4. Greedy Algorithm Notes

Solutions to 8 Leetcode problems with Greedy approach.

#greedy-algorithms

1. Maximum Subarray

Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`

Code

```
class Solution:
    def maxSubArray(self, nums):
        maxSub = nums[0]
        curSum = 0
        for n in nums:
            if curSum < 0:
                curSum = 0
            curSum+=n
            maxSub = max(maxSub, curSum)
        return maxSub

s = Solution()
array = [-2,1,-3,4,-1,2,1,-5,4]
result = s.maxSubArray(array)
print(result)
```

Explanation

```
class Solution:
        def maxSubArray(self, [-2,1,-3,4,-1,2,1,-5,4])
        maxSub = -2
        curSum = 0
        for n in nums:
                #iteration 1
                n = -2
                if 0 < 0 --> False
                curSum+=-2 >> curSum = -2
                maxSub = max(-2, -2) >> -2
                #iteration 2
```

```
n = 1
if -2<0 --> True
        curSum = 0
curSum+=1 >> curSum = 1
maxSub = max(-2, 1) >> 1
#iteration 3
n = -3
if 1<0 --> False
curSum+=-3 >> -2
maxSub = max(1, -2) >> 1
#iteration 4
n = 4
if -2  < 0 --> True:
        curSum = 0
curSum+=4 >> 4
maxSub = max(1, 4) >> 4
#iteration 5
n = -1
if 4 < 0 --> False
curSum+=-1 >> 3
maxSub = max(4, 3) >> 4
#iteration 6
n = 2
if 3<0 --> False
curSum += 2 >> 5
maxSub = max(5, 4) >> 5
#iteration 7
n = 1
if 5<0 --> False
curSum+=1 >> 6
maxSub = max(6, 5) >> 6
#iteration 8
n = -5
if 6<0 --> False
curSum+=-5 >> 1
maxSub = max(1, 6) >> 6
#iteration 9
n = 4
if 1<0 --> False
curSum += 1 >> 5
maxSub = max(5, 6) >> 6
return 6
```

2. 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 length is 0, which makes it impossible to reach the last index.

Constraints:

- `1 <= nums.length <= 104`
- `0 <= nums[i] <= 105`

Code

```
class Solution:
    def canJump(self, nums):
```

```
        goal = len(nums) -1

        for i in range(len(nums)-1, -1, -1):
            if i + nums[i] >= goal:
                goal = i
        return True if goal == 0 else False

s = Solution()
array = [2,3,1,1,4]
result = s.canJump(array)
print(result)
```

Explanation

```
def canJump(self, [2,3,1,1,4]):
        goal = 5-1 >> 4
        for i in range(4, -1, -1):
                #iteration 1
                i = 4
                if 8 >= 4:
                        goal = 4
                #iteration 2
                i = 3
                if 4 >= 4:
                        goal = 3
                #iteration 3
                i = 2
                if 3 >= 3:
                        goal = 2
                #iteration 4
                i = 1
                if 4 >= 2:
                        goal = 1
                #iteration 5
                i = 0
                if 2 >= 1:
                        goal = 0
        return True
```

3. 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 from index 0 to 1, then 3 steps to the last index.

Example 2:

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

Constraints:

- `1 <= nums.length <= 104`
- `0 <= nums[i] <= 1000`
- It's guaranteed that you can reach `nums[n - 1]`.

Code

```python
class Solution(object):
    def jump(self, nums):
        res = 0
        l = r = 0
        while r<len(nums)-1:
            farthest = 0
            for i in range(l, r+1):
                farthest = max(farthest, i+nums[i])
            l = r+1
            r = farthest
            res+=1
        return res


s = Solution()
array = [2,3,1,1,4]
result = s.jump(array)
print(result)
```

Explanation

```python
def jump(self, [2, 3, 1, 1, 4]:
        while 0<4:
                farthest = 0
                for i in range(0, 1):
                        farthest = max(0, 0 + 2) >> 2
                l = 1
                r = 2
                res+=1 >> 1

                farthest = 0
                for i in range(1, 3):
                        farthest = max(0, 1 + 3) >> 4
                        farthest = max(4, 2 + 1) >> 4
                l = 3
                r = 4
                res+=1 >> 2
        return 2
```

[4. Gas Station]

There are `n` gas stations along a circular route, where the amount of gas at the `ith` station is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from the `ith` station to its next `(i + 1)th` station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays `gas` and `cost`, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* `-1`. If there exists a solution, it is **guaranteed** to be **unique**

Example 1:

Input: `gas = [1,2,3,4,5], cost = [3,4,5,1,2]`

`cost[i]` is the expenditure of gas to move to the `gas[i+1]` from `gas[i]`
Output: `3`

Explanation:

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.

Example 2:

Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.

Constraints:

- `n == gas.length == cost.length`
- `1 <= n <= 105`
- `0 <= gas[i], cost[i] <= 104`

Code

```
class Solution(object):
    def canCompleteCircuit(self, gas, cost):
        if sum(gas)<sum(cost):
            return -1
        total = 0
        start = 0
        for i in range(len(gas)):
            total += gas[i]-cost[i]
            if total < 0:
                total = 0
                start = i + 1
        return start

if __name__ == '__main__':
    gas = [1,2,3,4,5]
    cost = [3,4,5,1,2]
    s = Solution()
    result = s.canCompleteCircuit(gas, cost)
    print(result)
```

Explanation

```
def canCompleteCircuit(self, [1,2,3,4,5], [3,4,5,1,2]):
if sum([1,2,3,4,5])<sum([3,4,5,1,2]): --> False
total = 0
start = 0
for i in range(0, 5):
        # i = 0
        total += -2 >> -2
        if total<0: --> True
                total = 0
                start = 0+1 >> 1
        # i = 1
        total += -2 >> -2
        if total<0: --> True
                total = 0
                start = 1+1 >> 2
        # i = 2
        total += -2 >> -2
        if total<0: --> True
                total = 0
                start = 2+1 >> 3
        # i = 3
        total += 3 >> 3
        if total<0: --> False
        # i = 4
        total += 3 >> 6
```

```
                if total<0: --> False
        return 3
```

5. Hand of Straights

Alice has some number of cards and she wants to rearrange the cards into groups so that each group is of size `groupSize`, and consists of `groupSize` consecutive cards.

Given an integer array `hand` where `hand[i]` is the value written on the `ith` card and an integer `groupSize`, return `true` if she can rearrange the cards, or `false` otherwise.

Example 1:

Input: `hand = [1,2,3,6,2,3,4,7,8], groupSize = 3`
Output: true
Explanation: Alice's hand can be rearranged as `[1,2,3],[2,3,4],[6,7,8]`

Example 2:

Input: `hand = [1,2,3,4,5], groupSize = 4`
Output: false
Explanation: Alice's hand can not be rearranged into groups of 4.

Constraints:

- `1 <= hand.length <= 104`
- `0 <= hand[i] <= 109`
- `1 <= groupSize <= hand.length`

Code

```
import heapq
class Solution(object):
    def isNStraightHand(self, hand, groupSize):
        if len(hand) % groupSize:
            return False
        count = {}
        for n in hand:
            count[n] = 1 + count.get(n, 0)
        minH = list(count.keys())
        heapq.heapify(minH)
        while minH:
            first = minH[0]
            for i in range(first, first+groupSize):
                if i not in count:
                    return False
                count[i] -= 1
                if count[i] == 0:
                    if i!= minH[0]:
                        return False
                    heapq.heappop(minH)
        return True

if __name__ == '__main__':
    hand = [1,2,3,6,2,3,4,7,8]
    groupSize = 3
    s = Solution()
    result = s.isNStraightHand(hand, groupSize)
    print(result)
```

Explanation

```
def isNStraightHand(self, [1, 2, 3, 6, 2, 3, 4, 7, 8], 3):
        if len(hand) % groupSize: --> False
        count = {}
        for n in [1, 2, 3, 6, 2, 3, 4, 7, 8]:
```

```
                    count[1] = 1
                    count[2] = 1
                    count[3] = 1
                    count[6] = 1
                    count[2] = 2
                    count[3] = 2
                    count[4] = 1
                    count[7] = 1
                    count[8] = 1
    count = {1: 1, 2: 2, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}
    minH = [1, 2, 3, 6, 4, 7, 8]
    heapq.heapify([1, 2, 3, 6, 4, 7, 8]) >> [1, 2, 3, 6, 4, 7, 8]

    while [1, 2, 3, 6, 4, 7, 8]:
                    first = 1
                    for i in range(1, 4):
                                    if 1 not in {1: 1, 2: 2, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                                    count[1] -= 1   >> {1: 0, 2: 2, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}
                                    if count[1] == 0: --> True
                                                    if 1!= 1: --> False
                                                    heapq.heappop([1, 2, 3, 6, 4, 7, 8]) >> minH=[2, 4, 3, 6, 8, 7]

                                    if 2 not in {1: 0, 2: 2, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                                    count[2] -= 1   >> {1: 0, 2: 1, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}
                                    if count[2] == 0: --> False
                    if 3 not in {1: 0, 2: 1, 3: 2, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                    count[3] -= 1   >> {1: 0, 2: 1, 3: 1, 6: 1, 4: 1, 7: 1, 8: 1}
                    if count[3] == 0: --> False

    first = 2
    for i in range(2, 5):
                    if 2 not in {1: 0, 2: 1, 3: 1, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                    count[2] -= 1   >> {1: 0, 2: 0, 3: 1, 6: 1, 4: 1, 7: 1, 8: 1}
                    if count[2] == 0: --> True
                                    if 2!= 2: --> False
                                    heapq.heappop([2, 4, 3, 6, 8, 7]) >> minH=[3, 4, 7, 6, 8]

                    if 3 not in {1: 0, 2: 0, 3: 1, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                    count[3] -= 1   >> {1: 0, 2: 0, 3: 0, 6: 1, 4: 1, 7: 1, 8: 1}
                    if count[3] == 0: --> True
                                    if 3!= 3: --> False
                                    heapq.heappop([3, 4, 7, 6, 8]) >> minH=[4, 6, 7, 8]

                    if 4 not in {1: 0, 2: 0, 3: 0, 6: 1, 4: 1, 7: 1, 8: 1}: --> False
                    count[4] -= 1   >> {1: 0, 2: 0, 3: 0, 6: 1, 4: 0, 7: 1, 8: 1}
                    if count[4] == 0: --> True
                                    if 4!= 4: --> False
                                    heapq.heappop([4, 6, 7, 8]) >> minH=[6, 8, 7]

    first = 6
    for i in range(6, 9):
                                    if 6 not in {1: 0, 2: 0, 3: 0, 6: 1, 4: 0, 7: 1, 8: 1}: --> False
                                    count[6] -= 1   >> {1: 0, 2: 0, 3: 0, 6: 0, 4: 0, 7: 1, 8: 1}
                                    if count[6] == 0: --> True
                                                    if 6!= 6: --> False
                                                    heapq.heappop([6, 8, 7]) >> minH=[7, 8]

                                    if 7 not in {1: 0, 2: 0, 3: 0, 6: 0, 4: 0, 7: 1, 8: 1}: --> False
                                    count[7] -= 1   >> {1: 0, 2: 0, 3: 0, 6: 0, 4: 0, 7: 0, 8: 1}
                                    if count[7] == 0: --> True
                                                    if 7!= 7: --> False
                                                    heapq.heappop([7, 8]) >> minH=[8]

                                    if 8 not in {1: 0, 2: 0, 3: 0, 6: 0, 4: 0, 7: 0, 8: 1}: --> False
                                    count[8] -= 1   >> {1: 0, 2: 0, 3: 0, 6: 0, 4: 0, 7: 0, 8: 0}
                                    if count[8] == 0: --> True
                                                    if 8!= 8: --> False
                                                    heapq.heappop([8]) >> minH=[]
        return True
```

6. Merge Triplets to Form Target Triplet

A **triplet** is an array of three integers. You are given a 2D integer array `triplets`, where `triplets[i] = [ai, bi, ci]` describes the `ith` **triplet**. You are also given an integer array `target = [x, y, z]` that describes the **triplet** you want to obtain.

To obtain `target`, you may apply the following operation on `triplets` **any number** of times (possibly **zero**):

- Choose two indices (**0-indexed**) `i` and `j` (`i != j`) and **update** `triplets[j]` to become `[max(ai, aj), max(bi, bj), max(ci, cj)]`.
  - For example, if `triplets[i] = [2, 5, 3]` and `triplets[j] = [1, 7, 5]`, `triplets[j]` will be updated to `[max(2, 1), max(5, 7), max(3, 5)] = [2, 7, 5]`.

Return `true` *if it is possible to obtain the* `target` ***triplet*** `[x, y, z]` *as an* ***element*** *of* `triplets`*, or* `false` *otherwise*.

Example 1:

Input: `triplets = [[2,5,3],[1,8,4],[1,7,5]], target = [2,7,5]`
Output: true
Explanation: Perform the following operations:
- `Choose the first and last triplets [[2,5,3],[1,8,4],[1,7,5]]. Update the last triplet to be [max(2,1), max(5,7), max(3,5)] = [2,7,5]. triplets = [[2,5,3],[1,8,4],[2,7,5]]`
The target triplet `[2,7,5]` is now an element of triplets.

Example 2:

Input: `triplets = [[3,4,5],[4,5,6]], target = [3,2,5]`
Output: false
Explanation: It is impossible to have `[3,2,5]` as an element because there is no 2 in any of the triplets.

Example 3:

Input: triplets = `[[2,5,3],[2,3,4],[1,2,5],[5,2,3]], target = [5,5,5]`
Output: `true`
Explanation: Perform the following operations:

- Choose the first and third triplets `[[2,5,3],[2,3,4],[1,2,5],[5,2,3]]`. Update the third triplet to be `[max(2,1), max(5,2), max(3,5)] = [2,5,5]`. triplets = `[[2,5,3],[2,3,4],[2,5,5],[5,2,3]].`
- Choose the third and fourth triplets `[[2,5,3],[2,3,4],[2,5,5],[5,2,3]]`. Update the fourth triplet to be `[max(2,5), max(5,2), max(5,3)] = [5,5,5]`. triplets = `[[2,5,3],[2,3,4],[2,5,5],[5,5,5]]`
  The target triplet [5,5,5] is now an element of triplets.

Constraints:

- `1 <= triplets.length <= 105`
- `triplets[i].length == target.length == 3`
- `1 <= ai, bi, ci, x, y, z <= 1000`

Code

```
class Solution(object):
        def mergeTriplets(self, tiplets, target):
                good = set()
                for t in triplets:
                        if t[0] > target[0] or t[1] > target[1] or t[2] > target[2]:
                                continue
                        for i, v in enumerate(t):
                                if v == target[i]:
                                        good.add(i)
                return len(good) == 3

if __name__ == '__main__':
    s = Solution()
    triplets = [[2,5,3],[1,8,4],[1,7,5]]
    targets = [[2,7,5],[3,2,5]]
    for target in targets:
        result = s.mergeTriplets(triplets, target)
        print(target, result)
```

Explanation

```
def mergeTriplets(self, [[2, 5, 3], [1, 8, 4], [1, 7, 5]], [2, 7, 5]):
        good = set()
        for t in [[2, 5, 3], [1, 8, 4], [1, 7, 5]]:
                if t[0] > target[0] --> False or t[1] > target[1] --> False or t[2] > target[2] --> False:
                for i, v in enumerate([2, 5, 3]):
                        if v == target[i]: -->True
                                good.add(0)
                                good = {0}
                        if v == target[i]: -->False
                        if v == target[i]: -->False
                if t[0] > target[0] --> False or t[1] > target[1] --> True or t[2] > target[2] --> False:
                        continue
                if t[0] > target[0] --> False or t[1] > target[1] --> False or t[2] > target[2] --> False:
                for i, v in enumerate([1, 7, 5]):
                        if v == target[i]: -->False
                        if v == target[i]: -->True
                                good.add(1)
                                good = {0, 1}
                        if v == target[i]: -->True
                                good.add(2)
                                good = {0, 1, 2}
        return True
# Output: [2, 7, 5] True
```

```
def mergeTriplets(self, [[2, 5, 3], [1, 8, 4], [1, 7, 5]], [3, 2, 5]):
        good = set()
        for t in [[2, 5, 3], [1, 8, 4], [1, 7, 5]]:
                if t[0] > target[0] --> False or t[1] > target[1] --> True or t[2] > target[2] --> False:
                        continue
                if t[0] > target[0] --> False or t[1] > target[1] --> True or t[2] > target[2] --> False:
                        continue
                if t[0] > target[0] --> False or t[1] > target[1] --> True or t[2] > target[2] --> False:
                        continue
        return False

#Output: [3, 2, 5] False
```

7. Partition Labels

You are given a string `s`. We want to partition the string into as many parts as possible so that each letter appears in at most one part.

Note that the partition is done so that after concatenating all the parts in order, the resultant string should be `s`.

Return *a list of integers representing the size of these parts*.

Example 1:

Input: `s = "ababcbacadefegdehijhklij"`
Output: `[9,7,8]`
Explanation:
The partition is "ababcbaca", "defegde", "hijhklij".
This is a partition so that each letter appears in at most one part.
A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits s into less parts.

Example 2:

Input: s = "eccbbbbdec"
Output: [10]

Constraints:

- `1 <= s.length <= 500`
- `s` consists of lowercase English letters.

Code

```
class Solution:
    def partitionLabels(self, s):
        # Create a dictionary to store the last index of each character in the string
        last_index = {v: i for i, v in enumerate(s)}
```

```
        res = []  # Initialize an empty list to store the partition lengths
        size, end = 0, 0

        # Iterate through the string
        for i, c in enumerate(s):
            size += 1
            end = max(end, last_index[c])

            # If we reach the end of the current partition, append its size to the result
            if i == end:
                res.append(size)
                size = 0

        return res

if __name__ == '__main__':
    solution = Solution()
    s = "ababcbacadefegdehijhklij"
    result = solution.partitionLabels(s)
    print(result)
```

Explanation

```
def partitionLabels(self, ababcbacadefegdehijhklij):
        lastIndex = {
             "a": 8,
         "b": 5,
         "c": 7,
         "d": 14,
         "e": 15,
         "f": 11,
         "g": 13,
         "h": 19,
         "i": 22,
         "j": 23,
         "k": 20,
         "l": 21
         }
        res = []
        size, end = 0, 0
        for i, c in enumerate(ababcbacadefegdehijhklij):
                #iteration 0
                size += 1 >> 1
                end = max(0, 8) >> 8
                if i == end / if 0 == 8: --> False


                #iteration 1
                size += 1 >> 2
                end = max(8, 5) >> 8
                if i == end / if 1 == 8: --> False


                #iteration 2
                size += 1 >> 3
                end = max(8, 8) >> 8
                if i == end / if 2 == 8: --> False


                #iteration 3
                size += 1 >> 4
                end = max(8, 5) >> 8
                if i == end / if 3 == 8: --> False


                #iteration 4
                size += 1 >> 5
                end = max(8, 7) >> 8
```

```
if i == end / if 4 == 8: --> False


#iteration 5
size += 1 >> 6
end = max(8, 5) >> 8
if i == end / if 5 == 8: --> False


#iteration 6
size += 1 >> 7
end = max(8, 8) >> 8
if i == end / if 6 == 8: --> False


#iteration 7
size += 1 >> 8
end = max(8, 7) >> 8
if i == end / if 7 == 8: --> False


#iteration 8
size += 1 >> 9
end = max(8, 8) >> 8
if i == end / if 8 == 8: --> True
        [].append(9) >> [9]
        size = 0


#iteration 9
size += 1 >> 1
end = max(8, 14) >> 14
if i == end / if 9 == 14: --> False


#iteration 10
size += 1 >> 2
end = max(14, 15) >> 15
if i == end / if 10 == 15: --> False


#iteration 11
size += 1 >> 3
end = max(15, 11) >> 15
if i == end / if 11 == 15: --> False


#iteration 12
size += 1 >> 4
end = max(15, 15) >> 15
if i == end / if 12 == 15: --> False


#iteration 13
size += 1 >> 5
end = max(15, 13) >> 15
if i == end / if 13 == 15: --> False


#iteration 14
size += 1 >> 6
end = max(15, 14) >> 15
if i == end / if 14 == 15: --> False


#iteration 15
size += 1 >> 7
end = max(15, 15) >> 15
if i == end / if 15 == 15: --> True
        [9].append(7) >> [9, 7]
        size = 0


#iteration 16
size += 1 >> 1
end = max(15, 19) >> 19
```

```
            if i == end / if 16 == 19: --> False


            #iteration 17
            size += 1 >> 2
            end = max(19, 22) >> 22
            if i == end / if 17 == 22: --> False


            #iteration 18
            size += 1 >> 3
            end = max(22, 23) >> 23
            if i == end / if 18 == 23: --> False


            #iteration 19
            size += 1 >> 4
            end = max(23, 19) >> 23
            if i == end / if 19 == 23: --> False


            #iteration 20
            size += 1 >> 5
            end = max(23, 20) >> 23
            if i == end / if 20 == 23: --> False


            #iteration 21
            size += 1 >> 6
            end = max(23, 21) >> 23
            if i == end / if 21 == 23: --> False


            #iteration 22
            size += 1 >> 7
            end = max(23, 22) >> 23
            if i == end / if 22 == 23: --> False


            #iteration 23
            size += 1 >> 8
            end = max(23, 23) >> 23
            if i == end / if 23 == 23: --> True
                    [9, 7].append(8) >> [9, 7, 8]
                    size = 0

    return [9, 7, 8]
```

---

8. 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 `'*'`.

Code

```
class Solution:

    def checkValidString(self, s):

        leftMin, leftMax = 0, 0

        for c in s:

            if c == "(":

                leftMin, leftMax = leftMin+1, leftMax+1

            elif c == ")":

                leftMin, leftMax = leftMin - 1, leftMax -1

            else:

                leftMin, leftMax = leftMin - 1 , leftMax + 1

            if leftMax < 0:

                return False

            if leftMin < 0:

                leftMin = 0

        return leftMin == 0


if __name__ == '__main__':

    solution = Solution()

    s = "(*)("

    result = solution.checkValidString(s)

    print(result)
```

Explanation

```
def checkValidString(self, "(*)("):
        leftMin, leftMax = 0, 0
        for ( in (*):
                if ( == "(": --> True
                        leftMin, leftMax = 0+1, 0+1      >> leftMin = 1, leftMax = 1

                #Second if else block

        for * in (*):
                 else:
                        leftMin, leftMax = 1-1, 1+1      >> leftMin = 0, leftMax = 2

                #Second if else block

        for ) in (*):
```

```
            elif c == ")":
                    leftMin, leftMax = 0-1, 2-1      >> leftMin = -1, leftMax = 1

            #Second if else block

            if leftMin=-1 < 0: --> True
                    leftMin=0
    return leftMin == 0 --> True
```

End