20. Valid Parentheses

```
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

```
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

```
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

```
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

```
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

```
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

```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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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
```