1530 Floating Point Numbers

Floating point numbers are represented at the hardware level of most computer systems in *normalized binary scientific notation*. We will consider a hypothetical computer in which floating point numbers are represented in a 16-bit word. An example is shown in Figure 1. In the common Intel chips, the exponent of a **float** has 8 bits and the mantissa has 23 bits, otherwise the representation is the same as in Figure 1.

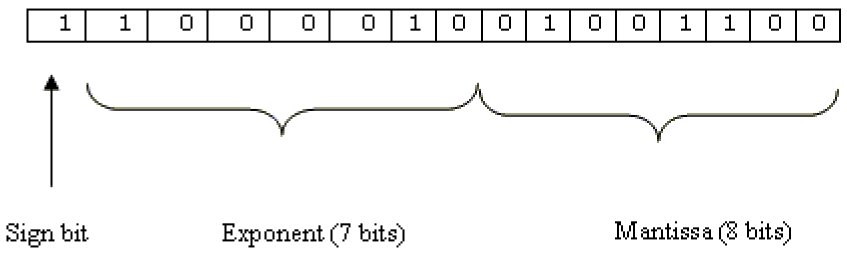

Figure 1

We will show that this particular example represents the number whose usual decimal representation is -10.375.

- The leftmost bit (the sign bit) is 1 if the floating point number is negative, 0 otherwise.
- The next seven bits (`1000010` in this example) represent the Exponent component of the normalized binary scientific notation. The following two operations will reveal the exponent's actual value:
  - Evaluate the binary number `1000010`: it is $1*64+0*32+0*16+0*8+0*4+1*2+0*1 = 66$.
  - Subtract the *bias correction* 63 from 66, which yields 3 as the actual value of the exponent that we will use below. (The purpose of the bias correction is to allow negative exponents. As long as the exponent has seven bits, which will always be the case in this program, the bias correction is $63 = 2^{7-1} - 1$.)
- The remaining 8 bits (`01001100` in this example) constitute the mantissa, which is the fractional part of the normalized binary scientific notation. You will see in the next step how it is used.
- The sign bit, the mantissa, and the exponent are used to calculate the value of the floating point number:

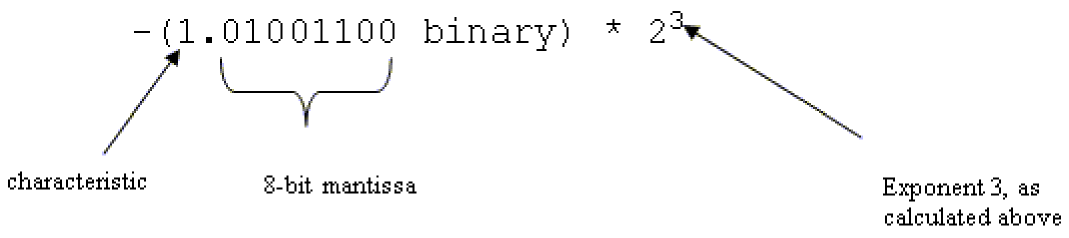

Figure 2

  - The period preceding the mantissa is the “binary point” which has the same meaning as the decimal point in a decimal (base ten) representation.
  - In the normalized binary scientific notation there is only one binary digit preceding the binary point, and it is *always* 1 (except if the floating point number is the number zero). It is called the characteristic. Since it is always 1, it does not explicitly appear in the 16-bit representation of Figure 1.

- Since the number `1.01001100` is in binary, the digits to the right of the binary point represent successive negative powers of 2. Therefore, the number shown in Figure 2 can be re-written as

$$-(1 + 2^{-2} + 2^{-5} + 2^{-6}) * 2^3 = -10.375 = -1.0375 * 10^1$$

  - Here the result has been expressed using scientific notation (base ten). Most programming languages would output this result as ‘`-1.0375e+001`’. The use of a lower case ‘`e`’ or upper case ‘`E`’, and the number of leading zeros in the exponent can vary from one programming language to another.

- An important exception: if all bits except the sign bit are zero, then the floating point number is zero, regardless of the sign bit.

Input

The input file contains an undetermined number of lines. Each line contains, starting in column 1, exactly 16 printable characters which can be any permutation of ‘`0`’s and ‘`1`’s. The first character represents the sign bit, the next seven characters, the exponent, and the last eight characters, the mantissa of a floating point number in normalized binary scientific notation according to the conventions discussed on the preceding page.

Output

Each floating point number read from the input will be written by your program to the output file, one number per line.

The following formatting conventions are required in your output:

- The floating point numbers will be displayed in scientific notation (base ten).
- Column 1 will either be blank (indicating a positive number or zero) or contain a minus sign (indicating a negative number).
- Column 2 will contain a digit.
- Column 3 will contain the decimal point.
- Columns 4 9 will contain six additional significant digits of the floating point number.
- The remaining columns will specify the exponent of scientific notation. The sign and magnitude of the exponent must of course be correct, but the exact format in which the exponent is displayed will be determined by the programming language you use.

Sample Input

```
1100001001001100
0011111100000000
1011111110000000
0000000010101010
0011011111100000
1001111011100000
0101011001010101
0100011011101101
0111111111111111
1100001000101100
0000000000000000
1000000000000000
```

Sample Output

```
-1.037500e+001
 1.000000e+000
-1.500000e+000
 1.804180e-019
 7.324219e-003
-2.182787e-010
 1.117389e+007
 2.465000e+002
 3.682143e+019
-9.375000e+000
 0.000000e+000
 0.000000e+000
```