
                if abs(stack[-1]) < abs(asteroids[i]):
                    stack.pop()
                    continue
                elif abs(stack[-1]) == abs(asteroids[i]):
                    stack.pop()
                break
            else:
                stack.append(asteroids[i])
        return stack
```

901. Online Stock Span

Design an algorithm that collects daily price quotes for some stock and returns **the span** of that stock's price for the current day.

The **span** of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

- For example, if the prices of the stock in the last four days is `[7,2,1,2]` and the price of the stock today is `2`, then the span of today is `4` because starting from today, the price of the stock was less than or equal `2` for `4` consecutive days.
- Also, if the prices of the stock in the last four days is `[7,34,1,2]` and the price of the stock today is `8`, then the span of today is `3` because starting from today, the price of the stock was less than or equal `8` for `3` consecutive days.

Implement the `StockSpanner` class:

- `StockSpanner()` Initializes the object of the class.
- `int next(int price)` Returns the **span** of the stock's price given that today's price is `price`.

Example 1:

```
Input
["StockSpanner", "next", "next", "next", "next", "next", "next", "next"]
[[], [100], [80], [60], [70], [60], [75], [85]]
Output
[null, 1, 1, 1, 2, 1, 4, 6]

Explanation
StockSpanner stockSpanner = new StockSpanner();
stockSpanner.next(100); // return 1
stockSpanner.next(80);  // return 1
stockSpanner.next(60);  // return 1
stockSpanner.next(70);  // return 2
stockSpanner.next(60);  // return 1
stockSpanner.next(75);  // return 4, because the last 4 prices (including today's pri
stockSpanner.next(85);  // return 6
```

Constraints:

- `1 <= price <= 10`^{5}
- At most `10`^{4} calls will be made to `next`.

```
class StockSpanner:
    def __init__(self):
        self.stack = []  # Each element is (price, span)

    def next(self, price: int) -> int:
        span = 1
        # Accumulate span by popping all prices <= current price
        while self.stack and self.stack[-1][0] <= price:
            span += self.stack.pop()[1]

        self.stack.append((price, span))
        return span
```

907. Sum of Subarray Minimums

Given an array of integers arr, find the sum of `min(b)` , where `b` ranges over every (contiguous) subarray of `arr` . Since the answer may be large, return the answer **modulo** $10^9 + 7$.

Example 1:

```
Input: arr = [3,1,2,4]
Output: 17
Explanation:
Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.
Sum is 17.
```

Example 2:

```
Input: arr = [11,81,94,43,3]
Output: 444
```

Constraints:

- $1 <= arr.length <= 3 * 10^4$
- $1 <= arr[i] <= 3 * 10^4$

```
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        stack = [] # keep index for the latest smaller values
        res = [0] * len(arr)

        for i in range(len(arr)):
            while stack and arr[stack[-1]] > arr[i]:
                stack.pop()

            j = stack[-1] if stack else -1
            res[i] = res[j] + (i - j) * arr[i]

            stack.append(i)

        return sum(res) % (10**9+7)
```

2104. Sum of Subarray Ranges

You are given an integer array `nums`. The **range** of a subarray of `nums` is the difference between the largest and smallest element in the subarray.

Return *the **sum of all** subarray ranges of* `nums`.

A subarray is a contiguous **non-empty** sequence of elements within an array.

Example 1:

```
Input: nums = [1,2,3]
Output: 4
Explanation: The 6 subarrays of nums are the following:
[1], range = largest - smallest = 1 - 1 = 0
[2], range = 2 - 2 = 0
[3], range = 3 - 3 = 0
[1,2], range = 2 - 1 = 1
[2,3], range = 3 - 2 = 1
[1,2,3], range = 3 - 1 = 2
So the sum of all ranges is 0 + 0 + 0 + 1 + 1 + 2 = 4.
```

Example 2:

```
Input: nums = [1,3,3]
Output: 4
Explanation: The 6 subarrays of nums are the following:
[1], range = largest - smallest = 1 - 1 = 0
[3], range = 3 - 3 = 0
[3], range = 3 - 3 = 0
[1,3], range = 3 - 1 = 2
[3,3], range = 3 - 3 = 0
[1,3,3], range = 3 - 1 = 2
So the sum of all ranges is 0 + 0 + 0 + 2 + 0 + 2 = 4.
```

Example 3:

```
Input: nums = [4,-2,-3,4,1]
Output: 59
Explanation: The sum of all subarray ranges of nums is 59.
```

Constraints:

- `1 <= nums.length <= 1000`
- -10^9 `<= nums[i] <=` 10^9

Follow-up: Could you find a solution with `O(n)` time complexity?

```
class Solution:
    def subArrayRanges(self, nums: List[int]) -> int:
        n = len(nums)

        # the answer will be sum{ Max(subarray) - Min(subarray) } over all possible
subarray
        # which decomposes to sum{Max(subarray)} - sum{Min(subarray)} over all poss
ible subarray
        # so totalsum = maxsum - minsum
        # we calculate minsum and maxsum in two different loops
        minsum = maxsum = 0

        # first calculate sum{ Min(subarray) } over all subarrays
        # sum{ Min(subarray) } = sum(f(i) * nums[i]) ; i=0..n-1
        # where f(i) is number of subarrays where nums[i] is the minimum value
        # f(i) = (i - index of the previous smaller value) * (index of the next sma
ller value - i) * nums[i]
        # we can claculate these indices in linear time using a monotonically incre
asing stack.
        stack = []
        for next_smaller in range(n + 1):
            # we pop from the stack in order to satisfy the monotonically increasin
g order property
            # if we reach the end of the iteration and there are elements present i
n the stack, we pop all of them
            while stack and (next_smaller == n or nums[stack[-1]] > nums[next_small
er]):
                i = stack.pop()
                prev_smaller = stack[-1] if stack else -1
                minsum += nums[i] * (next_smaller - i) * (i - prev_smaller)
            stack.append(next_smaller)

        # then calculate sum{ Max(subarray) } over all subarrays
        # sum{ Max(subarray) } = sum(f'(i) * nums[i]) ; i=0..n-1
        # where f'(i) is number of subarrays where nums[i] is the maximum value
        # f'(i) = (i - index of the previous larger value) - (index of the next lar
ger value - i) * nums[i]
        # this time we use a monotonically decreasing stack.
        stack = []
        for next_larger in range(n + 1):
            # we pop from the stack in order to satisfy the monotonically decreasin
g order property
            # if we reach the end of the iteration and there are elements present i
n the stack, we pop all of them
            while stack and (next_larger == n or nums[stack[-1]] < nums[next_large
r]):
```

```
            i = stack.pop()
            prev_larger = stack[-1] if stack else -1
            maxsum += nums[i] * (next_larger - i) * (i - prev_larger)
        stack.append(next_larger)

    return maxsum - minsum
```