Binary Search_3

@kbbhatt04

@July 2, 2023

Allocate minimum number of pages

You have N books, each with A[i] number of pages. M students need to be allocated contiguous books, with each student getting at least one book. Out of all the permutations, the goal is to find the permutation where the student with the most pages allocated to him gets the minimum number of pages, out of all possible permutations.

Note: Return -1 if a valid assignment is not possible, and allotment should be in contiguous order.

Input:
N = 4
A[] = {12,34,67,90}
M = 2
Output:113
Explanation:Allocation can be done in
following ways:
{12} and {34, 67, 90} Maximum Pages = 191
{12, 34} and {67, 90} Maximum Pages = 157
{12, 34, 67} and {90} Maximum Pages =113.
Therefore, the minimum of these cases is 113,
which is selected as the output.

- Approach
    - Brute-force
        - To give at least 1 book to each child, minimum pages per child = `max(arr)`
        - Likewise maximum number of pages for a child is = `sum(arr)` (Case where `M = 1`)
        - Thus we linear search our answer between `max(arr)` and `sum(arr)`
        - To count the students for each page_limit between `max(arr)` and `sum(arr)`, initialise `stud = 1` and `cur_stud_pages = 0`, then iterate over the array and increment the cur_stud_pages with arr[i] if ≤ page_limit else give it to next student (i.e. increment stud)
        - Time Complexity: $O(n * (sum - max + 1))$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search answer between `max(arr)` and `sum(arr)`
        - Time Complexity: $O(n * log(sum - max + 1))$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findPages(self, A, N, M):
        if M > N:   return -1

        def cntStudents(A, limit):
            stud = 1
            pages = 0
            for x in A:
                if pages + x <= limit:
                    pages += x
                else:
                    stud += 1
                    pages = x
            return stud
        l = max(A)
        r = sum(A)
        for i in range(l, r+1):
            cnt = cntStudents(A, i)
            if cnt <= M:
                return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def findPages(self, A, N, M):
        if M > N:   return -1

        def cntStudents(A, limit):
            stud = 1
            pages = 0
            for x in A:
                if pages + x <= limit:
                    pages += x
                else:
                    stud += 1
                    pages = x
            return stud

        l = max(A)
        r = sum(A)

        while l <= r:
            mid = (l + r) >> 1
            if cntStudents(A, mid) > M:
                l = mid + 1
            else:
                r = mid - 1
        return l
```

```
// C++
// Optimal Solution
class Solution
{
    public:
    int count_students(int A[], int N, int limit) {
        int stud_cnt = 1;
        int cur_stud_pages = 0;
        for (int x = 0; x < N; x++) {
            if (cur_stud_pages + A[x] <= limit) {
                cur_stud_pages += A[x];
            }
            else {
                stud_cnt++;
                cur_stud_pages = A[x];
            }
        }
        return stud_cnt;
    }

    int findPages(int A[], int N, int M)
    {
        if (M > N) {return -1;}
        int low = *max_element(A, A + N);
        int high = accumulate(A, A + N, 0);
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (count_students(A, N, mid) > M) {low = mid + 1;}
            else {high = mid - 1;}
        }
        return low;
    }
};
```

Split Array Largest Sum

Given an integer array `nums` and an integer `k`, split `nums` into `k` non-empty subarrays such that the largest sum of any subarray is **minimized**.

Return *the minimized largest sum of the split*.

- Approach
    - Brute-force
        - Answer could lie between max(nums) and sum(nums) as when `k == len(nums)`, max(nums) will be the answer and when `k == 1`, sum(nums) will be the answer respectively
        - Linear Search between this range, check if we can fit all elements in k subarrays and if we can, return limit as answer
        - Time Complexity: $O(n * (sum - max + 1))$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search between the range `max(nums)` and `sum(nums)`
        - Time Complexity: $O(n * log(sum - max + 1))$
        - Space Complexity: $O(1)$

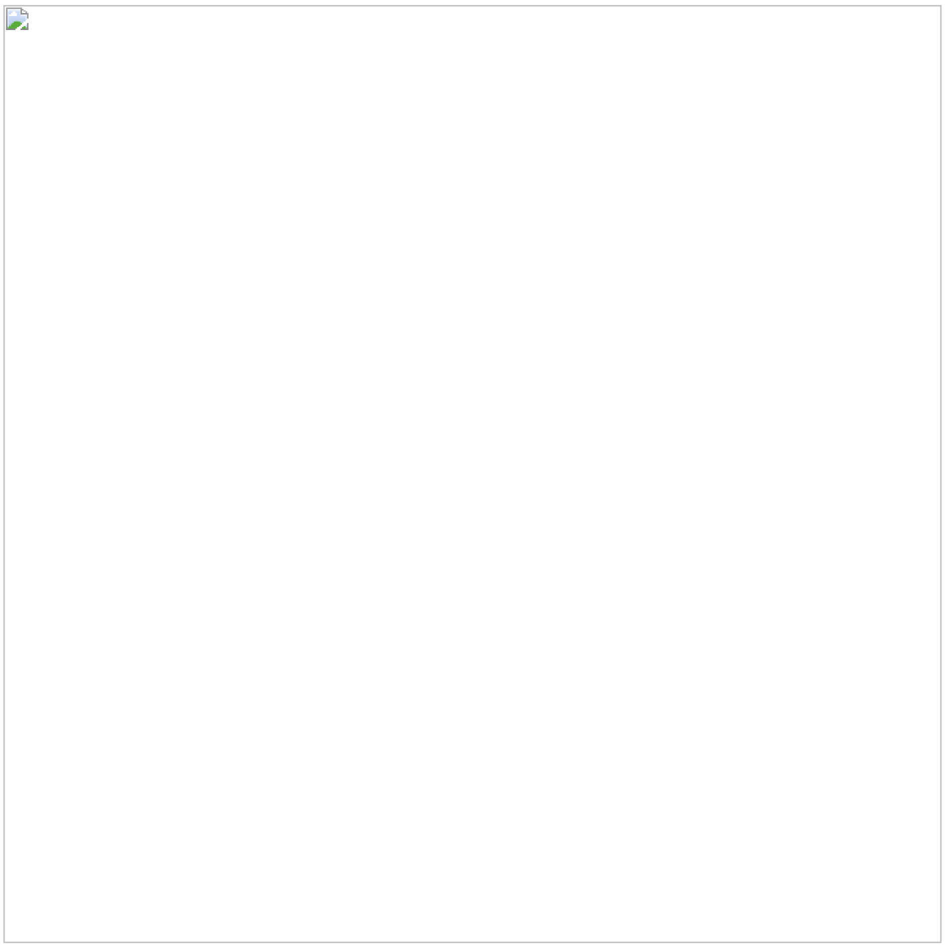
```
# Python3
# Brute-force Solution
class Solution:
    def feasible(self, nums, k, mid):
        cnt = 1
        cur_sum = 0
        for i in range(len(nums)):
            if cur_sum + nums[i] <= mid:
                cur_sum += nums[i]
            else:
                cnt += 1
                cur_sum = nums[i]
        return cnt

    def splitArray(self, nums: List[int], k: int) -> int:
        low = max(nums)
        high = sum(nums)

        for i in range(low, high+1):
            if self.feasible(nums, k, i) <= k:
                return i
        return -1
```

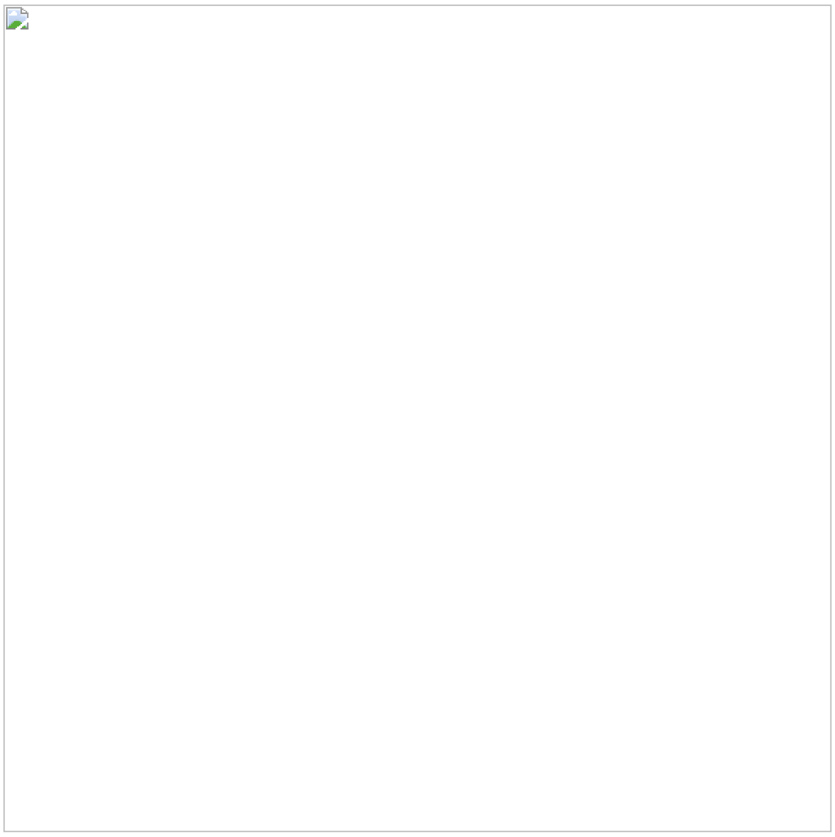
```
# Python3
# Optimal Solution
class Solution:
    def feasible(self, nums, k, mid):
        cnt = 1
        cur_sum = 0
        for i in range(len(nums)):
            if cur_sum + nums[i] <= mid:
                cur_sum += nums[i]
            else:
                cnt += 1
                cur_sum = nums[i]
        return cnt

    def splitArray(self, nums: List[int], k: int) -> int:
        low = max(nums)
        high = sum(nums)

        while low <= high:
            mid = (low + high) >> 1

            if self.feasible(nums, k, mid) > k:
                low = mid + 1
            else:
                high = mid - 1
        return low
```

```
// C++
// Optimal Solution
class Solution {
public:
    int feasible(vector<int>& nums, int k, int mid) {
        int cnt = 1;
        int cur_sum = 0;
        for (auto x: nums) {
            if (cur_sum + x <= mid) {
                cur_sum += x;
            }
            else {
                cnt++;
                cur_sum = x;
            }
        }
        return cnt;
    }
    int splitArray(vector<int>& nums, int k) {
        int low = *max_element(nums.begin(), nums.end());
        int high = accumulate(nums.begin(), nums.end(), 0);

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (feasible(nums, k, mid) > k) {
                low = mid + 1;
            }
            else {
                high = mid - 1;
            }
        }
        return low;
    }
};
```

Median of two sorted arrays

- Approach
    - Brute-force
        - Merge both arrays into one array and return the average of the middle two elements if the resultant array is of even length else return the middle element if the length is odd
        - Time Complexity: $O(n1 + n2)$
        - Space Complexity: $O(n1 + n2)$
    - Better
        - Maintain 2 pointers at the start of both arrays then increment the pointers (increment pointer of first array if `nums1[ptr1] < nums2[ptr2]` else increment pointer of second array) until `((n1 + n2) / 2)` if odd else `((n1 + n2) / 2 + 1)` if even
        - Time Complexity: $O((n1 + n2)/2)$
        - Space Complexity: $O(1)$
    - Optimal
        - We need to divide both arrays into parts such that the addition of the number of elements in the first parts of both elements == `(len(nums1) + len(nums2) / 2)` and `left1 ≤ right2` and `left2 ≤ right1` where left1, left2 are last elements of first parts of both arrays and similarly right1 and right2 are first elements of second parts of both the arrays





- We use binary search to identify these partition indices
- Time Complexity: $O(log(min(n1, n2)))$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        res = []
        i = 0
        j = 0
        while i < len(nums1) and j < len(nums2):
            if nums1[i] < nums2[j]:
                res += nums1[i],
                i += 1
            else:
                res += nums2[j],
                j += 1
        while i < len(nums1):
            res += nums1[i],
            i += 1
        while j < len(nums2):
            res += nums2[j],
            j += 1
        if len(res) & 1:
            return res[len(res)//2]
        return (res[len(res)//2] + res[len(res)//2 - 1]) / 2.0
```

```
# Python3
# Better Solution
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        n1, n2 = len(nums1), len(nums2)
        n = n1 + n2

        end = n // 2
        i, i1, i2  = 0, 0, 0             # Pointer for implict merge list, nums1 pointer, nums2 pointer
        current, previous = 0, 0         # current and previous value in implict merge

        # Implictly build half of the sorted merge list
        # but only save last values
        while i <= end:
            previous = current
            if i1 == n1:                 # First list is exhausted ==> choose from second list
                current = nums2[i2]
                i2 += 1
            elif i2 == n2:               # Second list ist exhaused ==> choose from first list
                current = nums1[i1]
                i1 += 1
            elif nums1[i1] < nums2[i2]:  # Choose element from first list
                current = nums1[i1]
                i1 += 1
            else:                        # Choose element from second list
                current = nums2[i2]
                i2 += 1

            i += 1

        if n % 2 == 0:
            return (previous + current) / 2.0
        else:
            return current
```

```
# Python3
# Optimal Solution
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if len(nums1) > len(nums2):
            return self.findMedianSortedArrays(nums2, nums1)


        m, n = len(nums1), len(nums2)
        left, right = 0, m

        while left <= right:
            partitionA = (left + right) // 2
            partitionB = (m + n + 1) // 2 - partitionA

            maxLeftA = float('-inf') if partitionA == 0 else nums1[partitionA - 1]
            minRightA = float('inf') if partitionA == m else nums1[partitionA]
            maxLeftB = float('-inf') if partitionB == 0 else nums2[partitionB - 1]
            minRightB = float('inf') if partitionB == n else nums2[partitionB]

            if maxLeftA <= minRightB and maxLeftB <= minRightA:
                if (m + n) % 2 == 0:
                    return (max(maxLeftA, maxLeftB) + min(minRightA, minRightB)) / 2
                else:
                    return max(maxLeftA, maxLeftB)
            elif maxLeftA > minRightB:
                right = partitionA - 1
            else:
                left = partitionA + 1
```

```
// C++
// Optimal Solution
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        if (nums2.size() < nums1.size()) return findMedianSortedArrays(nums2, nums1);
        int n1 = nums1.size();
        int n2 = nums2.size();

        int low = 0;
        int high = n1;

        while (low <= high) {
            int cut1 = (low + high) >> 1;
            int cut2 = (n1 + n2 + 1) / 2 - cut1;

            int left1 = cut1 == 0 ? INT_MIN : nums1[cut1 - 1];
            int left2 = cut2 == 0 ? INT_MIN : nums2[cut2 - 1];

            int right1 = cut1 == n1 ? INT_MAX : nums1[cut1];
            int right2 = cut2 == n2 ? INT_MAX : nums2[cut2];

            if (left1 <= right2 && left2 <= right1) {
                if ((n1 + n2) & 1) {
                    return max(left1, left2);
                }
                return (max(left1, left2) + min(right1, right2)) / 2.0;
            }
            else if (left1 > right2) {
                high = cut1 - 1;
            }
            else {
                low = cut1 + 1;
            }
        }
        return 0.0;
    }
};
```

Kth element of two sorted Arrays

- Approach
    - Brute-force
        - Merge both arrays into one array and return kth element
        - Time Complexity: $O(n+m)$
        - Space Complexity: $O(n+m)$
    - Better
        - Maintain two pointers at the start of both arrays and increment the pointers as necessary for k times
        - Time Complexity: $O(k)$
        - Space Complexity: $O(1)$
    - Optimal
        - We can part it in such a way that our kth element will be at the end of the left half array

- With help of binary search, we will divide arrays into partitions with keeping k elements in the left half and l1 ≤ r2 and l2 ≤ r1
- Then using the condition mentioned above, check if the left half is valid
- If valid, print the maximum of both array’s last element
- If not, move the range towards the right if l2 > r1, else move the range towards the left if l1 > r2
- Time Complexity: $O(log(min(n1, n2)))$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def kthElement(self,  arr1, arr2, n, m, k):
        res = []
        i = j = 0
        while i < n and j < m:
            if arr1[i] < arr2[j]:
                res += arr1[i],
                i += 1
            else:
                res += arr2[j],
                j += 1
        while i < n:
            res += arr1[i],
            i += 1
        while j < m:
            res += arr2[j],
            j += 1
        return res[k-1]
```

```
# Python3
# Better Solution
class Solution:
    def kthElement(self,  arr1, arr2, n, m, k):
        i = j = 0
        ans = None
        while k > 0 and i < n and j < m:
            if arr1[i] < arr2[j]:
                ans = arr1[i]
                i += 1
            else:
                ans = arr2[j]
                j += 1
            k -= 1
        while k > 0 and i < n:
            ans = arr1[i]
            k -= 1
            i += 1
        while k > 0 and j < m:
            ans = arr2[j]
            k -= 1
            j += 1
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def kthElement(self, arr1, arr2, n, m, k):
        if n > m:   return self.kthElement(arr2, arr1, m, n, k)

        low = max(0, k-m)
        high = min(k, n)

        while low <= high:
            cut1 = (low + high) >> 1
            cut2 = k - cut1

            left1 = float("-inf") if cut1 == 0 else arr1[cut1 - 1]
            left2 = float("-inf") if cut2 == 0 else arr2[cut2 - 1]
            right1 = float("inf") if cut1 == n else arr1[cut1]
            right2 = float("inf") if cut2 == m else arr2[cut2]

            if left1 <= right2 and left2 <= right1:
                return max(left1, left2)
            elif left1 > right2:
                high = cut1 - 1
            else:
                low = cut1 + 1
        return -1
```

```
// C++
// Optimal Solution
int kthelement(int arr1[], int arr2[], int m, int n, int k) {
    if(m > n) {
        return kthelement(arr2, arr1, n, m, k);
    }

    int low = max(0,k-m), high = min(k,n);

    while(low <= high) {
        int cut1 = (low + high) >> 1;
        int cut2 = k - cut1;
        int l1 = cut1 == 0 ? INT_MIN : arr1[cut1 - 1];
        int l2 = cut2 == 0 ? INT_MIN : arr2[cut2 - 1];
        int r1 = cut1 == n ? INT_MAX : arr1[cut1];
        int r2 = cut2 == m ? INT_MAX : arr2[cut2];

        if(l1 <= r2 && l2 <= r1) {
            return max(l1, l2);
        }
        else if (l1 > r2) {
            high = cut1 - 1;
        }
        else {
            low = cut1 + 1;
        }
    }
    return 1;
}
```

Find a Peak Element II (in the matrix)

A peak element in a 2D grid is an element that is strictly greater than all of its adjacent neighbors to the left, right, top, and bottom. You may assume that the entire matrix is surrounded by an outer perimeter with the value -1 in each cell.

- Approach
  - Brute-force
    - Go through all the elements and check for peak element
    - Time Complexity: $O(n * m)$
    - Space Complexity: $O(1)$
  - Optimal
    - For each row repeat the process
    - There are multiple sorted parts in the row
    - Thus we can apply binary search
    - Check if mid is the peak else search in the direction where the sequence is increasing i.e. search on left side if nums[mid-1] > nums[mid] and nums[mid-1] > nums[mid + 1] else search on right side
    - Time Complexity: $O(nlogm)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def peak(self, mat, n, m, i, mid):
        top = -1 if i < 0 else mat[i - 1][mid]
        bottom = -1 if i == n - 1 else mat[i + 1][mid]
        left = -1 if mid < 0 else mat[i][mid - 1]
        right = -1 if mid == m - 1 else mat[i][mid + 1]
        return (mat[i][mid] > top and mat[i][mid] > bottom and mat[i][mid] > left and mat[i][mid] > right)
    def findPeakGrid(self, mat: List[List[int]]) -> List[int]:
        n = len(mat)
        m = len(mat[0])

        for i in range(n):
            for j in range(m):
                if self.peak(mat, n, m, i, j):
                    return [i, j]
        return []
```

```
# Python3
# Optimal Solution
class Solution:
    def peak(self, mat, n, m, i, mid):
        top = -1 if i < 0 else mat[i - 1][mid]
        bottom = -1 if i == n - 1 else mat[i + 1][mid]
        left = -1 if mid < 0 else mat[i][mid - 1]
        right = -1 if mid == m - 1 else mat[i][mid + 1]
        return (mat[i][mid] > top and mat[i][mid] > bottom and mat[i][mid] > left and mat[i][mid] > right)

    def findPeakGrid(self, mat: List[List[int]]) -> List[int]:
        n = len(mat)
        m = len(mat[0])

        for i in range(n):
            if self.peak(mat, n, m, i, 0):  return [i, 0]
            if self.peak(mat, n, m, i, m - 1):  return [i, m - 1]

            low = 1
            high = m - 2

            while low <= high:
                mid = (low + high) >> 1

                if self.peak(mat, n, m, i, mid):
                    return [i, mid]
                elif mat[i][mid] < mat[i][mid + 1] and mat[i][mid - 1] < mat[i][mid + 1]:
                    low = mid + 1
                else:
                    high = mid - 1
        return []
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool peak(vector<vector<int>>& mat, int n, int m, int row, int col) {
        int top = row == 0 ? -1 : mat[row - 1][col];
        int bottom = row == n - 1 ? -1 : mat[row + 1][col];
        int left = col == 0 ? -1 : mat[row][col - 1];
        int right = col == m - 1 ? -1 : mat[row][col + 1];

        return (mat[row][col] > top && mat[row][col] > bottom && mat[row][col] > left && mat[row][col] > right);
    }

    vector<int> findPeakGrid(vector<vector<int>>& mat) {
        int n = mat.size();
        int m = mat[0].size();

        for (int row = 0; row < n; row++) {
            if (peak(mat, n, m, row, 0)) {return {row, 0};}
            if (peak(mat, n, m, row, m-1)) {return {row, m-1};}

            int low = 1;
            int high = m - 2;

            while (low <= high) {
                int mid = low + (high - low) / 2;

                if (peak(mat, n, m, row, mid)) {return {row, mid};}
                else if (mat[row][mid] < mat[row][mid + 1] && mat[row][mid - 1] < mat[row][mid + 1]) {low = mid + 1;}
                else {high = mid - 1;}
            }
        }
        return {};
    }
};
```

Median in a row-wise sorted matrix

- Approach
    - Brute-force
        - Copy all N*M elements in to 1D array and sort that array
        - Return mid element if N*M is odd else average of both middle elements
        - Time Complexity: $O((n*m)log(n*m))$
        - Space Complexity: $O(n*m)$
    - Optimal
        - By just traversing the first column, we find the minimum element and by just traversing the last column, we find the maximum element
        - We can apply binary search between min and max element or 1 and 10^9
        - Median means there must be equal element in both halves
        - Thus we find the number of elements that are less than mid element in the matrix using binary search on each row
        - Check if the number of elements less than mid == R*C // 2

- Time Complexity: $O(nlogm)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def median(self, matrix, R, C):
        res = []
        for i in matrix:
            for j in i:
                res += j,
        res.sort()
        return res[R*C//2]
```

```
# Python3
# Optimal Solution
class Solution:
    def cnt(self, matrix, mid):
        ans = 0
        for row in matrix:
            low = 0
            high = len(row) - 1

            while low <= high:
                m = (low + high) >> 1
                if row[m] <= mid:
                    low = m + 1
                else:
                    high = m - 1
            ans += low
        return ans

    def median(self, matrix, R, C):
        low = 1
        high = 1000000000

        while low <= high:
            mid = (low + high) >> 1
            if self.cnt(matrix, mid) <= (R*C)/2:
                low = mid + 1
            else:
                high = mid - 1
        return low
```

```
// C++
// Optimal Solution
int countSmallerThanMid(vector<int> &row, int mid)
{
  int l = 0, h = row.size() - 1;
  while (l <= h)
  {
    int md = (l + h) >> 1;
    if (row[md] <= mid)
    {
      l = md + 1;
    }
    else
    {
      h = md - 1;
    }
  }
  return l;
}
int findMedian(vector<vector<int>> &A)
{
  int low = 1;
  int high = 1e9;
  int n = A.size();
  int m = A[0].size();
  while (low <= high)
  {
    int mid = (low + high) >> 1;
    int cnt = 0;
    for (int i = 0; i < n; i++)
    {
      cnt += countSmallerThanMid(A[i], mid);
    }
    if (cnt <= (n * m) / 2)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return low;
}
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

- Optimal
  -
  - Time Complexity: $O(n^3)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```