Primitive Types

Java Bitwise and Shift Operators

In Java, bitwise operators perform operations on integer data at the individual bit-level. Here, the integer data includes byte, short, int, and long types of data.

There are 7 operators to perform bit-level operations in Java.

Operator	Meaning	Description
|	Bitwise OR	Returns 1 if at least one of the operands is 1. Otherwise, returns 0.
&	Bitwise AND	Returns 1 if and only if both the operands are 1. Otherwise, returns 0.
^	Bitwise XOR	Returns 1 if and only if one of the operands is 1. However, if both the operands are 0 or if both are 1, then the result is 0.
~	Bitwise Complement	The bitwise complement operator is a unary operator (works with only one operand). It changes binary digits 1 to 0 and 0 to 1.
<<	Left Shift	The left shift operator shifts all bits towards the left by a certain number of specified bits. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.
>>	Signed Right Shift	The signed right shift operator shifts all bits towards the right by a certain number of specified bits. When we shift any number to the right, the least significant bits (rightmost) are discarded and the most significant position (leftmost) is filled with the sign bit, i.e., 0 (for positive sign) or 1 (for negative sign).
>>>	Unsigned Right Shift	Java also provides an unsigned right shift. Here, the vacant leftmost position is filled with **0** instead of the sign bit.

1. Java Bitwise OR Operator (|)

The bitwise OR operator returns 1, if at least one of the operands is 1. Otherwise, it returns 0. The following truth table demonstrates the working of the bitwise OR operator. Let a and b be two operands that can only take binary values i.e. 1 or 0.

a	b	a | b	Example
0	0	0	Let's look at the bitwise OR operation of two integers 12 and 25.
0	1	1	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	1	Bitwise OR Operation of 12 and 25
1	1	1	00001100 | 00011001 _____ 00011101 = 29 (In Decimal)

Programme:

```
public class BitwiseOperator {

    public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        // bitwise OR between 12 and 25
        result = number1 | number2;
        System.out.println(result);    // prints 29

    }
}
```

2. Java Bitwise AND Operator (&)

The bitwise AND operator returns 1 if and only if both the operands are 1. Otherwise, it returns 0. The following table demonstrates the working of the bitwise AND operator. Let a and b be two operands that can only take binary values i.e. 1 and 0.

a	b	a & b	Example
0	0	0	Let's look at the bitwise AND operation of two integers 12 and 25.
0	1	0	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	0	Bitwise AND Operation of 12 and 25
1	1	1	00001100 & 00011001 _____ 00001000 = 8 (In Decimal)

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number1 = 12, number2 = 25, result;

                // bitwise AND between 12 and 25
                result = number1 & number2;
                System.out.println(result);    // prints 8
        }
}
```

3. Java Bitwise XOR Operator (^)

The bitwise XOR operator returns 1 if and only if one of the operands is 1. However, if both the operands are 0 or if both are 1, then the result is 0. The following truth table demonstrates the working of the bitwise XOR operator. Let a and b be two operands that can only take binary values i.e. 1 or 0.

a	b	a ^ b	Example
0	0	0	Let's look at the bitwise OR operation of two integers 12 and 25.
0	1	1	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	1	Bitwise XOR Operation of 12 and 25
1	1	0	00001100 ^ 00011001 _____ 00010101 = 21 (In Decimal)

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number1 = 12, number2 = 25, result;

           // bitwise XOR between 12 and 25
           result = number1 ^ number2;
           System.out.println(result);    // prints 21
        }
}
```

4. Java Bitwise Complement Operator (~)

The bitwise complement operator is a unary operator (works with only one operand). It changes binary digits **1** to **0** and **0** to **1**.

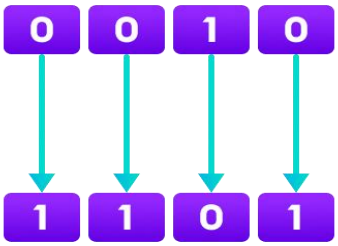


Figure 1: Java Bitwise Complement Operator

It is important to note that the bitwise complement of any integer **N** is equal to **- (N + 1)**. For example, consider an integer **35**. As per the rule, the bitwise complement of **35** should be **- (35 + 1)** = **- 36**. Now let's see if we get the correct answer or not.

```
   00100011 = 35 (In Decimal)
~_________________
   11011100 = -36 (In Decimal) / 220 (In Decimal)
```

In the above example, we get that the bitwise complement of **00100011** (**35**) is **11011100**. Here, if we convert the result into decimal we get **220**. However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result **11011100** is also equivalent to **- 36**.

To understand this we first need to calculate the binary output of **- 36**.

2's Complement

In binary arithmetic, we can calculate the binary negative of an integer using 2's complement.

1's complement changes **0** to **1** and **1** to **0**. And, if we add **1** to the result of the 1's complement, we get the 2's complement of the original number. For example,

```
// compute the 2's complement of 36
36 = 00100100 (In Binary)

1's complement = 11011011

2's complement:

  11011011
+         1
___________
  11011100
```

Here, we can see the 2's complement of **36** (i.e. **-36**) is **11011100**. This value is equivalent to the bitwise complement of **35**. Hence, we can say that the bitwise complement of **35** is **- (35 + 1) = -36**.

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number = 35, result;

                // bitwise complement of 35
                result = ~number;
                System.out.println(result);    // prints -36
        }
}
```

5. Java Left Shift Operator (<<)

The left shift operator shifts all bits towards the left by a certain number of specified bits. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.

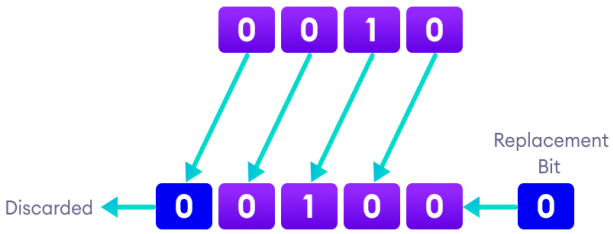


Figure 2: Java 1 bit Left Shift Operator

For example, we have a 4-digit number. When we perform a 1 bit left shift operation on it, each individual bit is shifted to the left by **1** bit. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.

Programme:

```
public class ShiftOperator {

        public static void main(String[] args) {

                int number = 2;
```

```
        // 2 bit left shift operation
        int result = number << 2;
        System.out.println(result);    // prints 8
    }
}
```

6. Java Signed Right Shift Operator (>>)

The signed right shift operator shifts all bits towards the right by a certain number of specified bits. When we shift any number to the right, the least significant bits (rightmost) are discarded and the most significant position (leftmost) is filled with the sign bit, i.e., 0 (for positive sign) or 1 (for negative sign).

Example: right shift of 8

In binary, the 8 is 1000.

8 >> 2 (perform 2 bit right shift)

1000 >> 2 = 0010 (equivalent to 2)

Here, we are performing the right shift of **8** (i.e. sign is positive). Hence, there no sign bit. So, the leftmost bits are filled with **0** (represents positive sign).

Example: right shift of -8

In binary, the 8 is 1000.

1's complement = 0111

2's complement:

```
 0111
  + 1
 _____
 1000
```

Signed bit = 1

// perform 2 bit right shift
8 >> 2:
1000 >> 2 = 1110 (equivalent to -2)

Here, we have used the signed bit **1** to fill the leftmost bits.

Programme:

```
public class ShiftOperator {
```

```
        public static void main(String[] args) {

                int number1 = 8;
                int number2 = -8;

                // 2 bit signed right shift
                System.out.println(number1 >> 2);    // prints 2
                System.out.println(number2 >> 2);    // prints -2
        }
}
```

7. Java Unsigned Right Shift Operator (>>>)

Java also provides an unsigned right shift. Here, the vacant leftmost position is filled with **0** instead of the sign bit.

For example,

// unsigned right shift of 8
8 = 1000

8 >>> 2 = 0010

// unsigned right shift of -8

-8 >>> 2 = 1073741822

Programme:

```
public class ShiftOperator {

        public static void main(String[] args) {

           int number1 = 8;
           int number2 = -8;

           // 2 bit signed right shift
           System.out.println(number1 >>> 2);    // prints 2
           System.out.println(number2 >>> 2);    // prints 1073741822

        }
}
```

Time and Space Complexity

- Sometimes, there are more than one way to solve a problem.
- Need to compare the performance different algorithms and choose the best one to solve a particular problem.

- For analyzing an algorithm, we mostly consider
  - time complexity
  - space complexity

- Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Time Complexity of an algorithm

- Time Complexity of an algorithm is the representation of the amount of time taken by the algorithm to complete the execution.

- Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

Space complexity of an algorithm

- Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

- Space needed by an algorithm is equal to the sum of the following two components

  - A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

  - A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

- Space complexity S(p) of any algorithm p is S(p) = A + S(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I.

Example:

Lets start with a simple example, addition of two n-bit integers.

SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 – Stop

- **Time Complexity**: Addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

- **Space Complexity:** Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Example:

Lets start with a simple example. Suppose you are given an array A with size N. an integer x present/absent in A and have to find if x exists in array A.

Simple solution to this problem is traverse the whole array A and check if the any element is equal to x.

```
for i : 0 to (length of A-1)
        if A[i] is equal to x
                return TRUE
return FALSE
```

- Each of the operation in computer takes approximately constant time c.

- The number of lines of code executed is actually depends on the position of value of x.
- During analyses of algorithm, mostly we will consider worst case scenario, i.e., when x is not present in the array A.

- In the worst case, the if condition will run N times where N is the length of the array A. So in the worst case, total execution time will be (N * c + c). (N * c) for the if condition and for the return statement.

- Space Complexity S(N).

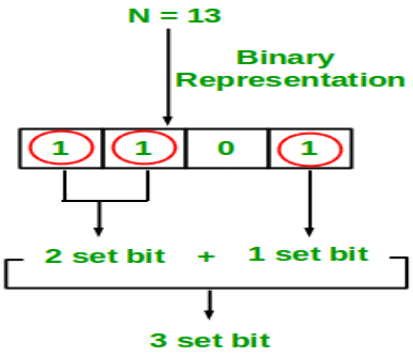
Problem: Write an efficient program to count number of 1s in the binary representation of an integer.

Let an integer number is provided. Usually the computer stores the numbers in binary number format. The binary number is formed as the combination of 0's and 1's. The digit 1 is known as set bit.

Example:

- Let the given integer is 13.

- The binary representation of 13 is 1101 and has 3 set bits or 3 number of 1's present.



Approach 1 - Simple Method: Move through each of the binary bit of an integer, check if a bit is 1, then increment the set bit counter. After passing through all the bits present in the binary number return the counter value. The following program tests one bit at-a-time starting with the least-significant bit. It illustrates shifting and masking.

Programme:

```
import java.util.Scanner;

public class CountSetBitSimple {

        public static void main(String[] args) {

                int x;
                short count;

                System.out.print("Enter an Integer Number: ");
                Scanner sc = new Scanner(System.in);
                x = sc.nextInt();

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the Number: " +
                                                                Long.toBinaryString(x));

                count = countBits(x);
                System.out.print("The Number of set bits in word " + x + " is " + count);

        }

        public static short countBits(int x) {

                short numBits = 0;

                while (x != 0) {

                        numBits += (x & 1) ;
                        x >>>= 1;
```

```
            }
            return numBits;
        }
}
```

Output:

Enter an Integer Number: 7
Binary representation of the Number: 111
The Parity of word 7 is 3

Time Complexity:

- Since we perform *0(1)* computation per bit, the time complexity is *0(N)*, where N is the number of bits in the integer word.

Approach 2 - Modulo Method: The idea behind this approach is similar to conversion of decimal number to binary number. Let a given number N. If N%2 ==1, means the list significant bit of that number is 1. Similarly, if N%2 ==0, means the list significant bit of that number is 0. This approach keeps track of number of 1 present in the given number.

Programme:

```
import java.util.Scanner;

public class CountSetBitModulo {

    public static void main(String[] args) {

        int x;
        short count;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));

        count = countbitsmodulo(x);
        System.out.print("The Number of set bits in word " + x + " is " + count);

    }

    public static short countbitsmodulo(int x) {

        short numBits = 0;
```

```
            while (x != 0) {

                if(x % 2 == 1) {
                    numBits += 1;
                }

                //x = x / 2
                x >>>= 1;
            }
            return numBits;
        }
}
```

Output:

Enter an Integer Number: 6
Binary representation of the Number: 110
The Parity of word 6 is 2

Time Complexity:

- This approach always processing whether the bit is either 0 or 1. For computation of each bit require *0(1)*. Hence, the time complexity of N bits number is *0(N)*.

Approach 3 – Brian Kernighan's Algorithm: By subtracting 1 from integer, an effective observation is that all the bits after the rightmost set bit are flipped including the rightmost set bit.

Example:

$N_1 = 9$ (1001 is Binary representation of 9)
$N_2 = N_1 - 1 = 8$ (1000 is Binary representation of 8)
$N_3 = N_2 - 1 = 7$ (0111 is Binary representation of 7)

In the above example we can see that the rightmost set bit in 8 is now unset and all the bits to its right are flipped. This un-setting of the rightmost set bit is useful at the same time we need suppress the toggling effect which is unwanted. One such algorithm to achieve it is the Brian Kernighan's Algorithm.

Brian Kernighan's Algorithm:

1 Initialize count: = 0

2 If integer n is not zero

(a) Do bitwise & with (n-1) and assign the value back to n
n = n & (n-1)

(b) Increment count by 1

(c) go to step 2

3 Else return count

Programme:

```
import java.util.Scanner;

public class CountSetBitsBrianKernighan {

	public static void main(String[] args) {

		int x;
		short count;

		System.out.print("Enter an Integer Number: ");
		Scanner sc = new Scanner(System.in);
		x = sc.nextInt();

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
							Long.toBinaryString(x));

		count = countbitsBrianKernighan(x);
		System.out.print("The Number of set bits in word " + x + " is " + count);

	}

	public static short countbitsBrianKernighan(int x) {

		short numBits = 0;

		while (x != 0) {

			x &= (x - 1);
			numBits += 1;
		}
		return numBits;
	}
}
```

Output:

```
Enter an Integer Number: 9
Binary representation of the Number: 1001
The Parity of word 9 is 2
```

Time Complexity:

- This algorithm goes through as much iteration as there is set bits. Hence, the time complexity is O(k), where k is the number of set bits.

Approach 4 – Lookup Table: This approach processes bits in a group. Here we group some bits together, say 4, and store the corresponding numbers of set bits in a lookup table. For example, we have all the 4 bits integers and the corresponding set bit count store in the lookup table. Now, whenever we are given an integer we can perform constant time lookups.

Here the number of lookups for a given integer is equal to log n that is the number of binary bits in the given integer divided by k where k is the number of bits grouped together, in this case, 4. To calculate the counts of set bits, we can simply add the counts returned by the lookups.

Decimal	Binary	Count Set bits		Decimal	Binary	Count Set bits
0	0000	0		8	1000	1
1	0001	1		9	1001	2
2	0010	1		10	1010	2
3	0011	2		11	1011	3
4	0100	1		12	1100	2
5	0101	2		13	1101	3
6	0110	2		14	1110	3
7	0111	3		15	1111	4

Example:

n = 254 (Binary representation: 1111 1110)

Lower Nibble: 1110 = 14 (3 set bits)

Upper Nibble: 1111 = 15 (4 set bits)

Output = 7

Programme:

```
import java.util.Scanner;

public class CountSetBitsLookupTable {

    public static void main(String[] args) {

        int x;
        short count;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();
```

```
            //Convert Decimal to Binary number
            System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

            count = countbitsLookupTable(x);
            System.out.print("The Number of set bits in word " + x + " is " + count);

    }

    public static short countbitsLookupTable(int x) {

            short numBits = 0;
             //index          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
            int[] Table = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2,   2,  3,  2,  3,  3,  4};

            while (x != 0) {

                    numBits += Table[x & 0x0f];  //0x0f means 0000 1111
                    x >>>= 4;
            }
            return numBits;
    }
}
```

Output:

Enter an Integer Number: 234
Binary representation of the Number: 11101010
The Parity of word 234 is 5

Time Complexity:

- The time complexity of this approach is O(N/group size). Here Nis the number of bits in number and group size is the number of bits in group, say 4 here.

Problem: Write an efficient programme to compute the parity of a word.

- Parity of a number refers to whether it contains an odd or even number of 1-bits.
- The number has “odd parity”, if it contains odd number of 1-bits and is “even parity” if it contains even number of 1-bits.
- The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0.

    1 → Parity of the set bit is odd.

    0 → Parity of the set bit is even.

- Example:
    - The parity of 1011 is 1, because there are 3 ones.
    - The parity of 10001000 is 0, because there are 2 ones.
- Parity checks are used to detect single bit errors in data storage and communication.

Approach 1 - Brute Force:

- The brute-force algorithm perform bit-wise right shift on the given number iteratively & check the number of 1s seen at least significant bit (LSB).

Programme:

```
import java.util.Scanner;

public class ParityBruteForce {

	public static void main(String[] args) {

		int x;
		short count;

		System.out.print("Enter an Integer Number: ");
		Scanner sc = new Scanner(System.in);
		x = sc.nextInt();

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

		count = parity (x);
		System.out.print("The Parity of word " + x + " is " + count);
	}

	public static short parity(long x){

		short result = 0;

		while (x != 0) {

			result ^= (x & 1);
			x >>>= 1;
		}

		return result;
	}
}
```

Output:

Enter an Integer Number: 3
Binary representation of the Number: 11
The Parity of word 3 is 0

Advantages:

- The solution is very easy to understand & implement.

Disadvantages:

- Processing all the bits manually, so this approach is hardly efficient at scale.

Time Complexity:

- O(n) where n is the total number of bits in the binary representation of the given number.

Approach 2 - Clear all the set bits one by one:

This approach can just go over only set bits. Let the given number N. In bitwise computation, this approach can clear the rightmost set bit with the following operation:

N = N & (N -1)

Example:

N =	1010	(10 in Decimal)
N – 1 =	1001	(9 in Decimal)
N & (N - 1) =	1000	(8 in Decimal)

In the above example this approach successfully cleared the lowest set bit (2nd bit from the right side in 1010). If repeat the above process, the number N will become 0 at a certain point in time. In this approach need to scan only k bits where k is the total number of set bits in the number & k <= length of the binary representation.

Programme:

```
import java.util.Scanner;

public class ClearSetBits {

    public static void main(String[] args) {

        long x;
        short count;
```

```
            System.out.print("Enter an Integer Number: ");
            Scanner sc = new Scanner(System.in);
            x = sc.nextInt();

            //Convert Decimal to Binary number
            System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));

            count = parity(x);
            System.out.print("The Parity of word " + x + " is " + count);
        }

        public static short parity(long x){

            short result = 0;

            while (x != 0) {

                result ^= 1;
                x &= (x - 1); // Drops the lowest set bit of x.
            }
            return result;
        }
}
```

Output:

Enter an Integer Number: 10
Binary representation of the Number: 1010
The Parity of word 10 is 0

Advantages:

- Simple to implement.
- More efficient than brute force solution.

Disadvantages:

- It's not the most efficient solution.

Time Complexity:

- O(k) where k is the total number of set bits in the number. For example, for 10001010, $k = 3$.

Approach 3 - Caching:

The problem statement refers to computing the parity for a very large number of words. The size of a long type number is 64 bits or 8 bytes, so total memory size required is: 2^{64} bits of storage, which is of the order of ten trillion Exabyte.

- To compute the parity of a 64-bit integer by grouping its bits into four non overlapping 16 bit sub words.
- Compute the parity of each 16 sub words, because 2^{16} = 65536 is relatively small, which makes it feasible to cache the parity of all 16-bit words using an array.
- Then XOR parity of these four sub results with each other, since XOR is associative & commutative. The order in which we fetch those groups of bits & operate on them does not even matter.

Example: Illustrate the approach with an 8-bit word 11101010 and with a lookup table for 2-bit words.

- The cache is < 0,1,1, 0 >, these are the parities of (00), (01), (10), (11), respectively.
- To compute the parity of (11101010) we would compute the parities of (11), (10), (10), (10).
- By table lookup we see these are 0, 1, 1, 1 respectively.
- Final result is the parity of 0, 1, 1, 1 which is 0 ^ 1 ^ 1 ^ 1 = 1.

Programme:

```
import java.util.Scanner;

public class ParityCatch {

	public static void main(String[] args) {

		long x;
		short count;

		System.out.print("Enter an Integer Number: ");
		Scanner sc = new Scanner(System.in);
		x = sc.nextInt();

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

		count = parity(x);
		System.out.print("The Parity of word " + x + " is " + count);

	}

	public static short parity(long x){
```

```
        final int W0RD_SIZE = 2;
        final int BIT_MASK = 3; //00000011 (in Binary)
                          //index  0 1  2 3
        int precomputedParity[] = {0, 1, 1, 0};

        return (short) (
              precomputedParity[(int)((x >>> (3 * W0RD_SIZE)) & BIT_MASK)]
            ^ precomputedParity[(int)((x >>> (2 * W0RD_SIZE)) & BIT_MASK)]
            ^ precomputedParity[(int)((x >>> W0RD_SIZE) & BIT_MASK)]
            ^ precomputedParity[(int)(x & BIT_MASK)]);
    }
}
```

Output:

Enter an Integer Number: 234
Binary representation of the Number: 11101010
The Parity of word 234 is 1

Advantages:

- At the cost of relatively small memory for the cache, we get better efficiency since we are reusing a group of 16-bit numbers across inputs.
- This solution can scale well as we are serving millions of numbers.

Disadvantages:

- If this algorithm needs to be implemented in an ultra-low memory device, the space complexity has to be well thought of in advance in order to decide whether it's worth it to accommodate such amount of space.

Time Complexity:

- O(N / WORD_SIZE) where N is the total number of bits in the binary representation. All right / left shift & bitwise &, |, ~ etc operations are word level operations which are done extremely efficiently by CPU. Hence their time complexity is supposed to be O(1).

Approach 4 - Using XOR & Shifting operations:

The XOR of two bits is 0, if both the bits are 0 or both bits are 1; otherwise it is 1. XOR has the property of being associative, as well as commutative, i.e., the order in which we perform the XORs does not change the result. Since parity occurs when an odd number of set bits are there in the binary representation, we can use XOR operation to check if an odd number of 1 exists there.

- For example, the parity of 64 bit numbers $< b_{63}, b_{62}, b_{61,\ldots,} b_3, b_2, b_1, b_0 >$, equals the parity of the XOR of $< b_{63}, b_{62}, b_{61,\ldots,} b_{35}, b_{34}, b_{33}, b_{32} >$ and $< b_{31}, b_{30}, b_{29,\ldots,} b_3, b_2, b_1, b_0 >$. Hence right shift the number by half of the total number of digits, XOR that shifted

number with the original number, assign the XOR-ed result to the original number. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit. The least-significant bit extract by bitwise-AND with (00000001).

Example: Illustrate the approach with an 8-bit word 11010111

1. Split the 8–bit integer into 4–bit chunks and XOR-ed between them:
```
    0111   (considering last 4 bits of  11010111)
^   1101   (considering fast 4 bits of  11010111)
----------------------
    1010   The parity of (11010111) is the same as the parity of (1101) XORed with
           (0111)
```

2. Split the 4–bit integer into 2–bit chunks and XOR-ed between them:
```
    10   (considering last 2 bits of  1010)
^   10   (considering fast 2 bits of  1010)
----------------------
    00       The parity of (1010) is the same as the parity of (10) XORed with (10)
```

3. Split the 2–bit integer into 1–bit chunks and XOR-ed between them:
```
    0   (considering last 1 bits of  00)
^   0   (considering fast 1 bits of  00)
----------------------
    0        The parity of (00) is the same as the parity of (0) XORed with (0)
```

4. The last bit is the result and to extract it, for which bitwise AND with 1
```
    0
&   1
----------------------
    0        The even parity
```

Programme:

```
import java.util.Scanner;

public class ParityXORShifting {

    public static void main(String[] args) {

        long x;
        short count;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));
```

```
            count = parity(x);
            System.out.print("The Parity of word " + x + " is " + count);

        }

        public static short parity(long x){

            x ^= x >>> 32;
            x ^= x >>> 16;
            x ^= x >>> 8;
            x ^= x >>> 4;
            x ^= x >>> 2;
            x ^= x >>> 1;

            return (short) (x & 0x1);       //means 1
        }
}
```

Output:

```
Enter an Integer Number: 7
Binary representation of the Number: 111
The Parity of word 7 is 1
```

Advantages:

1. No extra space uses word-level operations to compute the result.

Disadvantages:

1. Might be little difficult to understand for developers.

Time Complexity:

- O(log n) where n is the total number of bits in the binary representation.

Problem: Write a programme to Swap bits in a given integer number.

- Given a number x and two positions (indices i and j from the right side) in the binary representation of x, swaps the bits at indices i and j, and returns the result. It is also given that the two sets of bits do not overlap.

- This problem can be solved - First by checking if the two bits at given positions are the same or not.

    - If they are same (i.e., one is 0 and the other is 0) no need to be swap, because the swap does not change the integer.

- If they are not same (i.e., one is 0 and the other is 1), then we can XOR them with 1 << position. This logic will work because:

  XOR with 1 will toggle the bits
  0 ^ 1 = 1
  1 ^ 1 = 0

  XOR with 0 will have no impact
  0 ^ 0 = 0
  1 ^ 0 = 1

- A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit (LSB), and the bit at index 63 corresponding to the most significant bit (MSB).

- **Example:** bit swapping for an 8-bit integer.

  - **Input:**
    - x = 73 (In binary 01001001)
    - i = 1 (Start from the second bit from the right side)
    - j = 6 (Start from the 7th bit from the right side)

  - **Output:**
    - **11** (In binary 0001011)

  - **Explanation:**

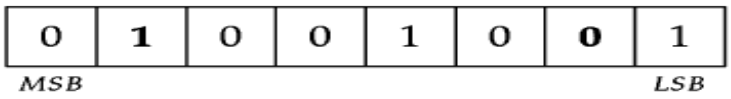


Figure Before Swapping: The 8-bit integer 73 can be viewed as array of bits, with the LSB being at index 0.

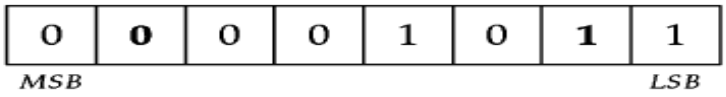


Figure After Swapping: The result of swapping the bits at indices 1 and 6, with the LSB being at index 0. The corresponding integer is 11.

Programme:

```
import java.util.Scanner;

public class SwapBitProg1 {

    public static void main(String[] args) {

        long x, y;
        int i, j;

        System.out.print("Enter an Integer Number: ");
```

```
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter 1st index value i (start from the right side): ");
        i = sc.nextInt();

        System.out.print("Enter 2nd index value j (start from the right side): ");
        j = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (Before Swapping):
                                                    " + Long.toBinaryString(x));

        y = swapBits(x, i, j);
        System.out.println("After Swapping the integer is " + y);
        System.out.println("Binary representation of the Number (After Swapping): "
                                                    + Long.toBinaryString(y));
    }

    public static long swapBits(long x, int i, int j) {

        // Extract the i-th and j-th bits, and see if they differ.
        if( ((x >>> i) & 1) != ((x >>> j) & 1)){

            // We will swap them by flipping their values.
            // Select the bits to flip with bitMask.
            long bitMask = (1L << i) | (1L << j);
            x ^= bitMask;
        }
        return x;
    }
}
```

Output:

```
Enter an Integer Number: 73
Enter 1st index value i (start from the right side): 1
Enter 2nd index value j (start from the right side): 6
Binary representation of the Number (Before Swapping): 1001001
After Swapping the integer is 11
Binary representation of the Number (After Swapping): 1011
```

Time Complexity:

- The time complexity is *0(1),* independent of the word size.

Problem: Write a programme to Reverse bits in a given integer number.

- Write a program that takes a 64-bit word and returns the 64-bit word consisting of the bits of the input word in reverse order.

- For example, if the input is alternating 1s and Os, i.e., (1010...10), the output should be alternating Os and 1s, i.e., (0101...01).

- **Example:** Illustrate the approach with an 8-bit word 10010011 and with a lookup table for 2-bit words.

    - Build an array based lookup-table, rev = ((00), (10), (01), (11)).

    - Let the input is (10010011).

    - Now divide the 8-bit word into 4 groups and each group has 2-bits, such as (10), (01), (00), (11).

    - Its reverse is rev(ll), rev(00), rev(01),rev(10), i.e., (11001001).

Programme:

```
import java.util.Scanner;

public class ReverseBitsProg1 {

    public static void main(String[] args) {

        long x;
        long reverse;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (Before Reverse): "
                                                    + Long.toBinaryString(x));

        reverse = reverseBits(x);
        System.out.println("After Reverse the integer is " + reverse);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (After Reverse): " +
                                                    Long.toBinaryString(reverse));
    }

    public static long reverseBits(long x) {

        final int WORD_SIZE = 2;
        final int BIT_MASK = 3; //00000011 (in Binary)

                                //index   0  1  2  3
        int precomputedReverse[] = {0, 2, 1, 3};
```

```
        return (precomputedReverse [(int)(x & BIT_MASK)] << (3 * WORD_SIZE)
        | precomputedReverse [(int)((x >>> WORD_SIZE) & BIT_MASK)] << (2 * WORD_SIZE)
        | precomputedReverse[(int)((x >>> (2 * WORD_SIZE)) & BIT_MASK)] << WORD_SIZE
        | precomputedReverse[(int)((x >>> (3 * WORD_SIZE)) & BIT_MASK)]);
        }
}
```

Output:

```
Enter an Integer Number: 147
Binary representation of the Number (Before Reverse): 10010011
After Reverse the integer is 201
Binary representation of the Number (After Reverse): 11001001
```

Time Complexity:

- O(N / WORD_SIZE) where N is the total number of bits in the binary representation. All right / left shift & bitwise &, |, ~ etc operations are word level operations which are done extremely efficiently by CPU. Hence their time complexity is supposed to be O(1).

Given an integer, write a programme to find a closest integer with the same weight.

- The weight of a non-negative integer can be defined as, the number of set bits (i.e., 1's) present in the binary representation of that integer.
    - **Example:** let the given integer is 92. The binary representation of 92 is 1011100. The weight of 92 is 4, because the binary representation of 92 consist 4 numbers of set bits.
- The closest integer with the same weight can be defined as, suppose a given positive integer x and the task is to find an integer y such that:
    - Both x and y weights must be same, i.e., the number of set bits (1's) present in y is equal to the number of set bits (1's) present in x.
    - The integer y which is not equal to integer x, i.e., x != y.
    - The difference between x and y is as small as possible, i.e., |y – x| is minimum.
    - Assume x is not 0 or all 1s.

- **Example:** For integer 7 (0111 in binary), the possible closest integer numbers are 11 (1011 in binary), 13(1101 in binary) and 14 (1110 in binary), because all three have the same weight; but 11 is the closet integer to 7 because the difference between 7 and 11 is minimum.

- **Approach:**

  - The problem can be viewed as "which differing bits to swap in a bit representation of a number, so that the resultant number is closest to the original?"

  - Since the number of bits in both the numbers has to be the same, if a set bit is flipped then an unset bit will also have to be flipped.

  - Now the problem reduces to choosing two bits, say bit at index i and bit at index j for the flipping. Both index i and j are from the LSB (least significant bit) and always i > j.

  - Suppose one bit at index i is flipped and another bit at index j is flipped. To preserve the weight of an integer, the bit at index i has to be different from the bit in j, otherwise the flips lead to an integer with different weight.

  - Then the absolute value of the difference between the original integer and the new one is $2^i - 2^j$. To minimize this, i has to be as small as possible and j has to be as close to i. This means the smallest i can be is the rightmost bit that's different from the LSB, and j must be the very next bit, such that i-j=1. In summary, the correct approach is to swap the two rightmost consecutive bits that differ.

Programme:

```
import java.util.Scanner;

public class ClosestIntegerProg1 {

    public static void main(String[] args) {

        long x;
        long closeInt;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the given integer: " +
                                                    Long.toBinaryString(x));

        closeInt = closestIntSameBitCount(x);
        System.out.println("The closest integer of " + x + " is " + closeInt);
```

```
            //Convert Decimal to Binary number
            System.out.println("Binary representation of the closest integer: " +
                                                    Long.toBinaryString(closeInt));

    }

    public static long closestIntSameBitCount(long x) {

            int NUM_UNSIGN_BITS = 64;
            // x is assumed to be non-negative so we know the leading bit is 0. We
            // restrict to our attention to 63 LSBs.
            for (int i = 0; i < NUM_UNSIGN_BITS - 1; ++i) {

                if ((((x >>> i) & 1) != ((x >>> (i + 1)) & 1))) {

                    x ^= (1L << i) | (1L << (i + 1)); // Swaps bit-i and bit-(i + 1).
                    return x ;
                }
            }
            // Throw error if all bits of x are 0 or 1.
            throw new IllegalArgumentException("All bits are 0 or 1");
    }
}
```

Output:

Enter an Integer Number: 11
Binary representation of the given integer: 1011
The closest integer of 11 is 13
Binary representation of the closest integer: 1101

Time Complexity:

- The time complexity is *0(n)*, where *n* is the integer width.

Write a program that multiplies two nonnegative integers without arithmetic operators.

- The only operators are allowed to use are
    - assignment,
    - the bitwise operators >>, <<, |, &, ~, ^
    - Equality checks and Boolean combinations thereof.
- The constraints imply,
    - Cannot use increment or decrement, or test if $x < y$.

- **Approach:** The algorithm uses shift and add to achieve a much better time complexity.
    - Let x and y are the two non-negative integers.
    - To multiply *x* and *y, first* initialize the result to 0.
    - Then iterate through the bits of *x,*
        - If the k^{th} bit of *x* is 1, and then add $2^k y$ to the result.
        - The value $2^k y$ can be computed by left-shifting *y* by *k.*
        - Since cannot use add operator directly, hence need to implement.
            - Perform binary number addition, i.e., compute the sum bit-by-bit, and "rippling" the carry along.

- **Example:** Multiply two non negative integers 13 and 9.

x = 13 (1101 in binary)
y = 9 (1001 in binary)

- Iterate through the bits of x (such as 1101) from least significant bit (LSB)
- In the first iteration, since the LSB (i.e., 0th bit) of 1101 is 1, set the result to 1001.

result = 1001

- In the second iteration, the second bit (i.e., 1th bit) of 1101 is 0, so no change in result.

result = 1001

- In the third iteration, the third bit (i.e., 2nd bit) is 1, hence shift 1001 to the left by 2 (i.e., 1001 << 2), obtain 100100 and it added (i.e., binary addition performed) to the result (i.e., 1001)

```
    100100          // obtain by 1001 << 2
+     1001          // result (previous),
----------------------------------
    101101          // result (current)
```

- In the fourth or final iteration (because only 4 bits present), the fourth and final bit (i.e., 3rd bit) of 1101 is 1, hence shift 1001 to the left by 3 (i.e., 1001 << 3), obtain 1001000, which is added to the result (i.e., 101101)

```
    1001000         // obtain by 1001 << 3
+    101101         // result (previous),
----------------------------------
    1110101         // result (current),
```

- The result 1110101 is equivalent to integer 117.

Programme:

```
import java.util.Scanner;

public class MultiplicationProg1 {

    public static void main(String[] args) {

        long x, y;
        long result;

        System.out.print("Enter first Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter second Integer Number: ");
        y = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the first Integer: " +
                                                    Long.toBinaryString(x));

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the second Integer: " +
                                                    Long.toBinaryString(y));

        result = multiply(x, y);
        System.out.println("The multiplication of " + x + " and " + y + " is " + result);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of multiplication result: " +
                                                    Long.toBinaryString(result));

    }

    public static long multiply(long x, long y){

        long sum = 0;

        while (x != 0) {

            // Examines each bit of x.
            if ((x & 1) != 0) {

                sum = add(sum, y);
            }
```

```
            x >>>= 1;
            y <<= 1;
        }
        return sum;
    }

    private static long add(long a, long b) {

        long sum = 0, carryin = 0, k = 1, tempA = a, tempB = b;

        while (tempA != 0 || tempB != 0) {

            long ak = a & k, bk = b & k ;

            long carryout = (ak & bk) | (ak & carryin) | (bk & carryin);

            sum |= (ak ^ bk ^ carryin);

            carryin = carryout << 1;

            k <<= 1;

            tempA >>>= 1;

            tempB >>>= 1;
        }
        return sum | carryin;
    }
}
```

Output:

```
Enter second Integer Number: 13
Enter second Integer Number: 9
Binary representation of the first Integer: 1101
Binary representation of the second Integer: 1001
The multiplication of 13 and 9 is 117
Binary representation of multiplication result: 1110101
```

Time Complexity:

- The time complexity of addition is 0(n), where n is the width of the operands. Since we do n additions to perform a single multiplication, the total time complexity is $0(n^2)$.

Given two positive integers, such as x and y. Write a programme to compute x/y (their quotient), using only the addition, subtraction, and shifting operators.

Approach:

- Initially find the largest k such that $2^k y <= x$, subtract $2^k y$ from *x* and add 2^k to the quotient.
- In subsequent iterations, test $2^{k-1} y$, $2^{k-2} y$, $2^{k-3} y$... with x, and update the x and quotient.
- The process will stop, when the y > x.

Example: Divide two positive integers 11 and 2.

x = 11 (1011 in binary)
y = 2 (10 in binary)

- The large k value is determined as 2, (i.e. k=2), because $2^2 \times 2 < 11$ and $2^3 \times 2 > 11$. [$\because 2^k y <= x$]
- Then subtract $2^k y$ from x, i.e., $2^2 \times 2 = 8$ (in binary 1000) from 11 (in binary 1011).

```
      1011        // x = 11 (in integer)
  -   1000        // 2^k y=2^2 x 2 = 8
-----------------------------------
      0011        // updating x to 3 (in binary 0011)
```

- Add, $2^k = 2^2 = 4$ (in binary 100) to the quotient, initially the quotient was zero.

```
      0000        // initially the quotient is zero
  +   0100        // 2^k = 2^2 = 4 (in binary 100)
-----------------------------------
      0100        // updating quotient to 4 (in binary 100)
```

- Continue with x = 3 (in binary 0011) and determined the largest k such that $2^k y < 3$ (in binary 0011), hence k=0, show that, $2^\circ \times 2 = 2$ (in binary 0010). Then subtract $2^k y$ from x, i.e., $2^\circ \times 2 = 2$ (in binary 0010) from 3 (in binary 0011).

```
      0011        // x = 3(in integer)
  -   0010        // 2^k y=2^0 x 2 = 2
-----------------------------------
      0001        // updating x to 1 (in binary 0001)
```

- Add, $2^k = 2^0 = 1$ (in binary 0001) to the quotient

```
      0100        // quotient
  +   0001        // 2^k = 2^0 = 1 (in binary 0001)
```

```
---------------------------------
        0101        // updating quotient to 5 (in binary 101)
```

- Now the *y >x, hence stop processing and the* quotient is 5 (in binary101).

Programme:

```
import java.util.Scanner;

public class DivisionProg1 {

public static void main(String[] args) {

        long x, y;
        long quotient;

        System.out.print("Enter first Integer (Dividend): ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter second Integer (Divisor): ");
        y = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the first Integer (Dividend): " +
                                                        Long.toBinaryString(x));

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the second Integer (Divisor): " +
                                                        Long.toBinaryString(y));

        quotient = divide(x, y);
        System.out.println("The Division of " + x + " and " + y + " is " + quotient);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of quotient: " +
                                                    Long.toBinaryString(quotient));
    }


    public static long divide(long x, long y) {

        long result = 0;
        int k = 4; //32

        long yPower = y << k ;

        while (x >= y) {

            while (yPower > x) {
```

```
                    yPower >>>= 1;
                    --k ;
                }
                result += 1L << k;
                x -= yPower;
            }
            return result;
        }
}
```

Output:

Enter first Integer (Dividend): 11
Enter second Integer (Divisor): 2
Binary representation of the first Integer (Dividend): 1011
Binary representation of the second Integer (Divisor): 10
The Division of 11 and 2 is 5
Binary representation of quotient: 101

Time Complexity:

- If it takes n bits to represent x/y, there are 0(n) iterations. If the largest k such that $2^k y <= x$ is computed by iterating through k, each iteration has time complexity O(n). This leads to $O(n^2)$ algorithm.

Write a program that takes a double x and an integer y and returns x^y.

Approach:

- To develop an algorithm that works for general *y*, it is instructive to look at the binary representation of *y*, as well as properties of exponentiation, specifically $x^{(y_0 + y_1)} = x^{y_0} * x^{y_1}$

- **Example:** Suppose, y = 5 (101 in binary), then $\mathbf{x^y}$ can be determined as

$$\boldsymbol{x^{(101)_2} = x^{(101)_2 + (1)_2} = x^{(100)_2} \times x = x^{(10)_2} \times x^{(10)_2} \times x}$$

- Assume *y* is nonnegative:
    - If the least significant bit (LSB) of *y* is 0, the result is $(x^{y/2})^2$
    - If the least significant bit (LSB) of *y* is 1, the result is $x * (x^{y/2})^2$
- When *y* is negative, the only changes
    - Replacing *x by 1/x* and *y* by *-y*.

Programme:

```
import java.util.Scanner;

public class PowerProg1 {

        public static void main(String[] args) {

                double x;
                int y;
                double result;

                System.out.print("Enter first Double Number: ");
                Scanner sc = new Scanner(System.in);
                x = sc.nextDouble();

                System.out.print("Enter second Integer Number: ");
                y = sc.nextInt();

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the first Integer: " +
                                                        Long.toBinaryString((long)x));

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the second Integer: " +
                                                                Long.toBinaryString(y));

                result = power(x, y);
                System.out.println("The " + x + "to the power " + y + " is " + result);

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the " + x + " to the power " + y +
                                                        " is " + Long.toBinaryString((long)result));
        }


public static double power(double x, int y){

                double result = 1.0;
                long power = y;

                //check power is positive
                //or negative
                if (y < 0) {
                        power = -power;
                        x = 1.0 / x;
                }
                while (power != 0) {
```

```
                    // If power is odd, multiply
                    // x with result
                    if ((power & 1) != 0) {
                            result *= x;
                    }

                    x *= x ;                // Change x to x^2
                    power >>>= 1;           //power must be even now
            }

            return result;
        }
}
```

Output:

Enter first Integer Number: 2
Enter second Integer Number: 5
Binary representation of the first Integer: 10
Binary representation of the second Integer: 101
The 2.0to the power 5 is 32.0
Binary representation of the 2.0 to the power 5 is 100000

Time Complexity:

- The number of multiplications is at most twice the index of y's MSB, implying an *0(n)* time complexity.

Write a program which takes an integer and returns the integer corresponding to the digits of the input written in reverse order.

Example:
The given integer is of 42, the reverse is 24.
The given integer is of -314, the reverse is -413.

Approach:

- Let the input be x and x > 0.
- First, find the remainder of the given number by using the modulo (%) operator (i.e., x % 10).
- Multiply the variable reverse by 10 and add the remainder into it.
- The subsequent digits are found by dividing the number by 10, (i.e., x % 10).
- Repeat the above steps until the number becomes 0.

- if x < 0, then record its sign, solve the problem for |x|, and apply the sign to the result.

Example: Reverse the number 1234.

- Consider three variables named number (the number to be reversed), remainder = 0 (it stores the remainder), reverse = 0 (it stores the reverse number).

- Iteration 1:

  number = 1234
  remainder = 1234 % 10 = 4
  reverse = 0 * 10 + 4 = 0 + 4 = 4
  number = 1234 / 10 = 123

  Now the value of the number and reverse variable is 123 and 4, respectively.

- Iteration 2:

  number = 123
  remainder = 123 % 10 = 3
  reverse = 4 * 10 + 3 = 40 + 3 = 43
  number = 123 / 10 = 12

  Now the value of the number and reverse variable is 12 and 43, respectively.

- Iteration 3:

  number = 12
  remainder = 12 % 10 = 2
  reverse = 43 * 10 + 2 = 430 + 2 = 432
  number = 12 / 10 = 1

  Now the value of the number and reverse variable is 1 and 432, respectively.

- Iteration 4:

  number = 1
  remainder = 1 % 10 = 1
  reverse = 432 * 10 + 1 = 4320 + 1 = 4321
  number = 1 / 10 = 0

  Now the variable number becomes 0. Hence, we get the reverse number 4321.

Programme:

```
import java.util.Scanner;

public class ReverseProg1 {
```

```
    public static void main(String[] args) {

        int x;
        long reverse;

        System.out.print("Enter an Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        reverse = reverse(x);
        System.out.println("The reverse integer of " + x + " is " + reverse);
    }

    public static long reverse(int x){

        long result = 0;
        long xRemaining = Math.abs(x);

        while (xRemaining != 0) {

            result = result * 10 + xRemaining % 10;
            xRemaining /= 10;
        }

        return x < 0 ? -result : result;
    }
}
```

Output:

Enter an Integer Number: 1234
The reverse integer of 1234 is 4321

Time Complexity:

- The time complexity is ***0(n),*** where ***n*** is the number of digits in ***x.***

Write a program that takes an integer and check that number is palindrome or not.

- A Palindrome Number is a number that remains the same number when it is reversed. For example, 131 is a palindrome, because when its digits are reversed, it remains the same number.

- Examples of palindrome Number:

0, 1, 7, 11, 121, 393, 34043, 111, 555, 48084

Approach:

- If the input is negative, then the given integer cannot be palindrome Number, since it begins with a minus symbol (i.e., -)
- If the input is positive, then iteratively compare pair wise digits, such as the least significant digit (LSB) and the most significant digit (MSB).
- Let x be an inputted integer number and n is the number of digits, then

$$n = \lfloor log_{10} x \rfloor + 1$$

- The least significant digit (LSB) and the most significant digit (MSB) can be extracted from x as follows
    - The least significant digit (LSB) is x%10
    - The most significant digit (MSB) is $x/10^{n-1}$
- If all the pair wise digits (i.e., all compared LSB and MSB digits) are matched, then process return true (i.e., the number is a palindrome).
- If any mismatch occurs between the pair wise digits (i.e., LSB and MSB digits) then stop the process and return false (i.e., the number is not a palindrome).

Example: Suppose the input integer is 157751

- In Integer 157751, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 1 and 1. Since these are equal, then update the integer to 5775.
- In Integer 5775, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 5 and 5. Since these are equal, so update the integer to 77.
- In Integer 77, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 7 and 7. Since these are equal and no more digits, the program return true.

Programme:

```
import java.util.Scanner;

public class PalindromeProg1 {

    public static void main(String[] args) {

        int x;
        boolean result;

        System.out.print("Enter an Integer Number: ");
```

```
		Scanner sc = new Scanner(System.in);
		x = sc.nextInt();

		result = isPalindromeNumber(x);    // true or false

		if(result) {
			System.out.println("The integer " + x + " is palindrome.");
		}
		else {
			System.out.println("The integer " + x + " is not palindrome.");

		}
	}


public static boolean isPalindromeNumber(int x) {

		if (x < 0) {

			return false;
		}

		final int numDigits = (int) (Math . floor (Math . log10(x))) + 1;

		int msdMask = (int)Math .pow(10 , numDigits - 1);

		for (int i = 0; i < (numDigits / 2); ++i) {

			if (x / msdMask != x % 10) {

				return false ;
			}

			x %= msdMask; // Remove the most significant digit of x.

			x /= 10; // Remove the least significant digit of x.

			msdMask /= 100;
		}

		return true ;
	}
}
```

Output:

Case -1: (True)
Enter an Integer Number: 157751
The integer 157751 is palindrome.

Case -2: (True)
Enter an Integer Number: 152751
The integer 152751 is not palindrome.

Time Complexity:

- The time complexity is O(n), and the space complexity is O(1). The space complexity is O(1), because this programme directly extracting the digits from the input.

Write a program which tests if two rectangles have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.

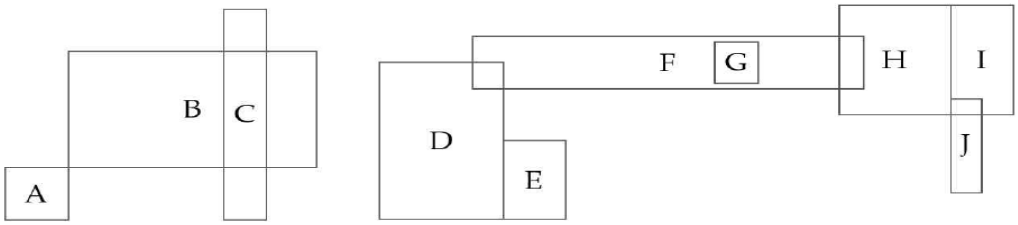


Figure 5.2: Examples of XY-aligned rectangles.

In Figure 5.2, There are many qualitatively different ways in which rectangles can intersect, e.g.,

- The Rectangle A and B share a common corner
- Two rectangles form a cross (B and C)
- The Rectangle D and E share a common side.
- The rectangle D and F have partial overlap
- One contains the other (such as Rectangle F contains G)
- Two rectangles form a tee (such as F and *H)*, etc.

Programme:

```
import java.util.Scanner;

class Rectangle {

    int x, y, width, height;

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this. width = width;
```

```
            this. height = height;
        }

        void display(){

            System.out.println(" x: " + x  + ", y: " + y + ", width: " + width + ", hight: " + height);
        }
}


public class RectangleIntersectProg1 {

        public static Rectangle intersectRectangle(Rectangle R1 , Rectangle R2) {

            if (!islntersect(R1 , R2)) {
                return new Rectangle(0, 0, -1, -1); // No intersection.
            }

            //At which point rectangle are intersect
            return new Rectangle(

            Math.max(R1.x , R2.x), Math.max(R1.y , R2.y),

            Math.min(R1.x + R1.width, R2.x + R2.width) - Math.max(R1.x , R2.x),

            Math.min(R1.y + R1.height, R2.y + R2.height) - Math.max(R1.y , R2.y));
        }

        //rectangle are intersect or not intersect
        public static boolean islntersect(Rectangle Rl , Rectangle R2) {

            return Rl.x <= R2.x + R2.width && Rl.x + Rl.width >= R2.x
                    && Rl.y <= R2.y + R2.height && Rl.y + Rl.height >= R2.y;
        }

        public  static void main(String[] args) {

            Rectangle R1 = new Rectangle(2, 3, 4, 5);
            R1.display();

            Rectangle R2 = new Rectangle(4, 5, 10, 12);
            R2.display();

            boolean result = islntersect(R1, R2);

            //rectangle are intersect or not intersect
            if(result) {

                System.out.println(" Rectangles are intersect.");
            }
```

```
                else {

                        System.out.println(" Rectangles are not intersect.");
                }

                //At which point rectangle are intersect
                Rectangle R3 = intersectRectangle(R1, R2);
                R3.display();
        }
}
```

Output:

Case -1: (Rectangles are intersect)

```
 x: 2, y: 3, width: 4, hight: 5
 x: 4, y: 5, width: 10, hight: 12
 Rectangles are intersect.
 x: 4, y: 5, width: 2, hight: 3
```

Case -2: (Rectangles are not intersect)

```
x: 2, y: 3, width: 4, hight: 5
x: 7, y: 10, width: 10, hight: 12
Rectangles are not intersect.
x: 0, y: 0, width: -1, hight: -1
```

Time Complexity:

- The time complexity is *0(1)*, since the number of operations is constant.