MATH & GEOMATRY

No	Problem Statement	Solution	Time complexity	Space complexity
1	**Happy Number**			

	A happy number is a number defined by the following process:
	- Starting with any positive integer, replace the number by the sum of the squares of its digits.
	- Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
	- Those numbers for which this process ends in 1 are happy.
	Return true if n is a happy number, and false if not.
	Ex. n = 19 --> true
	Explanation:
	12 + 92 = 82
	82 + 22 = 68
	62 + 82 = 100
	12 + 02 + 02 = 1
		- Idea: **Slow & Fast Pointers**
		- Initialize slow = n. , fast = get_sum(n)
		- Detect cycle using slow & fast pointers
		- If cycle detected then not a hapy number
		`while(slow!=fast && fast!=1)`
		`slow = get_sum(slow);`
		`fast = get_sum(get_sum(fast));`
		`if(fast==1) return true;`
			O(logN)	O(1)

2	**Plus One**			
		Appraoch_1: Traverse the array from the back, keep a variable to track carry	O(NlogN)	O(N)

	You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. Increment the large integer by one and return the resulting array of digits.
	Ex. digits = [1,2,3] -> [1,2,4]
		Approach_2: Space Optimized
		1) If digits[n-1]<9
		`digits[n-1] += 1;`
		`return digits;`
		2) As long as digits[i]==9, keep the carry=1, set digits[i]=0
		`while(i>=0 && digits[i]==9){`
		`digits[i] = 0;`
		`i--;`
		3) As soon as we encounter digits[i]<9, do digits[i]+= carry
		`if(i>=0) digits[i] += carry;`
		`else // Suggests 'digits' has all 9s`
		`digits[0]=1;`
		`digits.push_back(0);`
			O(N)	O(1)

3	**Rotate Image**			

	Given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).	1) Transpose the matrix
		`for i: 0->n`
		`for j: i+1->n`
		`swap(matrix[i][j], matrix[j][i])`
		2) Reverse each row
			O(N^2)	O(1)

4	**Spiral Matrix**			

	Given an m x n matrix, return all elements of the matrix in spiral order.	1) Set up boundaries: left=0, right=n-1, top=0, bottom=m-1
		2) Traverse in spiral form keeping in mind the boundaries
		`while(left<=right && up <=down)`
		`left -> right`
		`up -> down`
		`right -> left --> Again check if(up<=down)`
		`down -> up --> Again check if(left<=right)`
			O(M*N)	O(1)

5	**Set Matrix Zero**			

	Given an m x n integer matrix, if an element is 0, set its entire row and column to 0's.	Approach_1
		- Use **vector<int> row** to keep track of rows that needs to be set to 0
		- Use **vector<int> col** to keep track of columns that needs to be set to 0
		- Traverse the matrix and set the rows and columns to 0
		`for(int i=0; i<m; i++)`
		`for(int j=0; j<n; j++)`
		`if(row[i] || col[j]) matrix[i][j] = 0;`
			O(M*N)	O(M+N)

MATH & GEOMATRY

No	Problem Statement	Solution	Time complexity	Space complexity
		Approach_2 - Use **first row and first column as flag** to determine if entire row and col needs to be set to 0 - if matrix[0][i] = 0 then **ith column** should be set to all zeros - if matrix[i][0] = 0 then **ith row** should be set to all zeros	O(M*N)	O(1)
6	**Power(x,n)**			

	Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).	Approach_1: '**Divide & Conquer** '
		`double helper(double x, int n)`
		`if(n==0) return 1;`
		`if(n%2==0) return helper(x, n/2) * helper(x, n/2);`
		`else return helper(x, n/2) * helper(x, n/2) * x;`
			O(N)	O(logN)

		Aprroach_2: '**Optimized Divide & Conquer** '
		`double helper(double x, int n)`
		`if(n==0) return 1;`
		`double temp = helper(x, n/2);`
		`if(n%2==0) return temp * temp;`
		`else return temp * temp * x;`
			O(logN)	O(logN)

		Appraoch_3: '**Iterative Divide & Conquer** '
		`double ans = 1.0;`
		`for(int i=abs(n); i>0; i=i/2)`
		`if(i%2==1) ans *= x;`
		`x *= x;`
			O(logN)	O(1)

7	**Multiply Strings**			

	Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string. *Note: You must not use any built-in BigInteger library or convert the inputs to integer directly.*	- Idea: **Follow the rules of normal multiplication**
		`string ans(n1+n2, '0')`
		`pos = (n1+n2-1) - (n2-1);` // 'm-1' = no. of zeros that needs to be added before multiplication
		// '(n+m-1) - (m-1)' = initial position for multiplication in 'ans'
		`for(int i=0; i<n2; i++)`
		`carry = 0;`
		`k = pos;`
		`for(int j=n1-1; j>=0; j--)`
		`temp = (num1[j] - '0') * (num2[i] - '0') + carry + (ans[k] - '0');`
		`ans[k] = temp%10 + '0';`
		`carry = temp/10;`
		`k--;`
		`while(carry>0){`
		`temp = carry + ans[k] - '0';`
		`ans[k] = temp%10 + '0';`
		`carry = temp/10;`
		`k--;`
		`pos++;`
			O(N1*N2)	O(1)

8	**Detect Squares**			

	You are given a stream of points on the X-Y plane. Design an algorithm that:
	- Adds new points from the stream into a data structure. Duplicate points are allowed and should be treated as different points.
	- Given a query point, counts the number of ways to choose three points from the data structure such that the three points and the query point form an axis-aligned square with positive area. (An axis-aligned square is a square whose edges are all the same length and are either parallel or perpendicular to the x-axis and y-axis.)
		- Idea: If (p,q) forms a diagonal with any of the points, then we found a square
		- unordered_map<int, unordered_map<int, int>> mp --> [x, [y, cnt]]
		`for(auto v1: mp)`
		`x=v1.first;`
		`for(auto v2:v1.second)`
		`y=v2.first;`
		// Skip if (p,q) is in the same x or y axis as (x,y)
		`if(p==x || q==y) continue;`
		// Counting the frequency of other 3 points in the square besides (p,q)
		// Check if the distance between (p,q) and (x,y) is equal in both dimensions (forms a diagonal)
		`if(abs(x-p)==abs(y-q)) ans+= mp[x][y] * mp[x][q] * mp[p][y];`
			O(K)
			K: Total no. of points added
				O(K)