**Basic Maths Concept for Coding**

**I) Digit concept**

|  |  |
| --- | --- |
| Extract Digits from A Given Number - Tutorial | **Pseducode:-**  **while(N > 0){**  **// Extract the last digit of N**  **int lastDigit = N % 10;**  **SOP(lastDigit);**  **// Remove the last digit from N**  **N = N / 10;**  **}**  **Extracts the digits in reverse order** |

**1. Count Digits:-**

|  |  |
| --- | --- |
| **Problem statement** | |
| You are given a number ’n’.  Find the number of digits of ‘n’ that evenly divide ‘n’.  Note:  A digit evenly divides ‘n’ if it leaves no remainder when dividing ‘n’.  Example:  Input: ‘n’ = 336  Output: 3 | |
| public static int countDigits(int n) {  // Initialize a counter variable  int cnt = 0;  while (n > 0) {  // Increment the counter for each digit  **cnt = cnt + 1;**  // Divide 'n' by 10 to remove the last digit.  n = n / 10;  }  // Return the count of digits.  return cnt;  }  **Time Complexity: O(log10N + 1),** in the while loop we divide N by 10 until it becomes 0 which takes log10N iterations and +1 is added for the assumption N is divisible by 10.  **Space Complexity : O(1)** | **Optimal Approach**  static int countDigits(int n) {  // Initialize a variable 'cnt' to  int cnt = (int) (Math.log10(n) + 1);  // The expression (int)(Math.log10(n) + 1)  // calculates the number of digits in 'n'  // and casts it to an integer.  // Adding 1 to the result accounts  // for the case when 'n' is a power of 10,    // Return the count of digits in 'n'.  return cnt;  }  **Time Complexity: O(1)**  **Space Complexity : O(1)** |

**NOTE:-** If the time complexity depends on the division , then time complexity will be logarithmic

**2. Reversing a Number:-**

|  |  |
| --- | --- |
| https://static.takeuforward.org/content/reverse-number-image1-Dvuaeusy | **Pseducode:-**  **while(N > 0){**  **// Extract the last digit of N**  **int lastDigit = N % 10;**  **revNum = (revNum \* 10) + lastDigit;**  **// Remove the last digit from N**  **N = N / 10;**  **}** |

|  |
| --- |
| **Problem statement** |
| There is a song concert going to happen in the city. The price of each ticket is equal to the number obtained after reversing the bits of a given 32 bits unsigned integer ‘n’.  Example 1:  Input: x = 123  Output: 321 |
| public static void reverseDigits(int n) {    int revNum=0;  while (n > 0) {  // Extract last digit  Int lastDigit = n%10;    **revNum = (revNum \* 10) + lastDigit;**  // Divide 'n' by 10 to remove the last digit.  n = n / 10;  }  System.out.println(revNum);  }  **Time Complexity: O(log10N + 1),** in the while loop we divide N by 10 until it becomes 0 which takes log10N iterations and +1 is added for the assumption N is divisible by 10.  **Space Complexity : O(1)** |

**3.Palindrome**

|  |
| --- |
| **Problem statement** |
| Check whether a given number ***’n’*** is a palindrome number.  **Example:** Input: 'n' = 51415 Output: true |
| public static boolean palindrome(int n) {  **int dup=n;**  int revNum=0;  while (n > 0) {  // Extract last digit  Int lastDigit = n%10;  revNum = (revNum \* 10) + lastDigit;  // Divide 'n' by 10 to remove the last digit.  n = n / 10;  }  **if(dup==revNum)**  return true;  else  return false;  }  **Time Complexity: O(log10N + 1),** in the while loop we divide N by 10 until it becomes 0 which takes log10N iterations and +1 is added for the assumption N is divisible by 10.  **Space Complexity : O(1)** |

**4.Armstrong Number**

|  |
| --- |
| **Problem statement** |
| You are given an integer *'n'*. Return *'true'* if 'n' is an Armstrong number, and *'false'* otherwise.  An Armstrong number is a number (with 'k' digits) such that the sum of its digits raised to 'kth' power is equal to the number itself. For example, 371 is an Armstrong number because 3^3 + 7^3 + 1^3 = 371. |
| import java.lang.Math;  public static boolean isArmstrong(int num) {  **int k = String.valueOf(num).length();**  int sum = 0;  int n = num;  while(n > 0){  int ld = n % 10;  **sum += Math.pow(ld, k);**  n = n / 10;  }  return sum == num ? true : false;  }  **Time Complexity: O(log10N + 1),** in the while loop we divide N by 10 until it becomes 0 which takes log10N iterations and +1 is added for the assumption N is divisible by 10.  **Space Complexity : O(1)** |

**5.print All Divisors**

|  |  |
| --- | --- |
| Print all Divisors of a given Number - Tutorial | **for (int i = 1; i <= sqrtN; ++i) {**  **if (n % i == 0) {**  **divisors.add(i);**  **if (i != n / i) {**  **divisors.add(n / i);**  **}**  **}**  **}** |
| public static int[] printDivisors(int n, int[] size) {  int[] divisors = new int[n];  int count = 0;  for (int i = 1; i <= n; i++) {  if (n % i == 0) {  divisors[count++] = i;  }  }  size[0] = count;  return divisors;  }  **Time Complexity: O(N)**  **Space Complexity : O(N)** | public static ArrayList<Integer> findDivisors(int n) {  **ArrayList<Integer> divisors = new ArrayList<>();**  // Iterate up to the square root of n  **int sqrtN = (int) Math.sqrt(n);**  for (int i = 1; i <= sqrtN; ++i) {  if (n % i == 0) {  divisors.add(i);  **if (i != n / i) {**  **divisors.add(n / i);**  **}**  }  }  return divisors;  }  **Time Complexity: O(N/2)**  **Space Complexity : O(N)** |

**6.check for prime**

|  |  |
| --- | --- |
| **Problem statement** | |
| A prime number is a positive integer that is divisible by exactly 2 integers, 1 and the number itself.  You are given a number *'n'*.  Find out whether 'n' is prime or not.  Example :  Input: 'n' = 5 Output: YES | |
| static boolean checkPrime(int n) {  int cnt = 0;  for (int i = 1; i <= n; i++) {  if (n % i == 0) {  cnt = cnt + 1;  }  }  **if (cnt == 2)**  return true;  else  return false;  }  **Time Complexity: O(N)**  **Space Complexity : O(1)** | static boolean checkPrime(int n){  int cnt = 0;  for(int i = 1; i <= Math.sqrt(n); i++){  if(n % i == 0){  cnt = cnt + 1;  if(n / i != i){  cnt = cnt + 1;  }  }  }  if(cnt == 2)  return true;  else  return false;  }  **Time Complexity: O(sqrt(N))**  **Space Complexity : O(1)** |

**Prefix sum**

|  |  |
| --- | --- |
| **Subarray Sum Equals K - Naukri Code 360** | A subarray sum is the sum of contigious elements in a subarray  **Prefix Sum** |

**Example :-**

Num[]= {1,2,3,-3, 1,1,1,4,2,-3};

k=3;

HashMap

|  |  |
| --- | --- |
| 9 | 1 |
| 12 | 1 |
| 10 | 1 |
| 5 | 1 |
| 4 | 1 |
| 6 | 1+1 |
| 3 | 1+1 |
| 1 | 1 |
| 0 | 1 |

**Count=8**

|  |  |
| --- | --- |
| **Problem statement** | |
| You are given an integer array 'arr' of size 'N' and an integer 'K'.  Your task is to find the total number of subarrays of the given array whose sum of elements is equal to k.  A subarray is defined as a contiguous block of elements in the array.  Example:  Input: ‘N’ = 4, ‘arr’ = [3, 1, 2, 4], 'K' = 6  Output: 2  Explanation: The subarrays that sum up to '6' are: [3, 1, 2], and [2, 4]. | |
| public static int findAllSubarraysWithGivenSum(int arr[], int k) {  int n = arr.length; // size of the given array.  int cnt = 0; // Number of subarrays:  for (int i = 0 ; i < n; i++) { // starting index i  for (int j = i; j < n; j++) { // ending index j  // calculate the sum of subarray [i...j]  int sum = 0;  **for (int K = i; K <= j; K++)**  **sum += arr[K];**  // Increase the count if sum == k:  if (sum == k)  cnt++;  }  }  return cnt;  }  **Time Complexity: O(N^3)**  **Space Complexity : O(1)** | public static int findAllSubarraysWithGivenSum(int arr[], int k) {  int n = arr.length; // size of the given array.  int cnt = 0; // Number of subarrays:  for (int i = 0 ; i < n; i++) { // starting index i  int sum = 0;  for (int j = i; j < n; j++) { // ending index j  // calculate the sum of subarray [i...j]  **sum += arr[K];**  // Increase the count if sum == k:  if (sum == k)  cnt++;  }  }  return cnt;  }  **Time Complexity: O(N^2)**  **Space Complexity : O(1)** |
| import java.util.Arrays;  import java.util.HashMap;  import java.util.Map;  public class Main {  public static void main(String[] args) {  int num2[]= {1,2,3,-3, 1,1,1,4,2,-3};  int k=3;  System.out.println(subarraySum(num2,k));  }  public static int subarraySum(int[] nums, int k) {  int ans = 0;  int prefix = 0;  Map<Integer, Integer> count = new HashMap<>();  count.put(0, 1);  for (final int num : nums) {  prefix += num;  ans += count.getOrDefault(prefix - k, 0);  count.merge(prefix, 1, Integer::sum);  }  return ans;  }  }  **Time Complexity: O(N) or O(N\*logN)**  **Space Complexity : O(N)** | |