Chapter 5 - Bit Manipulation

// **Bit Facts & Tricks**

x ^ 0000 = x
x ^ 1111 = ~x
x ^ x = 0000
x & 0000 = 0000
x & 1111 = 1111
x & x = x
x | 0000 = x
x | 1111 = 1111
x | x = x

// **Two's Complement and Negative Numbers** → Two's complement of an N-bit number (N is the number of bits used excluding the sign bit) is the complement of the number with respect to 2^N

e.g. → 4-bit integer → 1 bit for sign and 3 bits for the value → complement with respect to 2^3 (8) → 8 = 1000 → 3 is 110 → complement is 1000 - 110 = 101 → -3 = 1101 (first bit is the sign)

// binary representation of -K (negative K) as a N-bit number is **concat(1, 2^N-1 - K)**

// Another way to look at it is to invert the bits in the positive representation and then add 1 → concat this with the sign bit

// Arithmetic vs. Logical Right Shift → there are two types of logical shift operators

- arithmetic right shift operator essentially divides by two → we shift values to the right but fill in the new bits with the value of the sign → we use the >> operator
- logical right shift operator visually shifts the bits → we use the >>> operator → in a negative number we shift bits and put 0 in the most significant bit

// **Common Bit Tasks**

```
// Get Bit
boolean getBit(int num, int i) {
        // 1 << i = 0001000 if i = 3
        return ((num & (1 << i)) != 0);
}
```

```
// Set Bit
int setBit(int num, int i) {
	return num | (1 << i);
}

// Clear Bit
int clearBit(int num, int i) {
	int mask =  ~(1 << i);
	return num & mask;
}

// Clear Bits from Most Significant trough i (inclusive)
int clearBitsMSThroughI(int num, int i) {
	int mask = (1 << i) - 1;
	return num & mask;
}

// Clear Bits from i through 0 (inclusive)
int clearBitsIThrough0(int num, int i) {
	// -1 = 11111111111
	int mask = (-1 << (i + 1));
}

// Update bit
int updateBit(int num, int i, boolean bitIs1) {
	int value = bitIs1 ? 1 : 0;
	int mask = ~(1 << i);
	return (num & mask) | (value << i);
}
```

// **Insertion**

```
public class Question {

	public static int updateBits(int n, int m, int i, int j) {
		int allOnes = ~0;  // 1111111...
		int leftMask = allOnes << (j + 1);  // 11100000…
		int rightMask == ((1 << i) - 1);  // 00001000 - 00000001 = 00000111

		int mask = leftMask | rightMask;
		int nCleared = n & mask;
		int mShifted = m << i;  // 0110 << 3 = 0110000
```

```
            return nCleared | mShifted;
        }
}
```

// **Binary To String** → represent real number (double) in binary form

```
public class Question {

        public static String printBinary(double num) {

                if (num >= 1 || num <= 0) {
                        return "ERROR";
                }

                StringBuilder binary = new StringBuilder();
                binary.append(".");
                while (num > 0) {
                        if (binary.length() >= 32) {
                                return "ERROR";
                        }
                        double r = num * 2;
                        if (r > 1) {
                                binary.append(1);
                                num = r - 1;
                        } else {
                                binary.append(0);
                                num = r;
                        }
                }
                return binary.toString();
        }
}
```

// **Flip Bit to Win** → create the longest sequence of 1s by flipping exactly 1 bit

```
public class Question {

        public static int flipBit(int a) {
                // If all 1s, this is the longest sequence
                if (~a == 0) {
                        return Integer.BYTES * 8;
                }
```

```
        int currentLength = 0;
        int previousLength = 0;
        int maxLength = 1;

        while (a != 0) {
            if ((a & 1) == 1) {
                // Current bit is a 1, from right to left
                currentLength++;
            } else if ((a & 1) == 0) {
                previousLength = (a & 2) == 0 ? 0 : currentLength;  // a & 2 is next
                currentLength = 0;
            }
            maxLength = Math.max(previousLength + currentLength + 1,
        maxLength);
            a >>>= 1;
        }
        return maxLength;
    }
}
```

// **Next Number** → given a positive integer print the next smallest number and the next largest number that have the same number of 1 bits in their binary representation

```
public class Question {

    public static int getNext(int n) {
        int c = n;
        int c0 = 0;
        int c1 = 0;

        while (((c & 1) == 0) && (c != 0)) {
            c0++;
            c >>= 1;  // number of 0s to the right of p
        }

        while ((c & 1) == 1) {
            c1++;
            c >>= 1;  // number of 1s to the right of p
        }

        if (c0 + c1 == 31 || c0 + c1 == 0) {
            return -1;
        }
```

```
            int p = c0 + c1;  // position of the rightmost non-trailing 0

            n |= (1 << p);  // flip rightmost non-trailing zero
            n &= ~((1 << p) - 1);  // clear all bits to the right of p
            n |= (1 << (c1 - 1)) - 1;  // insert (c1 -1) ones on the right

            return n;
        }

        public static int getPrev(int n) {
            int temp = n;
            int c0 = 0;  // c0 is the size of the block of zeros to the left of the trailing 1s
            int c1 = 0;  // c1 is the number of trailing 1s

            while ((temp & 1) == 1) {
                c1++;
                temp >>= 1;
            }

            if (temp == 0) {
                return -1;
            }

            while ((temp & 1) == 0 && (temp != 0)) {
                c0++;
                temp >>= 1;
            }

            int p = c0 + c1;
            n &= ((~0) << (p + 1));  // clear from bit p onwards
            int mask = (1 << (c1 + 1)) - 1;
            n |= mask << (c0 - 1);

            return n;
        }
}
```

// **Debugger** → (n & (n-1)) == 0

- if A & B == 0 → A and B never have a 1 bit in the same place
- n needs to be a power of two or 0 for the expression to be true

// **Conversion** → write a function to determine the number of bits you would need to flip to convert integer A to integer B

// How to figure out which bits in two numbers are different → using XOR

```
public class Question {

	public static int bitSwapRequired(int a, int b) {
		int count = 0;
		for (int c = a ^ b; c != 0; c = c >> 1) {
			count += c & 1;
		}
		return count;
	}

	// Optimized code
	public static int bitSwap(int a, int b) {
		int count = 0;
		for (int c = a ^ b; c != 0; c = c & (c - 1)) {
			count++;
		}
		return count;
	}
}
```

// **Pairwise Swap** → swap odd and even bits in an integer

```
public class Question {

	public static int swapOddEvenBits(int x) {
		return ( ( (x & 0xaaaaaaaa) >>> 1 ) | (x & 0x55555555) << 1 );
	}
}
```

// **Draw Line**

```
public class Question {

  public static void drawLine(byte[] screen, int width, int x1, int x2, int y) {
    int startOffset = x1 % 8;
    int firstFullByte = x1 / 8;

    if (startOffset != 0) {
```

```
      firstFullByte++;
    }

    int endOffset = x2 % 8;
    int lastFullByte = x2 / 8;

    if (endOffset != 7) {
      lastFullByte--;
    }

// Set full bytes
    for (int b = firstFullByte; b <= lastFullByte; b++) {
      screen[(width / 8) * y + b] = (byte) 0xFF;
    }

// Create masks for start and end of line
    byte startMask = (byte) (0xFF >> startOffset);
    byte endMask = (byte) ~(0xFF >> (endOffset + 1));

    // Set start and end of line
    if ((x1 / 8) == (x2 / 8)) { // If x1 and x2 are in the same byte
      byte mask = (byte) (startMask & endMask);
      screen[(width / 8) * y + (x1 / 8)] |= mask;
    } else {
      if (startOffset != 0) {
        int byte_number = (width / 8) * y + firstFullByte - 1;
        screen[byte_number] |= startMask;
      }
      if (endOffset != 7) {
        int byte_number = (width / 8) * y + lastFullByte + 1;
        screen[byte_number] |= endMask;
      }
    }
  }
}
```