Notebook

August 6, 2024

[ ]:

```
from typing import List
```

[ ]:

```
##### Median of Two Sorted Arrays using binary search
##### https://leetcode.com/problems/median-of-two-sorted-arrays/description/

import sys

class Solution:
    def findMedianSortedArrays(self, nums1: list[int], nums2: list[int]) ->␣
 ↪float:
        m, n = len(nums1), len(nums2)

        if n < m:
            return self.findMedianSortedArrays(nums2, nums1)

        low, high = 0, m

        while low <= high:
            x_part = (low + high) // 2
            y_part = (m + n + 1) // 2 - x_part

            maxLeftX = -sys.maxsize - 1 if x_part == 0 else nums1[x_part - 1]
            minRightX = sys.maxsize if x_part == m else nums1[x_part]
            maxLeftY = -sys.maxsize - 1 if y_part == 0 else nums2[y_part - 1]
            minRightY = sys.maxsize if y_part == n else nums2[y_part]

            if maxLeftX <= minRightY and maxLeftY <= minRightX:
                if (m + n) % 2 == 0:
                    return (max(maxLeftX, maxLeftY) + min(minRightX,␣
 ↪minRightY)) / 2.0
                else:
                    return float(max(maxLeftX, maxLeftY))
            elif maxLeftX > minRightY:
                high = x_part - 1
            else:
                low = x_part + 1
        raise ValueError("The input arrays are not valid for finding the median.
↪")
```

[ ]:

```
# https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/
↪description/
# find the minimum element in a rotated sorted array

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums)-1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] > nums[right]:
                left = mid+1
            else:
                right = mid
        return nums[left]

# follow up to above question
# https://leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/
↪description/
# find the minimum element in a rotated sorted array with duplicates

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums)-1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] > nums[right]:
                left = mid+1
            elif nums[mid] < nums[right]:
                right = mid
            else:
                right -= 1 # when mid and right are the same, we can't tell␣
↪which side the minimum is on, so we just decrement right
        return nums[left]

# https://leetcode.com/problems/search-in-rotated-sorted-array/
# search for a target in a rotated sorted array (no duplicates)

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums)-1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] > nums[right]:
                left = mid+1
            else:
                right = mid

        rot = left
        left, right = 0, len(nums)-1

        while left <= right:
            mid = (left + right) // 2
            realmid = (mid + rot) % (len(nums))
            if nums[realmid] == target: return realmid
            if nums[realmid] < target: left = mid+1
            else: right = mid-1

        return -1


# https://leetcode.com/problems/search-in-rotated-sorted-array-ii/
# search for a target in a rotated sorted array with duplicates

class Solution:
    def search(self, nums, target):
        l, r = 0, len(nums)-1
        while l <= r:
            mid = l + (r-l)//2
            if nums[mid] == target:
                return True
            while l < mid and nums[l] == nums[mid]: # tricky part
                l += 1
            # the first half is ordered
            if nums[l] <= nums[mid]:
                # target is in the first half
                if nums[l] <= target < nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            # the second half is ordered
            else:
                # target is in the second half
                if nums[mid] < target <= nums[r]:
                    l = mid + 1
                else:
                    r = mid - 1
        return False
```

[ ]:

```
# https://leetcode.com/problems/minimize-the-maximum-difference-of-pairs/

"""
```

```
Input: nums = [10,1,2,7,1,3], p = 2
Output: 1
Explanation: The first pair is formed from the indices 1 and 4, and the second␣
 ↪pair is formed from the indices 2 and 5.
The maximum difference is max(|nums[1] - nums[4]|, |nums[2] - nums[5]|) =␣
 ↪max(0, 1) = 1. Therefore, we return 1.
"""

class Solution:
    def minimizeMax(self, nums: List[int], p: int) -> int:
        nums.sort()

        def check(nums, mid, p):
            i = 1
            while i < len(nums):
                if nums[i]-nums[i-1] <= mid:
                    p -= 1
                    i += 1
                    if p <= 0 : return True
                i += 1
            return p <= 0

        l, r = 0, nums[-1] - nums[0]
        ans = -1
        while l <= r:
            mid = l + (r - l) // 2
            if check(nums, mid, p):
               # print(mid)
                ans = mid
                r = mid-1
            else:
                l = mid+1
        return ans
```

```
[ ]: # https://leetcode.com/problems/house-robber-iv/solutions/3143741/
      ↪binary-search-c-with-similar-problems/
     # Minimum of maximum
     """
     Input: nums = [2,7,9,3,1], k = 2
     Output: 2
     Explanation: There are 7 ways to rob the houses. The way which leads to minimum␣
      ↪capability is to rob the house at index 0 and 4. Return max(nums[0],␣
      ↪nums[4]) = 2.
     """

     class Solution:
         def minCapability(self, nums: List[int], k: int) -> int:
```

```
        def helper(nums, mid, k):
            i, n = 0, len(nums)
            while i < n:
                if nums[i] <= mid:
                    k -= 1
                    i += 2
                else:
                    i += 1
                if k == 0: return True
            return k <= 0
        l, h = 0, 1000000000
        while l < h:
            mid = l + (h - l) // 2
            if (helper(nums, mid, k)):
                h = mid
            else:
                l = mid+1
        return l
```

[ ]:

```
# https://leetcode.com/problems/find-peak-element/description/
# return any peak element which is greater than its neighbors

# use binary search to find the peak element
# Time complexity: O(logn)
class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        left, right = 0, len(nums)-1
        while left < right-1:
            mid = (left + right) // 2
            if nums[mid] > nums[mid+1] and nums[mid] > nums[mid-1]:
                return mid

            if nums[mid] < nums[mid+1]: left = mid+1
            else: right = mid-1
        return left if nums[left] >= nums[right] else right


# https://leetcode.com/problems/find-a-peak-element-ii/description/
# return peak in 2D array matrix

# Time complexity: O(m*logn), m is the number of rows, n is the number of␣
 ↪columns
class Solution(object):
    def findPeakGrid(self, mat):
        startCol = 0
        endCol = len(mat[0])-1
        while startCol <= endCol:
            maxRow = 0
            midCol = (endCol+startCol)//2

            for row in range(len(mat)):
                maxRow = row if (mat[row][midCol] >= mat[maxRow][midCol]) else maxRow

            leftIsBig    =   midCol-1 >= startCol  and  mat[maxRow][midCol-1] > mat[maxRow][midCol]
            rightIsBig   =   midCol+1 <= endCol    and  mat[maxRow][midCol+1] > mat[maxRow][midCol]

            if (not leftIsBig) and (not rightIsBig):   # we have found the peak element
                return [maxRow, midCol]
            elif rightIsBig:              # if rightIsBig, then there is an element in 'right' that is bigger than all the elements in the 'midCol',
                startCol = midCol+1       # so 'midCol' cannot have 'peakPlane'
            else:                           # leftIsBig
                endCol = midCol-1

        return []


# https://leetcode.com/problems/peak-index-in-a-mountain-array/description/
# here peak where values are increasing and then decreasing arr[i] < arr[i+1] > arr[i+2]

class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 0, len(arr)-1
        while left < right:
            mid = (left + right) // 2
            if arr[mid] < arr[mid+1]:
                left = mid+1
            else:
                right = mid
        return left
```

[ ]:

```
# https://github.com/doocs/leetcode/blob/main/solution/2100-2199/2137.Pour%20Water%20Between%20Buckets%20to%20Make%20Water%20Levels%20Equal/README_EN.md
# Pour Water Between Buckets to Make Water Levels Equal
```

```
"""
Input: buckets = [1,2,7], loss = 80       Output: 2.00000
Explanation: Pour 5 gallons of water from buckets[2] to buckets[0].
5 * 80% = 4 gallons are spilled and buckets[0] only receives 5 - 4 = 1 gallon␣
 ↪of water.
All buckets have 2 gallons of water in them so return 2.

Input: buckets = [2,4,6], loss = 50      Output: 3.50000
Explanation: Pour 0.5 gallons of water from buckets[1] to buckets[0].
0.5 * 50% = 0.25 gallons are spilled and buckets[0] only receives 0.5 - 0.25 =␣
 ↪0.25 gallons of water.
Now, buckets = [2.25, 3.5, 6].
Pour 2.5 gallons of water from buckets[2] to buckets[0].
2.5 * 50% = 1.25 gallons are spilled and buckets[0] only receives 2.5 - 1.25 =␣
 ↪1.25 gallons of water.
All buckets have 3.5 gallons of water in them so return 3.5.
"""
# answer upto 1e-5 accepted
class Solution:
    def equalizeWater(self, buckets: List[int], loss: int) -> float:
        def check(v):
            a = b = 0
            for x in buckets:
                if x >= v:
                    a += x - v
                else:
                    b += (v - x) * 100 / (100 - loss)
            return a >= b

        l, r = 0, max(buckets)
        while r - l > 1e-5:
            mid = (l + r) / 2
            if check(mid):
                l = mid
            else:
                r = mid
        return l
```

[ ]:

```
### https://leetcode.com/problems/split-array-largest-sum/description/
#### split array into m subarrays such that sum of each subarray is minimized

class Solution:
    def splitArray(self, nums: List[int], m: int) -> int:

        def satisfy(nums, m, mid):
            currSum = 0
            currAns = 0
            for num in nums:
                if num > mid:
                    return False

                if (currSum + num > mid):
                    currAns += 1
                    currSum = num
                else:
                    currSum += num

            if (currSum == 0):
                return currAns <= m
            else:
                return currAns + 1 <= m

        l, h = 0, 1e9
        ans = 0

        while l <= h:
            mid = l + (h - l) // 2

            if satisfy(nums, m , mid):
                ans = mid
                h = mid - 1
            else:
                l = mid + 1

        return int(ans)
```

[ ]:

```
# https://leetcode.com/problems/3sum/description/
# https://leetcode.com/problems/4sum/description/

# k-sum to solve any k-sum problem
# sort the array
# start from 0 to n-1
# if k == 2 then use two pointer approach
# if k > 2 then use recursion to solve k-1 sum
# remove duplicates by checking if current element is same as previous element

class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:

        def recurr(start, end, k, target, curr_ans, ans):
            #print(start, end, k, target, curr_ans, ans)
            if start > end:
                return ans
            if k == 2:
                l, r = start, end
                while l < r:
                    cur_sum = nums[l] + nums[r]
                    if cur_sum > target:
                        r -= 1
                    elif cur_sum < target:
                        l += 1
                    else:
                        ans.append(curr_ans + [nums[l], nums[r]])
                        while l+1 < r and nums[l] == nums[l+1]:
                            l += 1
                        while r-1 >= 0 and nums[r] == nums[r-1]:
                            r -= 1
                        l += 1
                        r -= 1
            else:
                while start < end:
                    recurr(start+1, end, k-1, target-nums[start],␣
 ↪curr_ans+[nums[start]], ans)
                    while start+1 < end and nums[start] == nums[start+1]:
                        start += 1
                    start += 1

        def ksum(k):
            res = []
            nums.sort()
            recurr(0, len(nums)-1, k, target, [], res)
            return res

        return ksum(4)
```

[ ]:

```
#### https://leetcode.com/problems/
 ↪find-first-and-last-position-of-element-in-sorted-array/description/
#### find first and last position of element in sorted array

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:

        def search(x):
            lo, hi = 0, len(nums)
            while lo < hi:
                mid = (lo + hi) // 2
                if nums[mid] < x:
                    lo = mid+1
                else:
                    hi = mid
            return lo

        lo = search(target)
        hi = search(target+1)-1

        if lo <= hi:
            return [lo, hi]

        return [-1, -1]
```

[ ]:

```
#### https://leetcode.com/problems/koko-eating-bananas/description/


class Solution:
    def possible(self, p: list[int], m: int, h: int) -> bool:
        ans = 0

        for x in p:
            div = x // m
            rem = x % m
            if rem:
                ans += 1
            ans += div

        return ans <= h

    def minEatingSpeed(self, piles: list[int], h: int) -> int:
        l, r = 1, int(1e9)
        ans = int(1e9)

        while l < r:
            mid = l + (r - l) // 2
            if self.possible(piles, mid, h):
                ans = mid
                r = mid
            else:
                l = mid + 1

        return ans
```