Floating Point Thought Questions

For IEEE single precision floating point representation:

Why is there a bias?

There is a bias in order to:

(1) represent about as many positive exponents as negative exponents
(2) while still allowing for "easy" comparisons of floating point numbers

In single precision floating point, there are 8 bits for the exponent field and a bias of 127.

With single precision, we can represent exponents from $11111110_2 - 127_{10}$ to $00000001_2 - 127_{10}$ (127 to -126). And at the same time, the largest exponent "looks" like the largest exponent: 11111110_2 is the largest exponent, 11111101_2 is the next largest, … , and 00000001_2 is the smallest exponent. So comparison of floating point numbers uses the same algorithm as integers: compare the most significant bits, if those are equal, compare the next most significant bits, etc.

Why is there an implied 1?

If you did not have the implied 1, many numbers would have more than one bit pattern representation (another related issue: comparison becomes harder when a number has multiple bit patterns). For example, without an implied 1, 2.5_{10} would have the following representations: 0.101×2^2 or 0.0101×2^3 or 0.00101×2^4 or …

How many numbers can you represent?

2^{32} – <number of bit patterns which represent NaN or infinity> – <number of extra zeros>
= 2^{32} – <number of bit patterns with all 1's in the Exp> – <number of extra zeros>
= $2^{32} - 2^{24} - 1$

What is the smallest/largest positive numbers you can represent?

Largest Positive Number: 0 11111110 11111111111111111111111 => $1.11111111111111111111111 \times 2^{254-127}$
= $1.11111111111111111111111 \times 2^{127}$
Smallest Positive Number (denorm): 0 00000000 00000000000000000000001 => $0.00000000000000000000001 \times 2^{1-127}$
= $0.00000000000000000000001 \times 2^{-126}$

How do compare two floating point numbers?

Compare the most significant bits.
If those are equal, compare the next most significant bits.
If those are equal, compare the next most significant bits.
Etc…

Just as integers are compared.

How do you add/subtract two floating point numbers?

Align the binary point (in other words, shift the binary point so that the exponents are the same), and add/subtract the significands as you did in 3rd grade. Then, if necessarily, normalize and round the result.

$1.1011 \times 2^3 + 1.0101 \times 2^2$
= $1.1011 \times 2^3 + 0.10101 \times 2^3$
= 10.01011×2^3
= 1.001011×2^4

There is no need to round because we have 23 bits of significand.

True or False: For every 32-bit integer, there is a floating point number that exactly equals that integer (and vice versa).

For both cases: false. The 32-bit integer 0xFFFFFFFF has no exact floating point representation. The floating point 0.1 has no integer representation.

What is the largest integer you can cast into a floating point number and back again, and still get the same value? (integer → floating point → integer)

Assuming that the question was asking for the largest signed 32-bit integer:

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000\ 0000_2$

This integer can be casted to the following floating point. $1.11111111111111111111111 \times 2^{30}$.