KARATSUBA ALGORITHM

The goal of the algorithm is designed because the design space is surprisingly rich.

Its time complexity is follows and do not it as time complexity for this is very important and is sometimes do asked in interview questions.

Where n is the number of digits of the numbers multiplying. It is discussed by multiplying two big integer numbers to show internal working step by step.

The goal is to reduce the space complexity for which the integer numbers terms will be broken down in such a way to x and y are broken into a set of digits as the logic behind it is divide and conquer. If the numbers are smaller there is no need to multiply, standard mutilation of two integers is preferred.

The algorithm is standardized for 4 digits for sake of understanding. One can multiply as many digits taken into sets.

Algorithm Steps:

1. Compute starting set (a*c)
2. Compute set after starting set may it be ending set (b*d)
3. Compute starting set with ending sets
4. Subtract values of step 3 from step2 from step1
5. Pad up (Add) 4 zeros to the number obtained from Step1, step2 value unchanged, and pad up two zeros to value obtained from step4.

Input: x = 1234, y = 5678

Output:

x = 1234

y = 5678

a = 12, b = 34

c = 56, d = 78

Step 1: a * c = 172
Step 2: b * d = 2652
Step 3: (a+b)(c+d) = 134 * 36 = 6164
Step 4: 6164 - 2652 - 172 = 2840
Step 5: 1720000 + 2652 + 284000 = 7006652

Solution:

```
import java.util.Random;

// Main class
class KaratSuba{

    // Main driver method
    public static long mult(long x, long y) {

        // Checking only if input is within range
        if (x < 10 && y < 10) {

            // Multiplying the inputs entered
```

```
    return x * y;
}

// Declaring variables in order to
// Find length of both integer
// numbers x and y
int noOneLength = numLength(x);
int noTwoLength = numLength(y);

// Finding maximum length from both numbers
// using math library max function
int maxNumLength
    = Math.max(noOneLength, noTwoLength);

// Rounding up the divided Max length
Integer halfMaxNumLength
    = (maxNumLength / 2) + (maxNumLength % 2);

// Multiplier
long maxNumLengthTen
    = (long)Math.pow(10, halfMaxNumLength);

// Compute the expressions
```

```
        long a = x / maxNumLengthTen;
        long b = x % maxNumLengthTen;
        long c = y / maxNumLengthTen;
        long d = y % maxNumLengthTen;



        // Compute all mutilpying variables
        // needed to get the multiplication
        long z0 = mult(a, c);
        long z1 = mult(a + b, c + d);
        long z2 = mult(b, d);

        long ans = (z0 * (long)Math.pow(10, halfMaxNumLength * 2) +
                ((z1 - z0 - z2) * (long)Math.pow(10, halfMaxNumLength) +
z2));

        return ans;

    }

     // Method 1
    // To calculate length of the number
    public static int numLength(long n)
```

```
{
    int noLen = 0;
    while (n > 0) {
        noLen++;
        n /= 10;
    }

    // Returning length of number n
    return noLen;
}

 // Method 2
// Main driver function
public static void main(String[] args)
{
    // Showcasing karatsuba multiplication

  // Case 1: Big integer lengths
    long expectedProduct = 1234 * 5678;
    long actualProduct = mult(1234, 5678);

    // Printing the expected and corresponding actual product
    System.out.println("Expected 1 : " + expectedProduct);
```

```
  System.out.println("Actual 1 : " + actualProduct + "\n\n");

  assert(expectedProduct == actualProduct);


  expectedProduct = 102 * 313;
  actualProduct = mult(102, 313);

  System.out.println("Expected 2 : " + expectedProduct);
  System.out.println("Actual 2 : " + actualProduct + "\n\n");

assert(expectedProduct == actualProduct);

  expectedProduct = 1345 * 63456;
  actualProduct = mult(1345, 63456);

  System.out.println("Expected 3 : " + expectedProduct);
  System.out.println("Actual 3 : " + actualProduct + "\n\n");

assert(expectedProduct == actualProduct);

  Integer x = null;
  Integer y = null;
```

```
        Integer MAX_VALUE = 10000;
        // Boe creating an object of random class
        // inside main() method
        Random r = new Random();
        for (int i = 0; i < MAX_VALUE; i++) {
           x = (int) r.nextInt(MAX_VALUE);
           y = (int) r.nextInt(MAX_VALUE);
           expectedProduct = x * y;
           if (i == 9999) {
             // Prove assertions catch the bad stuff.
               expectedProduct = 1;
           }
           actualProduct = mult(x, y);
            // Again printing the expected and
           // corresponding actual product
           System.out.println("Expected: " + expectedProduct);
           System.out.println("Actual: " + actualProduct + "\n\n");

           assert(expectedProduct == actualProduct);
        }
    }
}
```

Longest Sequence of 1’s after flipping a bit

An **efficient solution** is to walk through the bits in the binary representation of the given number. We keep track of the current 1’s sequence length and the previous 1’s sequence length. When we see a zero, update the previous Length:

1. If the next bit is a 1, the previous Length should be set to the current Length.
2. If the next bit is a 0, then we can’t merge these sequences together. So, set the previous Length to 0.

We update max length by comparing the following two:

1. The current value of max-length
2. Current-Length + Previous-Length .

- **Result = return max-length+1** (// add 1 for flip bit count)

Input:

13 , 15, 1775.

Output:

13 → 4

1775 → 8

15 → 5

Solution:

```
class Longest_sequence_after_flip_1’s
{

   static int flipBit(int a)
```

```
{
    /* If all bits are l, binary representation
    of 'a' has all 1s */
    if (~a == 0)
    {
        return 8 * sizeof();
    }

    int currLen = 0, prevLen = 0, maxLen = 0;
    while (a != 0)
    {
        // If Current bit is a 1
        // then increment currLen++
        if ((a & 1) == 1)
        {
            currLen++;
        }

        // If Current bit is a 0 then
        // check next bit of a
        else if ((a & 1) == 0)
        {
            /* Update prevLen to 0 (if next bit is 0)
            or currLen (if next bit is 1). */
            prevLen = (a & 2) == 0 ? 0 : currLen;

            // If two consecutively bits are 0
            // then currLen also will be 0.
            currLen = 0;
        }

        // Update maxLen if required
        maxLen = Math.max(prevLen + currLen, maxLen);
```

```
            a >>= 1;
        }


        return maxLen + 1;
    }

    static byte sizeof()
    {
        byte sizeOfInteger = 8;
        return sizeOfInteger;
    }

    // Driver code
    public static void main(String[] args)
    {
        // input 1
        System.out.println(flipBit(13));

        // input 2
        System.out.println(flipBit(1775));

        // input 3
        System.out.println(flipBit(15));
    }
}
```

Swap two nibbles in a byte

A nibble is a four-bit aggregation, or half an octet. There are two nibbles in a byte.
Given a byte, swap the two nibbles in it.

For example 100 is be represented as 01100100 in a byte (or 8 bits). The two nibbles are (0110) and (0100). If we swap the two nibbles, we get 01000110 which is 70 in decimal.

Input:

100

Output:

70

Solution:

```
class Swap_two_Nibbles {

static int swapNibbles(int x)

{

   return ((x & 0x0F) << 4 | (x & 0xF0) >> 4);

}


// Driver code

public static void main(String arg[])

{
```

```
    int x = 100;
    System.out.print(swapNibbles(x));
}
}
```