BINARY SEARCH

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Search in a 2D Matrix**			
	Given an m x n integer matrix matrix with the following two properties: - Each row is sorted in non-decreasing order. -The first integer of each row is greater than the last integer of the previous row. Given an integer target, return true if target is in matrix or false otherwise.	- Approach_1: Row-wise binary search	O(M * logN)	O(1)

		- Approach_2: Use Banary Search to find row and then again to find the column
		1) Binary search on rows
		`if(target >= matrix[m_row][0] && target <= matrix[m_row][n-1]) break;`
		`else if(target >= matrix[m_row][n-1]) l = m_row+1;`
		`else r = m_row-1;`
		2) binary search on columns: use 'm_row'
			O(logM+ logN)	O(1)

		- Approach_3 : Binary Search on Flatterned Matrix
		- Initialize **l = 0** and **r = (m*n) - 1** , representing the indices of the first and last elements in the flattened matrix.
		`if(matrix[mid/n][mid%n] == target) return true`
		`if(target < matrix[mid/n][mid%n]) r = mid - 1`
		`else l = mid + 1`
			O(logM*N)	O(1)

2	**Koko Eating Bananas**			

	There are n piles of bananas, the ith pile has piles[i] bananas. The guards have gone and will come back in h hours. Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour. Return the minimum integer k such that she can eat all the bananas within h hours.	- Binary Search: **l = 1 and r = max(piles)** ---> as min value for 'k' can be 1 and max value for 'k' can be max(piles)
		- Use Binary search to calculate 'k'
		`while(l<=r)`
		`m = l + (r-l)/2;`
		`// Count 'total_hours' to eat all the bananas if 'min bananas per hour = m'`
		`for(auto x: piles) total_hours += ceil((double)x/m);`
		`// if we do 'if(total_hours==h) return k', then it's not always possible to get total_hours == h.`
		`//  Example: if piles=[10] and h=9. Then O/P should be 2. For k==2, the value of total_hours=5 which is not equal to h.`
		`if(total_hours<=h){`
		`k = min(k, m);`
		`r = m-1;`
		`}else l = m+1;`
			O(N*logM)
			M = max(piles)
				O(1)

3	**Find Minimum in Rotated Sorted Array**			

	Given the sorted rotated array nums of unique elements, return the minimum element of this array.
	Example: The array nums = [0,1,2,4,5,6,7] might become:
	- [4,5,6,7,0,1,2] if it was rotated 4 times.
	- [0,1,2,4,5,6,7] if it was rotated 7 times.
		- Idea: Rotated sorted array can be divided into two sorted halves.
		- By comparing the middle element with the rightmost element, Determine which half is sorted.
		`ans = INT_MAX`
		`while(l<=r)`
		`m = l + (r-l)/2;`
		`ans = min(ans, nums[m]);`
		`if(nums[m]<nums[r]) r = m-1;  // Suggests that right half is sorted`
		`else l = m+1;`
			O(logN)	O(1)

4	**Search in Rotated Sorted Array**			
	Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.	1) `if (nums[m] == target) return m` 2) `if (nums[l]<= nums[m])` --> array is left sorted - Check if the target lies in the left region: If yes then `r = m-1` else `l = m+1` 3) `if (nums[m] <= nums[r])` --> array is right sorted - Check if the target lies in the right region: If yes then `l = m+1` else `r = m-1`	O(logN)	O(1)
5	**Time Based Key-Value Store**			

	Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp. - void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp. - String get(String key, int timestamp) Returns a value such that timestamp_prev <= timestamp. If there are multiple such values, it returns the value associated with the largest timestamp_prev. If there are no values, it returns "".	- Hashing + Binary Search - `unordered_map<string, vector<pair<int, string>>> mp;` // {key --> {timestamp, value}} - `string get()` - Since, timestamps are naturally in order use binary search	O(logN)	O(N)
6	**Median Two Sorted Arrays**			

	Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.	- **Partition** both the arrays (p1 & p2) in such a way that elements on the left side of the partitions are <= the elements on the right side
		- The search iterates until a valid partition is found, allowing the calculation of the median.
		`if (max_l1 <= min_r2 && max_l2 <= min_r1) {` // Valid partition
		`if (n % 2 == 0) ans = (max(max_l1, max_l2) + min(min_r1, min_r2)) / 2.0;`
		`else ans = min(min_r1, min_r2);`
		`break;`
		`else if (max_l1 > min_r2) r = p1 - 1;`
		`else l = p1 + 1;`
			O(log(min(m,n))	O(1)