HEAPS & PRIORITY QUEUE

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Kth Largest Element In a Stream**			
	Given an integer k, return the kth largets element in the stream	- Priority Queue - Maintain a min heap of **size = k** - Top element will always be the kth largest element	O(N*logN)	O(N)
2	**Last Stone Weight**			
	Given array of stones to smash, return smallest possible weight of last stone. If x == y both stones destroyed, if x != y stone x destroyed, stone y = y - x Ex: stones = [2,7,4,1,8,1] -> 1, [2,4,1,1,1], [2,1,1,1], [1,1,1], [1]	- Priority Queue - Maintain a max heap - Pop first 2 elements everytime giving two stones with the heaviest weights	O(N*logN)	O(N)
3	**K Closest Points to Origin**			
	Given array of points & an int k, return k closest points to (0, 0) Ex. points = [[1,3],[-2,2]], k = 1 -> [[-2,2]]	- Priority Queue of pairs - Maintain a min heap	O(N*logN)	O(N)
4	**Kth Largest Element In An Array**			
	Given an integer k, return the kth largets element in the stream	- Priority Queue - Maintain a min heap of size = k --> stores 'k' largest elements at all times	O(N*logK)	O(K)
5	**Task Scheduler**			

	Given a characters array tasks, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle. However, there is a non-negative integer **'n'** that represents the cooldown period between two same tasks (the same letter in the array). Return the least number of units of times that the CPU will take to finish all the given tasks.	- Unordered_map
		1. Count the frequency of each task and store it in unordered_map
		  - max_freq: To store the highest frequency
		2. (max_freq-1) * (n+1) = Total no. of CPU cycles to schedule 'max_freq-1' occurences of the most frequent task
		  `for(auto p: mp)`
		    `// 1. We will need at least 1 CPU cycle for the last occurrence of most frequent task.`
		    `// 2. If there are multiple tasks with highest frequency, they will all need 1 more cycle.`
		    `if(p.second==max_freq) ans++;`
		  `ans = max(ans, tasks.size())`
			O(N)	O(N)

6	**Design Twitter**			

	Design a simplified version of Twitter.
	- void postTweet(int userId, int tweetId) Composes a new tweet with ID tweetId by the user userId. Each call to this function will be made with a unique tweetId.
	- vector<integer> getNewsFeed(int userId) Retrieves the 10 most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user themself. Tweets must be ordered from most recent to least recent.
	- void follow(int followerId, int followeeId) The user with ID followerId started following the user with ID followeeId.
	- void unfollow(int followerId, int followeeId) The user with ID followerId started unfollowing the user with ID followeeId.
		1. `vector<pair<int, int>> tweets` **--> Stores tweets as pairs of {userId, tweetId}**
		2. `unordered_map<int, unordered_set<int>> friends` **--> {UserId, list of all the users followed by this userId}**
		- **`postTweet`**`(int userId, int tweetId)`
		    `tweets.push_back({userId, tweetId});`
		- **`getNewsFeed`**`()`: Iterates through the posts vector in reverse order (from the most recent to the oldest).
		    `for(int i=posts.size()-1; i>=0 && ans.size()<10; i--)`
		      `// tweets[i].first == userId : To include its own tweets`
		      `if(tweets[i].first == userId || friends[userId].find(tweets[i].first)!=friends[userId].end())`
		        `ans.push_back(posts[i].second);`
		- **`follow`**`(int followerId, int followeeId)`
		    `friends[followerId].insert(followeeId);`
		- **`unfollow`**`(int followerId, int followeeId)`
		    `friends[followerId].erase(followeeId);`
			O(N)	O(N)

7	**Find Median From The Data Stream**			

HEAPS & PRIORITY QUEUE

No.	Problem Statement	Solution	Time complexity	Space complexity

	Implement data structures that gets the median from a data stream
	- void addNum(int num) adds the integer num from the data stream to
	  the data structure
	- double findMedian() returns the median of all elements so far.
		- Maintain **two** priority queue such that the difference in their sizes is either '**0**' or '**1**'
		- priority_queue<int> lower // Max value is at top i.e (3,2,1) --> maintains list of values **less** than lower.top()
		- priority_queue<int, vector<int>, greater<int>> higher // Min value is at top i.e (4,5) --> maintains list of values **more** than higher.top()
		- Median : `if size of max heap == min heap : median = max_heap.top() + min_heap.top() / 2`
		`if size of max heap > min heap : median = max_heap.top()`
		`if size of min heap > max heap : median = min_heap.top()`
			addNum: O(logN)
			findMedian: O(1)
				O(N)