DYNAMIC PROGRAMMING

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Climbing Stairs**			
	You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?	- We can reach `ith` step in 2 ways: - 1 step from i-1 - 2 steps from i-2 - **dp[i] = number of ways to reach step i from '0'** - Initialize dp[0] = 1, dp[1] = 1 - Recurrence relation: **dp[i] = dp[i - 1] + dp[i - 2]**	O(N)	O(N)
2	**Min Cost Climbing Stairs**			
	Given an integer array cost where cost[i] is the cost of ith step on a staircase. You can either climb one or two steps. You can either start from the step with index 0, or the step with index 1. Return the minimum cost to reach the top of the floor.	- We can reach ith step in 2 ways: - 1 step from i-1 - 2 steps from i-2 - **dp[i] = minimum cost to reach step i** - Initialize `dp[0] = cost[0],   (dp[1] = cost[1]   --> Because we can start from step 1)` - Recurrence relation: **dp[i] = min(cost[i] + dp[i - 1], cost[i] + dp[i - 2])**	O(N)	O(N)
3	**House Robber**			
	Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police. (The police will be contected automatically if the adjacent houses were broken into on the same night)	- We have 2 options at ith index - Rob ith house and move to i+2 - Skip ith house and move to i+1 - **dp[i] = max amount that can be robbed till index i** - Initialize `dp[0] = nums[0],    dp[1] = max(nums[0], nums[1])` - Recurrence relation: **dp[i] = max(nums[i] + dp[i-2], dp[i-1])**	O(N)	O(N)
4	**House Robber II**			
	Given an integer array nums representing the amount of money of each house (all houses are arranged in circle),return the maximum amount of money you can rob tonight without alerting the police. (The police will be contected automatically if the adjacent houses were broken into on the same night)	- We can - 1) Either rob house at index **0** - 2) OR house at index **n-1** (can't rob both at the same time) - **ans = max(helper(0, n-2, nums), helper(1, n-1, nums))**	O(N)	O(1)
5	**Longest Pelindromic Substring**			
	Given a string **s**, return the longest palindromic substring in **s**.	- Approach_1: Dynamic Programming - Boolean 2d array dp: **dp[i][j] = true if s[i....j] is a pelindrome** - Recurrence Relation: **dp[i][j] = true --> if(dp[i+1][j-1] && s[i]==s[j])**	O(N^2)	O(N^2)

		- Approach_2: Two Pointers
		- Pelindromic strings mirrors around cente of teh string
		- Pelindromic substrings having **odd** length is centered around **(i, i)**
		- Pelindromic substrings having **even** length is centered around **(i, i+1)**
		- Use a helper function to expand the string from center.
		`int len;`
		`while(l>=0 && r<s.size()){`
		`    if(s[l] == s[r])`
		`        len = r-l+1, l--, r++;`
			O(N^2)	O(1)

6	**Pelindromic Substring**			
		- Approach_1: Dynamic Programming - Boolean 2d array dp: **dp[i][j] = true if s[i....j] is a pelindrome** - Recurrence Relation: **dp[i][j] = true --> if(dp[i+1][j-1] && s[i]==s[j])**	O(N^2)	O(N^2)

DYNAMIC PROGRAMMING

No.	Problem Statement	Solution	Time complexity	Space complexity

	Given a string s, return the number of palindromic substrings in it.	- Approach_2: Two Pointers
		- Pelindromic strings mirrors around cente of teh string
		  - Pelindromic substrings having **odd** length is centered around **(i, i)**
		  - Pelindromic substrings having **even** length is centered around **(i, i+1)**
		- Use a helper function to expand the string from center.
		`int cnt = 0;`
		`while(l>=0 && r<s.size()){`
		  `if(s[l] == s[r])`
		    `cnt++``, l--, r++;`
			O(N^2)	O(1)

7	Decode Ways			
	Given a string with only digits, return # ways to decode it (letter -> digit) Ex: s = "12" -> 2 (1 2 or 12), Ex: s = "226" -> 3 (2 26 or 22 6 or 2 2 6)	- **dp[i] = Unique ways to decode s[0 ... i-1]** - `dp[0] = 1` --> Only one way to decode an empty string. - For each index i: - `If (s[i] != '0')` then `dp[i] = dp[i-1]` because the number of ways to decode up to index i would be equal to the number of ways to decode up to index i-1 - `Else dp[i] = 0` - `If(s[i-1]=='1' || (s[i-1]=='2' && s[i]<='6')` then `dp[i]` **+=** `dp[i-2]`	O(N)	O(N)
8	Coin Change			
	Given array of coins & amount, return fewest coins to make that amount Ex: coins = [1,2,5], amount = 11 --> 3, $11 = $5 + $5 + $1	- 2D DP : dp[n][amount+1] - dp[i][amt] = min # coins from coins[0 -> i] to generate amount 'amt' - Recurrence Relation: 1) Don't consider coins[i] --> **dp[i][amt] = dp[i-1][amt]** 2) Consider coins[i] --> **dp[i][amt] = min(dp[i][amt], 1 + dp[i][amt-coins[i]])**	O(N*Amount)	O(N*Amount)
9	Maximum Product Subarray			
	Given an integer array nums, find a subarray that has the largest product, and return the product.	- For every index **'i'** keep track of '**max_product** ' and '**min_product** ' - max_product : Represents the maximum product achievable by incorporating the element at index i Note: this doesn't imply the subarray begins at index 0; it can commence at any position prior to or at index i **- max_product = max(nums[i], max_product*nums[i])** **- min_product = min(nums[i], min_product*nums[i])**	O(N)	O(1)
10	Word Break			
	Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words. Worst case: s = "aaab" wordDict = ["a", "aa", "aaa"]	Approach_1: Dynamic Programming - **dp[i] = true if s(0 ... i-1) is a word in the dictionary** 1) i: 1->n 2) j: i-1 -> 0 3) if(dp[j]==true) then search if substring(j, i-j) is in word_dict or not	O(N^2 * K) K = Avg. length of words in wordDict	O(N)

		Approach_2: **Trie** Data structure
		- Use Trie to store words in 'wordDict'
		- Use 'Trie search' in step 3 above
			O(N^2 * K)
			K = Avg. length of words in wordDict
				O(N + M*K)
				M = wordDict.size()

11	Longest Increasing Subsequence			
		Approach_1: Dynamic Programming - **dp[i] : length of LIS ending at index i** - `i: 1->n` - `j: 0->i` - `if nums[j] < nums[i]` - Recurrence Relation: **dp[i] = max(dp[i], dp[j] + 1)**	O(N^2)	O(N)

DYNAMIC PROGRAMMING

No.	Problem Statement	Solution	Time complexity	Space complexity
	Given an integer array 'nums', return the length of the longest strictly increasing subsequence.	Approach_2: Greedy - **Maintain set of active lists of varying lengths** - Maintain 'tail' array that stores last elements of all the active lists - Traverse 'nums' array from i: 1->n - Case 1: if nums[i] is smallest: Start a new active list of length 1 - Case 2: if nums[i] is largest: Find the list with maximum size, clone that list and append nums[i] - Case 3: if nums[i] is in between: Find(Binary Search) the list where the last element is largest element smaller than nums[i] Clone that list and append nums[i], discard all the list having same length as the new list	O(N*logN)	O(N)
12	**Partition Equal subset Sum**			
	Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.	- target = sum/2 - dp[n][target] - **dp[i][j] = true if it's possible to achieve the sum of 'j' fom nums[0 ... i]** - For each index i: 1) Either consider the element at index i 2) Or dont consider the element at index i Recurrence Relation: **dp[i][j] = dp[i-1][j-nums[i] || dp[i-1][j]**	O(N*target)	O(N*target)
13	**Unique Paths**			
	Given grid, return # of unique paths from top-left to bottom-right Ex: m = 3, n = 2 -> 3 unique paths (R->D->D, D->D->R, D->R->D)	- We can reach (i, j) in 2 ways: - 1 step from (i-1, j) : up - 1 step from (i, j-1) : left - Recurrence relation: **dp[i][j] = dp[i - 1][j] + dp[i][j-1]**	O(M*N)	O(M*N)
14	**Longest Common Subsequence**			
	Given two strings text1 and text2, return the length of their longest common subsequence.	- At each (i, j) we have 2 cases: - 1) s1[i] == s2[j] - Recurrence relation: **dp[i][j] = 1+ dp[i-1][j-1]** - 2) s1[i] != s2[j] - Recurrence relation: **dp[i][j] = max(dp[i - 1][j] , dp[i][j-1])**	O(M*N)	O(min(M, N))
15	**Best Time to Buy and Sell Stock with Cooldown**			
	You are given an array prices where prices[i] is the price of a given stock on the ith day. After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day). Maximize the profit	- Create 3 dp arrays : buy, sell, rest - buy[i] , sell[i], rest[i] = profit at time i - Initialize, buy[0] = - prices[0] - Recurrence relation for i: 1-->n 1) **buy[i] = max(buy[i-1], rest[i-1] - prices[i])** 2) **sell[i] = buy[i-1] + prices[i]** 3) **rest[i] = max(rest[i-1], sell[i-1])**	O(N)	O(N)
16	**Coin change II**			
	Given array of coins & amount, Return the number of combinations that make up that amount. Ex: amount = 5, coins = [1,2,5] --> 4	- 2D DP : dp[n][amount+1] - **dp[i][amt] = # of combinations from coins[0 -> i] to generate amount 'amt'** - Recurrence Relation: 1) Don't consider coins[i] --> **dp[i][amt] = dp[i-1][amt]** 2) Consider coins[i] --> **dp[i][amt] += dp[i][amt-coins[i]]**	O(N*Amount)	O(N*Amount)
17	**Target Sum**			

DYNAMIC PROGRAMMING

No.	Problem Statement	Solution	Time complexity	Space complexity

	Given int array & a target, want to build expressions with '+' & '-'
	Return number of different expressions that evaluates to target.
	Example: nums = [1,1,1,1,1], target = 3 --> 5
	-1 + 1 + 1 + 1 + 1 = 3
	+1 - 1 + 1 + 1 + 1 = 3
	+1 + 1 - 1 + 1 + 1 = 3
	+1 + 1 + 1 - 1 + 1 = 3
	+1 + 1 + 1 + 1 - 1 = 3
		- 2D DP: vector of unordered map vector<unordered_map<int, int>> dp(n+1)
		- **dp[i] = unordered_map of {target, # ways to evaluate to target} from 'nums[0 … i-1]'**
		- Initialize, dp[0][0] = 1
		- `for(int i=1; i<=n; i++){`
		`    for(auto mp: dp[i-1]){`
		`        dp[i][mp.first + nums[i-1]] += mp.second;`
		`        dp[i][mp.first - nums[i-1]] += mp.second;`
			O(N*target)	O(N*target)

18	**Interleaving String**			
	Given 3 strings, find if s3 is formed by interleaving of s1 & s2 Example: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac" -> true	- 2D DP: dp[m+1][n+1] - dp[i][j]=1 : s1(0 … i-1) and s2(0 … j-1) is processed and make part of s3 dp[i][j]=0 : s1(0 … i-1) and s2(0 … j-1) is processed and don't make part of s3 - Recurrence Relation: **dp[i][j] = (dp[i-1][j] && s1[i-1] == s3[i+j-1]) || (dp[i][j-1] && s2[j-1] == s3[i+j-1])**	O(M*N)	O(M*N)
19	**Edit Distance**			

	Given 2 strings, return min number of operations to convert word1 to word2
	Exp: word1 = "horse", word2 = "ros". --> 3
	horse -> rorse (replace 'h' with 'r')
	rorse -> rose (remove 'r')
	rose -> ros (remove 'e')
		- 2D DP: dp[m+1][n+1]
		- dp[i][j] = min no of operations required to convert word1[0…i] to word2[0…j]
		`for(int i=1; i<=n1; i++){`
		`    for(int j=1; j<=n2; j++)`
		`        if(word1[i-1]==word2[j-1]) dp[i][j] = dp[i-1][j-1];`
		`        else`
		`           insert = 1 + dp[i][j-1];`
		`           remove = 1 + dp[i-1][j];`
		`           replace = 1 + dp[i-1][j-1];`
		`           dp[i][j] = min(insert, min(remove, replace));`
			O(M*N)	O(M*N)

20	**Longest Increasing Path In A Matrix**			
	Given matrix, return length of longest increasing path Example: matrix = [[9,9,4],[6,6,8],[2,1,1]] -> 4, [1,2,6,9]			
21	**Distinct Subsequences**			
	Given 2 strings s & t, Return # of distinct subsequences of s which equals t Example: s = "rabbbit", t = "rabbit" -> 3, RABBbIT, RAbBBIT, RABbBIT			
22	**Burst Balloons**			
	Given array of balloons w/ coins, if burst ith, get (i-1) + i + (i+1) coins Return max coins can collect by bursting the balloons wisely			
23	**Regular Expression Matching**			
	Given string & pattern, implement RegEx matching '.' -> matches any single character '*' -> matches zero or more of the preceding element Matching should cover the entire input string (not partial) Example: s = "aa", p = "a" -> false, "a" doesn't match entire string "aa"			