BINARY PALINDROME

Algorithm:

isPalindrome(x)

1) Find number of bits in x using sizeof() operator.
2) Initialize left and right positions as 1 and n respectively.
3) Do following while left ‘l’ is smaller than right ‘r’.

a) If bit at position ‘l’ is not same as bit at position ‘r’, then return false.

b) Increment ‘l’ and decrement ‘r’, i.e., do l++ and r––.

4) If we reach here, it means we didn’t find a mismatching bit.

To find the bit at a given position, we can use the idea similar to this post. The expression “**x & (1 << (k-1))**” gives us non-zero value if bit at k’th position from right is set and gives a zero value if k’th bit is not set.

Input:

Input 1= 9.

Input 2=10.

Input 3= 17.

Expected Output:

Binary of 9 : 1 0 0 1

It is a binary palindrome.

Binary of 10 : 1 0 1 0.

It is not a binary palindrome.

Binary of 17: 1 0 0 0 1

It is a binary palindrome.

Solution:

```
import java.util.*;
import java.lang.*;

public class Binary_palindrome
{

    public static long reverseBits(long n)
    {
        long rev = 0;


        while (n > 0)
        {
            rev <<= 1;


            if ((n & 1) == 1)
                rev ^= 1;

            n >>= 1;
        }
        return rev;
    }
    public static boolean isPalindrome(long n)
    {

   long rev = reverseBits(n);
        return (n == rev);
    }
```

```java
    public static void main(String args[])
    {
        long n = 9;
        if (isPalindrome(n))
            System.out.println("Yes");
        else
            System.out.println("No");
    }

}
```

Booth’s Algorithm:

This is a Java Program to implement Booth Algorithm. This is a program to compute product of two numbers by using Booth’s Algorithm. This program is implemented for multiplying numbers in the range -n to +n. However same principle can be extended to other numbers too.

Input:

Multiplier (m1) = 7.

Multiplicand (m2)= -7.

Product of two numbers=binary (m1*m);

Output:

```
Enter two integer numbers

m1=7 , m2= -7
```

```
A : 0111 0000 0
S : 1001 0000 0
P : 0000 1001 0

P : 1100 1100 1
P : 0001 1110 0
P : 0000 1111 0
P : 1100 1111 1

Result : 7 * -7 = -49
```

Solution:

```
import java.util.Scanner;
 public class Booth
{
    public static Scanner s = new Scanner(System.in);
    /** Function to multiply **/
    public int multiply(int n1, int n2)
    {
        int[] m = binary(n1);
        int[] m1 = binary(-n1);
        int[] r = binary(n2);
        int[] A = new int[9];
        int[] S = new int[9];
        int[] P = new int[9];
```

```
for (int i = 0; i < 4; i++)
{
    A[i] = m[i];
    S[i] = m1[i];
    P[i + 4] = r[i];
}
display(A, 'A');
display(S, 'S');
display(P, 'P');
System.out.println();

for (int i = 0; i < 4; i++)
{
    if (P[7] == 0 && P[8] == 0);
        // do nothing
    else if (P[7] == 1 && P[8] == 0)
        add(P, S);
    else if (P[7] == 0 && P[8] == 1)
        add(P, A);
    else if (P[7] == 1 && P[8] == 1);
        // do nothing

    rightShift(P);
```

```
        display(P, 'P');
    }
    return getDecimal(P);
}
/** Function to get Decimal equivalent of P **/
public int getDecimal(int[] B)
{
    int p = 0;
    int t = 1;
    for (int i = 7; i >= 0; i--, t *= 2)
        p += (B[i] * t);
    if (p > 64)
        p = -(256 - p);
    return p;
}
/** Function to right shift array **/
public void rightShift(int[] A)
{
    for (int i = 8; i >= 1; i--)
        A[i] = A[i - 1];
}
/** Function to add two binary arrays **/
public void add(int[] A, int[] B)
```

```
{
    int carry = 0;
    for (int i = 8; i >= 0; i--)
    {
        int temp = A[i] + B[i] + carry;
        A[i] = temp % 2;
        carry = temp / 2;
    }
}
/** Function to get binary of a number **/
public int[] binary(int n)
{
    int[] bin = new int[4];
    int ctr = 3;
    int num = n;
    /** for negative numbers 2 complement **/
    if (n < 0)
        num = 16 + n;
    while (num != 0)
    {
        bin[ctr--] = num % 2;
        num /= 2;
    }
```

```
    return bin;
}
/** Function to print array **/
public void display(int[] P, char ch)
{
    System.out.print("\n"+ ch +" : ");
    for (int i = 0; i < P.length; i++)
    {
        if (i == 4)
            System.out.print(" ");
        if (i == 8)
            System.out.print(" ");
        System.out.print(P[i]);
    }
}
/** Main function **/
public static void main (String[] args)
{
    Scanner scan = new Scanner(System.in);
    System.out.println("Booth Algorithm Test\n");
    /** Make an object of Booth class **/
    Booth b = new Booth();
     System.out.println("Enter two integer numbers\n");
```

```
        int n1 = scan.nextInt();

        int n2 = scan.nextInt();

        int result = b.multiply(n1, n2);

        System.out.println("\n\nResult : "+ n1 +" * "+ n2 +" = "+ result);

    }

}
```

Euclidean Algorithm:

In mathematics, the Euclidean algorithm, or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two integers (numbers), the largest number that divides them both without a remainder.

Basic Euclidean Algorithm for GCD

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger (we reduce a larger number), GCD doesn’t change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find remainder 0.

Input :

Write a program to find GCD implementing Euclid algorithm.

A1=GCD(30,10)

A2=GCD(15,20)

Output:

GCD(15, 20) = 5.

GCD(30, 10) = 10

Solution:

```
import java.util.*;
import java.lang.*;
class Euclid
{
   // extended Euclidean Algorithm
   public static int gcd(int a, int b)
   {
      if (a == 0)
         return b;

      return gcd(b%a, a);
   }
   public static void main(String[] args)
   {
      int a = 10, b = 15, g;
      g = gcd(a, b);
      System.out.println("GCD(" + a +  " , " + b+ ") = " + g);
      a = 35; b = 10;
      g = gcd(a, b);
      System.out.println("GCD(" + a +  " , " + b+ ") = " + g);
      a = 31; b = 2;
      g = gcd(a, b);
      System.out.println("GCD(" + a +  " , " + b+ ") = " + g);

   }
}
```