45. Jump Game II

You are given a **0-indexed** array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

- `0 <= j <= nums[i]` and
- `i + j < n`

Return *the minimum number of jumps to reach* `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step fr
```

Example 2:

```
Input: nums = [2,3,0,1,4]
Output: 2
```

Constraints:

- `1 <= nums.length <= 10^4`
- `0 <= nums[i] <= 1000`
- It's guaranteed that you can reach `nums[n - 1]`.

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        steps = 0
        current_end = 0
        farthest = 0

        for i in range(len(nums) - 1):
            farthest = max(farthest, i + nums[i])
            if i == current_end:
                steps += 1
                current_end = farthest

        return steps
```

---

55. Jump Game

You are given an integer array `nums` . You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` *if you can reach the last index, or* `false` *otherwise.*

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump lengt
```

Constraints:

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

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        max_reach = 0
        for i, num in enumerate(nums):
            if i > max_reach:
                return False
            max_reach = max(max_reach, i + num)
        return True
```

---

56. Merge Intervals

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
```

Example 2:

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Constraints:

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= start_i <= end_i <= 10^4`

```
class Solution:
    def merge(self, interval: List[List[int]]) -> List[List[int]]:
        interval.sort(key = lambda x:x[0])
        n = len(interval)
        i = 0
        ans = []
        while i <= n-1:
            start = interval[i][0]
            end = interval[i][1]
            while i < n-1 and end >= interval[i+1][0]:
                end = max(end,interval[i+1][1])
                start = min(start,interval[i+1][0])
                i+=1
            ans.append([start,end])
            i+=1
        return ans

# from typing import List

# class Solution:
#     def merge(self, intervals: List[List[int]]) -> List[List[int]]:
#         if not intervals:
#             return []

#         intervals.sort(key=lambda x: x[0])
#         merged = [intervals[0]]

#         for curr in intervals[1:]:
#             last = merged[-1]
#             if curr[0] <= last[1]:  # Overlap
#                 last[1] = max(last[1], curr[1])
#             else:
#                 merged.append(curr)

#         return merged
```

57. Insert Interval

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [start`$_i$`, end`$_i$`]` represent the start and the end of the `i`th interval and `intervals` is sorted in ascending order by `start`$_i$. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by $start_i$ and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` *after the insertion*.

Note that you don't need to modify `intervals` in-place. You can make a new array and return it.

Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
```

Example 2:

```
Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
```

Constraints:

- `0 <= intervals.length <=` 10^4
- `intervals[i].length == 2`
- `0 <=` $start_i$ `<=` end_i `<=` 10^5
- `intervals` is sorted by $start_i$ in **ascending** order.
- `newInterval.length == 2`
- `0 <= start <= end <=` 10^5

```
from typing import List

class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[Li
st[int]]:
        result = []
        i = 0
        n = len(intervals)

        # Step 1: Add all intervals before newInterval
        while i < n and intervals[i][1] < newInterval[0]:
            result.append(intervals[i])
            i += 1

        # Step 2: Merge all overlapping intervals with newInterval
        while i < n and intervals[i][0] <= newInterval[1]:
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1
        result.append(newInterval)

        # Step 3: Add all remaining intervals
        while i < n:
            result.append(intervals[i])
            i += 1

        return result
```

---

135. Candy

There are `n` children standing in a line. Each child is assigned a rating value given in the integer array `ratings`.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

Return *the minimum number of candies you need to have to distribute the candies to the children*.

Example 1:

```
Input: ratings = [1,0,2]
Output: 5
Explanation: You can allocate to the first, second and third child with 2, 1, 2 candi
```

Example 2:

```
Input: ratings = [1,2,2]
Output: 4
Explanation: You can allocate to the first, second and third child with 1, 2, 1 candi
The third child gets 1 candy because it satisfies the above two conditions.
```

Constraints:

- `n == ratings.length`
- `1 <= n <= 2 * 10^4`
- `0 <= ratings[i] <= 2 * 10^4`

```
class Solution:
    def candy(self, ratings: List[int]) -> int:
        n = len(ratings)
        candies = [1] * n

        # Left to right
        for i in range(1, n):
            if ratings[i] > ratings[i - 1]:
                candies[i] = candies[i - 1] + 1

        # Right to left
        for i in range(n - 2, -1, -1):
            if ratings[i] > ratings[i + 1]:
                candies[i] = max(candies[i], candies[i + 1] + 1)

        return sum(candies)
```

---

435. Non-overlapping Intervals

Given an array of intervals `intervals` where `intervals[i] = [start_i, end_i]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Note that intervals which only touch at a point are **non-overlapping**. For example, `[1, 2]` and `[2, 3]`

are non-overlapping.

Example 1:

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.
```

Example 2:

```
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overl
```

Example 3:

```
Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-
```

Constraints:

- $1 <= intervals.length <= 10^5$
- $intervals[i].length == 2$
- $-5 * 10^4 <= start_i < end_i <= 5 * 10^4$

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key = lambda x:x[1])
        if not intervals:return []
        merged = [intervals[0]]
        remove = 0
        for cur in intervals[1:]:
            last = merged[-1]
            if cur[0] < last[-1]:
                print(cur)
                remove +=1
            else:
                merged.append(cur)
        return  remove
```

455. Assign Cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child `i` has a greed factor `g[i]`, which is the minimum size of a cookie that the child will be content with; and each cookie `j` has a size `s[j]`. If `s[j] >= g[i]`, we can assign the cookie `j` to the child `i`, and the child `i` will be content. Your goal is to maximize the number of your content children and output the maximum number.

Example 1:

```
Input: g = [1,2,3], s = [1,1]
Output: 1
Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1
And even though you have 2 cookies, since their size is both 1, you could only make t
You need to output 1.
```

Example 2:

```
Input: g = [1,2], s = [1,2,3]
Output: 2
Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1
You have 3 cookies and their sizes are big enough to gratify all of the children,
You need to output 2.
```

Constraints:

- `1 <= g.length <= 3 * 10^4`
- `0 <= s.length <= 3 * 10^4`
- `1 <= g[i], s[j] <= 2^31 - 1`

Note: This question is the same as 2410: Maximum Matching of Players With Trainers. (https://leetcode.com/problems/maximum-matching-of-players-with-trainers/description/)

```
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        i = 0
        j = 0
        ans = 0
        g.sort()
        s.sort()
        while i<len(g) and j < len(s):
            if g[i] <= s[j]:
                ans+=1
                i+=1
            j+=1
        return ans
```

---

678. Valid Parenthesis String

Given a string `s` containing only three types of characters: `'('`, `')'` and `'*'`, return `true` *if* `s` *is* ***valid***.

The following rules define a **valid** string:

- Any left parenthesis `'('` must have a corresponding right parenthesis `')'`.
- Any right parenthesis `')'` must have a corresponding left parenthesis `'('`.
- Left parenthesis `'('` must go before the corresponding right parenthesis `')'`.
- `'*'` could be treated as a single right parenthesis `')'` or a single left parenthesis `'('` or an empty string `""`.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "(*)"
Output: true
```

Example 3:

```
Input: s = "(*))"
Output: true
```

Constraints:

- `1 <= s.length <= 100`
- `s[i]` is `'('`, `')'` or `'*'`.

```
class Solution:
    def checkValidString(self, s: str) -> bool:
        low = 0   # Min number of open brackets
        high = 0  # Max number of open brackets

        for ch in s:
            if ch == '(':
                low += 1
                high += 1
            elif ch == ')':
                low -= 1
                high -= 1
            else:  # '*'
                low -= 1   # could be ')'
                high += 1  # could be '('

            if high < 0:
                return False  # Too many closing ')'

            if low < 0:
                low = 0  # We can't have less than 0 open brackets

        return low == 0
```

---

860. Lemonade Change

At a lemonade stand, each lemonade costs `$5`. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a `$5`, `$10`, or `$20` bill. You must provide the correct change to each customer so that the net transaction is that the customer pays `$5`.

Note that you do not have any change in hand at first.

Given an integer array `bills` where `bills[i]` is the bill the `i`th customer pays, return `true` *if you can provide every customer with the correct change, or* `false` *otherwise.*

Example 1:

```
Input: bills = [5,5,5,10,20]
Output: true
Explanation:
From the first 3 customers, we collect three $5 bills in order.
From the fourth customer, we collect a $10 bill and give back a $5.
From the fifth customer, we give a $10 bill and a $5 bill.
Since all customers got correct change, we output true.
```

Example 2:

```
Input: bills = [5,5,10,10,20]
Output: false
Explanation:
From the first two customers in order, we collect two $5 bills.
For the next two customers in order, we collect a $10 bill and give back a $5 bill.
For the last customer, we can not give the change of $15 back because we only have tw
Since not every customer received the correct change, the answer is false.
```

Constraints:

- `1 <= bills.length <= 10`^{5}
- `bills[i]` is either `5`, `10`, or `20`.

```
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        five, ten = 0, 0

        for bill in bills:
            if bill == 5:
                five += 1
            elif bill == 10:
                if five == 0:
                    return False
                five -= 1
                ten += 1
            else:  # bill == 20
                if ten > 0 and five > 0:
                    ten -= 1
                    five -= 1
                elif five >= 3:
                    five -= 3
                else:
                    return False
        return True
```
