Contents

[**GCD(HCF)** 3](#_Toc127004149)

[**LCM** 3](#_Toc127004150)

[**1D Array** 3](#_Toc127004151)

[**Initializing an Array** 3](#_Toc127004152)

[**Input Elements in an Array** 3](#_Toc127004153)

[**Output Elements in an Array** 3](#_Toc127004154)

[**1D\_ArrayList** 4](#_Toc127004155)

[**Initializing 1D\_Arraylist** 4](#_Toc127004156)

[**Basic Operations** 4](#_Toc127004157)

[**1.** **add** : 4](#_Toc127004158)

[**2.** **get** : 4](#_Toc127004159)

[**3.** **size** : 4](#_Toc127004160)

[**4.** **set :** 4](#_Toc127004161)

[**2D Array** 5](#_Toc127004162)

[**Initializing an Array** 5](#_Toc127004163)

[**Scanning Elements in an Array** 5](#_Toc127004164)

[**Printing Elements in an Array** 5](#_Toc127004165)

[**Equations & Expressions** 5](#_Toc127004166)

[**2D ArrayList** 6](#_Toc127004167)

[**Initializing 2D ArrayList** 6](#_Toc127004168)

[**Basic Operations** 6](#_Toc127004169)

[**1.** **get :** 6](#_Toc127004170)

[**2.** **add :** 6](#_Toc127004171)

[**3.** **set :** 7](#_Toc127004172)

[**4.** **size :** 7](#_Toc127004173)

[**String** 8](#_Toc127004174)

[**Initializing and Scanning a String** 8](#_Toc127004175)

[**Basic Operations** 8](#_Toc127004176)

[**1.** **length() :** 8](#_Toc127004177)

[**2.** **charAt() :** 8](#_Toc127004178)

[**Time Complexity** 9](#_Toc127004179)

[**Range of numbers** 9](#_Toc127004180)

[**1.** **Include last number :** 9](#_Toc127004181)

[**2.** **Exclude last number :** 9](#_Toc127004182)

[**3.** **Arithmetic Progression(AP) :** 9](#_Toc127004183)

[**4.** **Geometric Progression(GP) :** 9](#_Toc127004184)

[**Log Basics** 10](#_Toc127004185)

[**Big O notation** 11](#_Toc127004186)

[**Space Complexity** 12](#_Toc127004187)

[**Arrays** 13](#_Toc127004188)

[**Prefix Sum** 13](#_Toc127004189)

[**Carry Forward** 13](#_Toc127004190)

[**Sub Arrays** 14](#_Toc127004191)

[**2D Matrices** 14](#_Toc127004192)

[**1.** **Diagonal Elements of matrices** 14](#_Toc127004193)

[**2.** **Non-Diagonal Elements(secondary diagonal) of matrices** 14](#_Toc127004194)

[**3.** **All diagonal Elements of matrices** 15](#_Toc127004195)

[**4.** **Transpose Matrix** 15](#_Toc127004196)

[**5.** **Rotating the matrices to 90o** 16](#_Toc127004197)

[**6.** **Print boundary elements in clockwise** 16](#_Toc127004198)

[**7.** **Print every boundary elements in clockwise** 16](#_Toc127004199)

[**Sliding Window** 17](#_Toc127004200)

[**Bit Manipulation** 18](#_Toc127004201)

[**Binary to Decimal** 18](#_Toc127004202)

[**Decimal to Binary** 19](#_Toc127004203)

[**Binary Addition** 20](#_Toc127004204)

[**Bitwise Operators** 20](#_Toc127004205)

[**1.** **& (AND)** 20](#_Toc127004206)

[**2.** **| (OR)** 20](#_Toc127004207)

[**3.** **~ (NOT)** 21](#_Toc127004208)

[**4.** **^ (XOR)** 21](#_Toc127004209)

[**5.** **<< (Left Shift)** 21](#_Toc127004210)

[**6.** **>> (Right Shift)** 23](#_Toc127004211)

[**Storing and Retrieving -ve number** 24](#_Toc127004212)

[**Storing -ve number** 24](#_Toc127004213)

[**Retrieving -ve number** 24](#_Toc127004214)

[**Max and Min numbers to be stored** 24](#_Toc127004215)

[**Bitwise Properties** 25](#_Toc127004216)

[**1.** **Commutative** 25](#_Toc127004217)

[**2.** **Associative** 25](#_Toc127004218)

# **GCD(HCF)**

* Largest integer that can exactly divide both no without any remainder.
  + (n1%i == 0) && (n2%i==0)

HCF = i;

# **LCM**

* Smallest integer that is perfectly divisible by both numbers.
  + n1 \* n2 = GCD \* LCM
  + LCM = (n1 \* n2)/GCD(HCF)

# **1D Array**

## **Initializing an Array**

dataType variable[] = new int[size]

or

dataType[] variable = new int[size]

## **Input Elements in an Array**

for(int i=0; i<variable.length ;i++){

variable[i] = scn.nextType();

}

Note: In scn.nextType Type-> datatype

## **Output Elements in an Array**

for(int i=0; i<variable.length ;i++){

variable[i] = scn.nextType();

}

Note: In scn.nextType Type-> datatype

# **1D\_ArrayList**

## **Initializing 1D\_Arraylist**

ArrayList<Integer> al = new ArrayList<Integer>();

Integer, Long, Double, String

## **Basic Operations**

### **add** :

* + add() function used to add an element in the arraylist.

ArrayList<Integer> al = new ArrayList<Integer>();

al.add(element)

**Example :**

al.add(1);

al.add(-7);

**Output :** [1, -7]

### **get** :

* + get() function is used to get an element from particular index.
  + If index is grater than size of the arraylist we will be getting a error called **index out of bound.**

ArrayList<Integer> al = new ArrayList<Integer>();

al.get(index)

**Example :**

al.get(0); 🡪 1

al.get(1); 🡪 -7

### **size** :

* + size() function is used to get the size of the array list.

ArrayList<Integer> al = new ArrayList<Integer>();

al.size(); 🡪 2

### **set :**

* + set() is used to replaces the current existing value of a index.
  + set() can be used on existing indexes. If there is no index in the arraylist then it gives error.

ArrayList<Integer> al = new ArrayList<Integer>();

al.set(index, value)

**Example:**

al.get(0, 5);

al.add(1, 7);

**Output:** [5,7]

# **2D Array**

## **Initializing an Array**

dataType variable[][] = new int[row][column]

or

dataType[][] variable = new int[row][column]

## **Scanning Elements in an Array**

**for(int row=0;row<N;row++){  
 for (int col=0; col<M; col++){  
 arr1[row][col] = scn.nextType();**

**}  
}**

Note: In scn.nextType Type-> datatype

## **Printing Elements in an Array**

**for(int row=0;row<N;row++){  
 for (int col=0; col<M; col++){  
 System.*out*.print(result[row][col] + " ");}  
 System.*out*.println();  
}**

## **Equations & Expressions**

* **a.length; *//getting the row length of array***
* **a[0].length; *//getting the col length of array***
* **dimensions of array -> N\*M(row\*column)**
* **Total no of elements -> N\*M**
* **Last element location of 2D array -> N-1, M-1**

# **2D ArrayList**

* Arraylist of arraylist

## **Initializing 2D ArrayList**

ArrayList<ArrayList<Integer>> variable = new ArrayList<ArrayList<Integer>>();

Integer, Long, Double, String

* Example:

ArrayList < ArrayList <Integer>> list = new ArrayList < ArrayList <Integer>>();

ArrayList<Integer> li = new ArrayList<Integer>();

li.add(1);

li.add(2);

li.add(3)

li -> [1,2,3]

list.add(li)

list -> { [1,2,3] }

ArrayList<Integer> li2 = new ArrayList<Integer>();

li2.add(4);

li2.add(5);

li2.add(6)

li2 -> [4,5,6]

list.add(li2)

list -> { [1,2,3], [4,5,6] }

## **Basic Operations**

### **get :**

list.get(1) -> [4,5,6]

list.get(1).get(1) -> 5

### **add :**

list.get(1).add(7)

list -> { [1,2,3], [4,5,6,7] }

### **set :**

ArrayList <Integer> li3 = new ArrayList <Integer>();

li3.add(8);

li3.add(9);

li3.add(10)

li3 -> [8,9,10]

list -> { [1,2,3], [4,5,6,7] }

list.set(0,li3);

list -> { [8,9,10], [4,5,6,7] }

list.get(1).set(1,10);

list -> { [8,9,10], [4,10,6,7] }

### **size :**

* + Count of arraylists in list -> list.size() -> 2
  + Length of 1st index arraylist -> list.get(0).size() ->3

# **String**

* String is nothing but sequence of characters which is present between “”.
* Characters :
  + [a-z]
  + [A-Z]
  + Special Characters
  + [0-9]

## **Initializing and Scanning a String**

Scanner scn = new Scanner(System.in);

* **String str = scn.next(); //** 
  + Java **next()** method can read the input before the space id found. It cannot read two words separated by space. It retains the cursor in the same line after reading the input.
* **String str = scn.nextLine(); //**
  + The **nextLine()** method of Scanner class is used to take a string from the user. The nextLine() method reads the text until the end of the line. After reading the line, it throws the cursor to the next line.

## **Basic Operations**

### **length() :**

* Used to get the length of the string

String name = “Mohnish”

name.length() 🡪 7

### **charAt() :**

* Used to get character at particular index of the string.

String name = “Mohnish”

name.charAt(3) 🡪 n

# **Time Complexity**

* Sum of n natural numbers.

1 + 2 + 3 + ………………+ n = **(n \* (n+1))/2**

## **Range of numbers**

### **Include last number :**

* [a,b] -> represents that b must be include in the range of numbers from a to b.

**Example:** [3,10]

**Output:** 3 4 5 6 7 8 9 10

* To get no of number in the particular range of 2 numbers we can use the following formula **b-a+1**.

**Example:** [3,10]

**Output:** 10-3+1 = 8 // total there are 8 numbers from 3 to 10.

### **Exclude last number :**

* [a,b) -> represents that b must be exclude in the range of numbers from a to b.

**Example:** [3,10)

**Output:** 3 4 5 6 7 8 9

* To get no of number in the particular range of 2 numbers we can use the following formula **b-a**.

**Example:** [3,10]

**Output:** 10-3 = 7 // total there are 7 numbers from 3 to 10.

### **Arithmetic Progression(AP) :**

3 6 9 12 15 18 21 24 . . . . . . .n

* In the above example the difference between all numbers is 3 which is constant different and the terms are consecutive.
* Sum of n numbers in the AP can be calculated using the following formula

**n/2[2a + (n-1)d]**

n = no of elements in the sequence.

a = starting number in the sequence.

d = difference between numbers.

### **Geometric Progression(GP) :**

2 4 8 16 32 64 . . . . . . n

* In the above example the difference between all numbers is not constant but the ratio is common and the terms are consecutive terms.
* Sum of n natural numbers in GP is calculated using the following formula

**a(rn - 1) / r-1**

a = starting number of the sequence

r = ratio between the numbers

## **Log Basics**

1. logba = ?
   * To get answer we need to find **to what power b need to be raised to get a**.
   * Example

log264 = ? 🡺 a = 64, b = 2

To get the answer we need to find the power of 2(b) so that we get 64(a)

2?  = 64 🡪 26 = 64

🡺 log264 = 6

Since we need to raise the power 6 of 2 to get 64 hence 6 is the value of log264 .

1. logab = x, where b = ax
2. if N = ak => **logaN = k**

**Explanation:**

N = 2k

log2N = log2b, wher b = 2k 🡺 Applied log2 on both side of above equation

log2N = k 🡺 If we compare above equation’s RHS to 2nd property we will be getting K

N = 3k

log3N = log3b, wher b = 3k 🡺 Applied log2 on both side of above equation

log3N = k 🡺 If we compare above equation’s RHS to 2nd property we will be getting K

**Examples:**

1. int s=0;

for(int i=0; i<=100; i++){

s = s+i;

}

* The above loop iterates 101 times since in the condition both 0 and 100 are involved
* So we can assume as [0,100] => 100-0+1 🡺101 ([a,b] => b-a+1).

1. s = 0

for(int i=35; i<=87; i=i+1){

s=s+i;

}

Ans => [35,87] => 87-35+1 => **53 [(35 36 37 38…….87) 🡪total 53 terms]**

1. s = 0

for(int i=1; i<=N; i=i+1){

s=s+i;

}

Ans => [1,N] => N-1+1 => **N [(1 2 3 4 …. N) 🡪total N terms]**

1. s = 0

for(int i=1; i<=N; i=i+1){

s=s+i;

}

for(int i=1; i<=M; i=i+1){

s=s+i;

}

Ans 🡺N+M

**Explainations:**

* The first loop runs N times
* The second loop runs M times
* So if we add both loops its runs **N+M** times

1. s = 0

for(int i=1; i\*i<=N; i=i+1){

s=s+i;

}

Ans 🡺

**Explanation:**

* i\*i <=N 🡺 i<=
* [1, ] => -1 + 1 🡺

1. s = 0

for(int i=1; i\*i<=N; i=i+1){

for(int j = I; j<=N; j++){

s=s+1;

}

}

Ans 🡺N2

**Explainations:**

* The first loop runs N times
* The second loop runs N times
* Since second loop is inside first loop we need to multiply both loop hence its N2

## **Big O notation**

* Calculate no of iterations.
* Ignore lower order terms.
* Ignore constant coefficient.

Example to ignore lower order terms and constant coefficient

1. 100logN 🡺 O(logN) [ here 100 is constant coefficient ].
2. N/100 🡺 O(N) [ here 1/100 is the constant coefficient ].
3. 4N2 + 5N + 6 🡺 O(N2) [ here N2 is the higher order term so remaining lower order and constant terms are ignored ].

# **Space Complexity**

* Total space taken by the algorithm which is completely based on the variable space.
* **Example:**

1. fun(int N){ 🡺 N = 4 bits space

int x = N; 🡺 x = 4 bits space

int y = x\*x; 🡺 y = 4 bits space

long z = x + y; 🡺 z = 8 bits space

}

Total = 4+4+4+8 🡺 20bits (constant bits)

So O(1).

1. fun(int N){ 🡺 N = 4 bits space

int x = N; 🡺 x = 4 bits space

int y = x\*x; 🡺 y = 4 bits space

long z = x + y; 🡺 z = 8 bits space

int[] arr = new int[N] 🡺arr= 4N bits space

}

Total = 4+4+4+8+4N 🡺 20Nbits (20 is constant so ignored but has a higher order terms which is N)

So O(N).

1. fun(int N){ 🡺 N = 4 bits space

int x = N; 🡺 x = 4 bits space

int y = x\*x; 🡺 y = 4 bits space

long z = x + y; 🡺 z = 8 bits space

int[] arr = new int[N] 🡺arr= 4N bits space

int[] arr1 = new int[N][N] 🡺arr1 = 4N2 bits space

}

Total = 4+4+4+8+4N+4N2 🡺 20Nbits (20 is constant so ignored but has a higher order terms which is N2)

So O(N2).

# **Arrays**

* Time complexity of single element in the array is O(1).
* Time complexity of access (scan, print) of the array is O(N).

## **Prefix Sum**

* Prefix sum is the technique where you precompute & store the cumulative sum of the sequence of elements that allows fast sum calculation of any continuous range.

* Let's say we have a sequence of elements A as mentioned below-

A = {a0, a1, a2, a3, a4, a5}

* So Prefix Sum P will be calculated as

P =  {p0, p1, p2, p3, p4, p5}

where

p0 = a0

p1 = a1 + a0

p2 = a0 + a1 + a2

p3 = a0 + a1 + a2 + a3

p4 = a0 + a1 + a2 + a3 + a4

p5 = a0 + a1 + a2 + a3 + a4 + a5

* Q) Say we need to sum get sum of all elements from indices   
   [2 to 5] => [a2 + a3 + a4 + a5] or [p5 - p1]

[1 to 4] => [a1 + a2 + a3 + a4]   or [p4 - p1]  
 [0 to 4] => [a0 + a1 + a2 + a3 + a4] or [p4]

## **Carry Forward**

* Carry forward is a process where we will tracing the array from [ n-1 to 0 ] and perform required operations or conditions or both.

## **Sub Arrays**

* We know that an array is a contiguous memory block. Similarly, a sub-array is any contiguous part of that array, which may consist of any number of elements with at least one element in it.
* Let us write all the subarrays for the array: [3, 5, 1, 2, 7, 4]
* The sub-arrays for this array are:

|  |  |  |
| --- | --- | --- |
| **Index of Element** | **Subarrays Possible** | **Count** |
| 0 (element 3) | {3}, {3, 5}, {3, 5, 1}, {3, 5, 1, 2}, {3, 5, 1, 2, 7}, {3, 5, 1, 2, 7, 4} | 6 |
| 1 (element 5) | {5}, {5, 1}, {5, 1, 2}, {5, 1, 2, }, {5, 1, 2, 7, 4} | 5 |
| 2 (element 1) | {1}, {1, 2}, {1, 2, 7}, {1, 2, 7, 4} | 4 |
| 3 (element 2) | {2}, { 2, 7}, { 2, 7, 4} | 3 |
| 4 (element 7) | {7}, {7, 4} | 2 |
| 5 (element 4) | {4} | 1 |

* Total number of subarrays possible in an array of length **N = ( N\*(N+1))/2**

## **2D Matrices**

### **Diagonal Elements of matrices**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* + i = 0 -> N-1 🡺 a[i][i]

### **Non-Diagonal Elements(secondary diagonal) of matrices**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* + i = 0 -> N-1
  + j = N-1 -> 0
  + a[i][j]

### **All diagonal Elements of matrices**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* The above matrices is divided into 2 parts
* **Part 1 : Part 2:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* Column 🡪 0 – M-1 Row🡪 0 – N-1
* i 🡪 0 – N-1 i 🡪 r – N-1 (r indicates every iteration of row).
* j 🡪 c – 0(c indicate every j 🡪 M-1 – 0

iteration of column).

* + - * + a[i][j]

### **Transpose Matrix**

* Converting rows of a matrix into columns and columns of a matrix into row is called transpose of a matrix.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* If we see the above matrices it is divided into 3 parts
* Part 1 : Upper triangle

Part 2 : Diagonals

Part 3 : Lower triangle

* We have to swap upper triangle value to lower triangle.
* Since when we convert row to column and column to row the diagonals will be in its absolute location so we have to swap upper and lower triangle.
* i 🡪 0 – N-1
  + j 🡪 i+1 – N
    - Swap a[i][j] & a[j][i]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |

* After performing transpose process the matrix will as above (the blue and orange part has been swapped).

### **Rotating the matrices to 90o**

* We have to perform 2 steps to rotate matrices to 90o
* First step : we have to perform transpose matrix process.
* Second step : we have to reverse the elements for each and every columns.

### **Print boundary elements in clockwise**

|  |  |  |  |
| --- | --- | --- | --- |
| **0** | **1** | **2** | **3** |
| **11** |  |  | **4** |
| **10** |  |  | **5** |
| **9** | **8** | **7** | **6** |

* We have to use 4 loops and default value i = 0, j = 0
* Loop 1 🡪 Top wall [ k🡪 1 – N-1 , j++ ]
* Loop 2 🡪 Right wall [ k🡪 1 – N-1 , i++ ]
* Loop 3 🡪 Bottom wall [ k🡪 1 – N-1 , j-- ]
* Loop 4 🡪 Left wall [ k🡪 1 – N-1 , i-- ]

### **Print every boundary elements in clockwise**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** |
| **15** | **16** | **17** | **18** | **5** |
| **14** | **23** | **24** | **19** | **6** |
| **13** | **22** | **21** | **20** | **7** |
| **12** | **11** | **10** | **9** | **8** |

* There are 2 steps
* i = 0, j = 0
* Step 1: while loop [ sizeOfArray – N )
  + In step1 there are 4 steps
    - Loop 1 🡪 Top wall [ k🡪 1 – N-1 , j++ ] print a[i][j]
    - Loop 2 🡪 Right wall [ k🡪 1 – N-1 , i++ ] print a[i][j]
    - Loop 3 🡪 Bottom wall [ k🡪 1 – N-1 , j-- ] print a[i][j]
    - Loop 4 🡪 Left wall [ k🡪 1 – N-1 , i-- ] print a[i][j]
  + N = N – 2, i++, j++
* Step 2: if condition ( N == 1 ) print a[i][j]

## **Sliding Window**

* **Carry forward technique + fixed size of subarray = Sliding Window**
* Given N element, return max subarray sum of length k ?

**Example:**

**0 1 2 3 4 5 6 7 8 9**

ar[10] = -3 4 -2 5 3 -2 8 2 -1 4

k = 5

0 – 4 = 7 = sum

1 – 5 = 7 – (-3) + (-2) = sum – a[0] + a[5] = 8

2 – 6 = 8 – 4 + 5 = sum – a[1] + a[6] = 9

. . . . . .

. . . . . .

. . . . . .

. . . . . .

a[s] – a[e] = sum – a[s] + a[e]

# **Bit Manipulation**

* Decimal number system :– { 0 – 9 }
* 1 2 3 🡺 3 place is units(0) place, 2 place is tens(1) place, 3 place is hundreds(2) place……

3 \* 100 + 2 \* 101 + 1\*102 🡺 here 10 represents there are total 10 numbers

* **∑ Digits + Baseposition**

Here,

Digits are 1 2 3

Base is 10

Position is digits places

* So base for decimal number system is 10
* Binary number system :– { 0, 1 }
* Since in binary number there are 2 numbers the base is 2.

## **Binary to Decimal**

* **∑ Digits + Baseposition (Base = 2)**
* Example

1. 1 1 0 🡺 0 \* 20 + 1 \* 21 + 1\*22 = 0 + 2 + 4 = 6

1 1 0 = 6

1. 1 0 1 🡺 1 \* 20 + 0 \* 21 + 1\*22 = 1 + 0 + 4 = 5

1 0 1 = 5

1. 1 1 1 0 0 1 🡺 ?

1 \* 20 + 0 \* 21 + 0 \* 22 + 1 \* 23 + 1 \* 24 + 1 \* 25

1 + 0 + 0 + 8 + 16 + 32

57

(1 1 1 0 0 1)2 = (57)10

## **Decimal to Binary**

* **Number % 2 (Until the quotient is 0)**
* Example

1. 57

57%2

Base Number Remainder

2 57 1

2 28 0

2 14 0

2 7 1

2 3 1

2 1 1

2 0

Combine remainder from bottom to top 1 1 1 0 0 1

(57)10 = ( 1 1 1 0 0 1 )2

1. 25

25%2

Base Number Remainder

2 25 1

2 12 0

2 6 0

2 3 1

2 1 1

2 0

Combine remainder from bottom to top 1 1 1 0 0 1

(57)10 = ( 1 1 1 0 0 1 )2

* Repeatedly divide with 2

Until number becomes 0

Read remainder from bottom to top

## **Binary Addition**

* Rules

0 + 0 = 0

1 + 0 = 1

0 + 1 = 1

1 + 1 = 10 ( Here 1 will be carried forward )

* Example

1. 0 1 1 1 0 0 1 + 0 1 0 0 1 0 1

1

0 1 1 1 0 0 1 57

+ 0 1 0 0 1 0 1 + 37

1 0 1 1 1 1 0 94

## **Bitwise Operators**

* The following operations can be performed

### **& (AND)**

A B A&B

0 0 0

0 1 0

1 0 0

1 1 1

* + If any of input is 0, result is 0.
  + If N is a number then

N & 1 🡪 1 [ N is odd ] (set)

N & 1 🡪 0 [ N is even ] (unset)

* + N & 0 🡪 0

### **| (OR)**

A B A|B

0 0 0

0 1 1

1 0 1

1 1 1

* + If any of input is 1, result is 1.
  + If N is a number then

N | 1 🡪 N [ N is odd ]

N |1 🡪 N + 1 [ N is even ]

* + N | 0 🡪 N

### **~ (NOT)**

A ~A

0 1

1 0

* + The result will be opposite to input.

### **^ (XOR)**

A B A^B

0 0 0

0 1 1

1 0 1

1 1 0

* + If both input is same, result is 0.
  + If both input is different, result is 1.
  + If N is a number then

N ^ N 🡪 0

N ^ 0 🡪 N

N & 0 🡪 0

### **<< (Left Shift)**

* + The left shift operator moves all bits by a given number of bits to the left.
  + **Example**

a = 0 0 1 1 1 0 0 1 57 🡪 57 X 20

a << 1 = 0 1 1 1 0 0 1 0 ( 1 bit has been moved to left ) 114 🡪 57 X 21

a << 2 = 1 1 1 0 0 1 0 0 ( 2 bit has been moved to left ) 228 🡪 57 X 22

a << 3 = 1 1 0 0 1 0 0 0 ( 3 bit has been moved to left ) 456 🡪 57 X 23

**a << n = a X 2n** ( If there is no overflow [only 0 moved to left] )

N, if ith bit is 1 (Set)

* + N | ( 1 << i )

> N, if ith bit is 0 (UnSet)

* + - **Example**

N = 57 🡪 1 1 1 0 0 1

Case 1

i = 3

N | (1 << i)

1 1 1 0 0 1 | 1 << 3

1 1 1 0 0 1

0 0 1 0 0 0

1 1 1 0 0 1 🡪 57 = N

* + - * If we see the above example the ith bit is set(1) so the operations gives N value as the result.

Case 2

i = 2

N | (1 << i)

1 1 1 0 0 1 | 1 << 2

1 1 1 0 0 1

0 0 0 1 0 0

1 1 1 1 0 1 🡪 61 = >N

* + - * If we see the above example the ith bit is unset(0) so the operations gives >N value as the result.

1 << i, if ith bit is 1

* + N & ( 1 << i )

0, if ith bit is 0

* + - **Example**

N = 57 🡪 1 1 1 0 0 1

Case 1

i = 3

N & (1 << i)

1 1 1 0 0 1 & 1 << 3

1 1 1 0 0 1

0 0 1 0 0 0

0 0 1 0 0 0 🡪 1 << 3 = 1 << i

* + - * If we see the above example the ith bit is set(1) so the operations gives 1<<i value as the result.

Case 2

i = 2

N & (1 << i)

1 1 1 0 0 1 & 1 << 2

1 1 1 0 0 1

0 0 0 1 0 0

0 0 0 0 0 0 🡪 0

* + - * If we see the above example the ith bit is unset(0) so the operations gives 0 value as the result.

< N, if ith bit is 1

* + N ^ ( 1 << i )

> N, if ith bit is 0

* + - **Example**

N = 57 🡪 1 1 1 0 0 1

Case 1

i = 3

N ^ (1 << i)

1 1 1 0 0 1 ^ 1 << 3

1 1 1 0 0 1

0 0 1 0 0 0

1 1 0 1 1 1 🡪 <N

* + - * If we see the above example the ith bit is set(1) so the operations gives <N value as the result.

Case 2

i = 2

N & (1 << i)

1 1 1 0 0 1 & 1 << 2

1 1 1 0 0 1

0 0 0 1 0 0

1 1 1 1 0 1 🡪 >N

* + - * If we see the above example the ith bit is unset(0) so the operations gives >N value as the result.

### **>> (Right Shift)**

* + The right shift operator moves all bits by a given number of bits to the right.
  + **Example**

a = 57 = 0 0 1 1 1 0 0 1 57 🡪 57/20

a >> 1 = 0 0 0 1 1 1 0 0 ( 1 bit has been moved to right ) 28 🡪 57/21

a >> 2 = 0 0 0 0 1 1 1 0 ( 2 bit has been moved to right ) 14 🡪 57/22

a >> 3 = 0 0 0 0 0 1 1 1 ( 3 bit has been moved to right ) 7 🡪 57/23

**a>>n = a/2n**

## **Storing and Retrieving -ve number**

### **Storing -ve number**

* Step 1 : remove -ve sign of number
* Step 2 : convert to 1’s complement(toggle all the bits)
* Step 3 : convert to 2’s complement( add 1 to number at step2)

### **Retrieving -ve number**

* Step 1: consider the number at step 3 in the storing process
* Step 2: consider the highest position bit as MSB
* Step 3: considered MSB bit is -ve number
* Step 4: add all the digital number of 1’s with MSB

**Example**

N = -57

Storing

Step 1: 57 (1 1 1 0 0 1)

Step 2: 1 1 1 0 0 1 🡪 0 0 0 1 1 0

Step 3: 0 0 0 1 1 0 + 1 = 0 0 0 1 1 1

Retrieving

Step 1: 0 0 0 1 1 1

Step 2: 0 0 0 1 1 1 (highlighted bit is the MSB)

Step 3: 5th position is MSB -25 = -32

Step 4:

0 0 0 1 1 1

-25 24 23 22 21 20

-64 0 0 4 2 1

-64 + 4 + 2 + 1 = -57

## **Max and Min numbers to be stored**

Range Bits Min Max

8 bits -27 27 – 1

int 32 bits -231 232 – 1

long 64 bits -263 263 – 1

## **Bitwise Properties**

### **Commutative**

a ^ b = b ^ a

a & b = b & a

a | b = b | a

### **Associative**

(a & b) & c = a & (b & c)

(a ^ b) ^ c = a ^ (b ^ c)

(a | b) | c = a | (b | c)