GREEDY

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Maximum Subarray**			
	Given an integer array 'nums', find the subarray with the largest sum, and return its sum.	- Approach_1: Dynamic Programming - **dp[i] = max_sum including element at nums[i]** --> Doesn't necessarily mean that the subarray begins from '0' - Recurrence Relation: **dp[i] = max(dp[i-1] + nums[i] , nums[i])** - `max_sum = max(max_sum, dp[i])`	O(N)	O(N)
		- Approach_2: Greedy - At every position determine if it's better to add current number or start over - `max_ending_here = max(nums[i], max_ending_here + nums[i])` - `max_so_far = max(max_so_far, max_ending_here)`	O(N)	O(1)
2	**Jump Game**			

	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.
		- Approach_1: Dynamic Programming
		- **dp[i] = true --> If it's possible to reach position 'i' from 'start'**
		`i: 1 -> n-1`
		`j: i-1 -> 0`
		`if(dp[j] && (j+nums[j])>=i)` // 'j+nums[j]' gives farthest step reachable from 'j'
		`dp[i] = true;`
			O(N^2)	O(N)

		- Approach_2: Greedy
		- **At each point, calculate furthest reachable point**
		`i: 0->n-1`
		`if(i>reachable) return false` // It suggests that we have reached a position 'i' that cannot be reached in any way
		`reachable = max(reachable, i+nums[i])`
		`if(reachbel>=n-1) return true`
			O(N)	O(1)

3	**Jump Game II**			

	You are initially positioned at nums[0], and each element in the array represents your maximum jump length at that position.
	Return minimum number of jumps to reach nums[n-1].
		- current_max_reach : maximum index that can be reached currently
		- next_max_reach: maximum index that can be reached next
		`for(int i=0; i<n; i++)`
		`if(curr_max_reach >= n-1) break;`
		`next_max_reach = max(next_max_reach, i+nums[i]);`
		`if(i==curr_max_reach){`
		`jumps++;`
		`curr_max_reach = next_max_reach;`
			O(N)	O(1)

4	**Gas Station**			

	Give '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. 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.	1) Calculate total_gas and total_cost.
		2) If total_cost > total_gas --> return -1
		3) Find the starting gas station by iterating through the gas stations and **checking whether you can start without running out of gas**
		`i : 0 --> n-1`
		`current_gas += gas[i] - cost[i];`
		`if(current_gas<0){`
		`current_gas = 0;` --> If running out of gas, reset starting station to the next one
		`ans = i+1;` --> 'ans' can be the next index 'i+1'
		`}`
			O(N)	O(1)

5	**Hand of Straights**			

GREEDY

No.	Problem Statement	Solution	Time complexity	Space complexity
	Given an integer array hand where hand[i] is the value written on the ith card and an integer 'groupSize', return true if you can rearrange the cards into groups so that each group is of size 'groupSize', and consists of 'groupSize' consecutive cards.	- Idea: Use '**map**' instead of '**unordered_map**' `while(!mp.empty())` `int x = mp.begin()->first;` `for(int i=0; i<groupSize; i++)` `if(mp.find(x)!= mp.end())` `mp[x]--;` `if(mp[x]==0) mp.erase(x);` `x++;` `else return false;`	O(N*logN)	O(N)
6	**Merge Triplets To Form Target Triplets**			
	Given a 2D integer array 'triplets' and an integer array target = [x, y, z], Return true if it is possible to obtain the 'target' by applying following operation. - 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].	- Idea: Use 'unordered_set' to keep track of unique indices. - In the end 'unordered_set' should have {0, 1, 2} indicating that we can form 'target' from 'vector of triplets' `for(auto v: triplets){` `if(v[0]>target[0] || v[1]>target[1] || v[2]>target[2]) continue;` --> Skip the current triplet `if(v[0]==target[0]) s.insert(0);` `if(v[1]==target[1]) s.insert(1);` `if(v[2]==target[2]) s.insert(2);`	O(N)	O(1)
7	**Partition Labels**			
	Partition the string 's' 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.	- Idea: Use 'last_index(26)' vector to store the last occurrence index of each character `i: 0 --> n-1` `end = max(last_index[s[i]-'a'], end)` --> Update the end index to the maximum of the current character's last occurrence `cnt++` --> Increment the count of characters in the current partition	O(N)	O(1)
8	**Valid Paranthesis 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 "".	- Idea: Check all the possibilities for the string to be **not** balanced if none found then return true 1) Consider * as '(' `for(`**`i=0`**`; i<n; i++)` `if(s[i]=='(' || s[i]=='*') balanced++;` `else balanced--;` `if(balanced<0) return false;` --> Suggests there are more **')'** than **'('** and **'*'** 2) Consider * as ')' `for(`**`i=n-1`**`; i>=0; i--){` `if(s[i]==')' || s[i]=='*') balanced++;` `else balanced--;` `if(balanced<0) return false;`	O(N)	O(1)