1.5 is the Midpoint Between 0 and Infinity

Peter Zhu

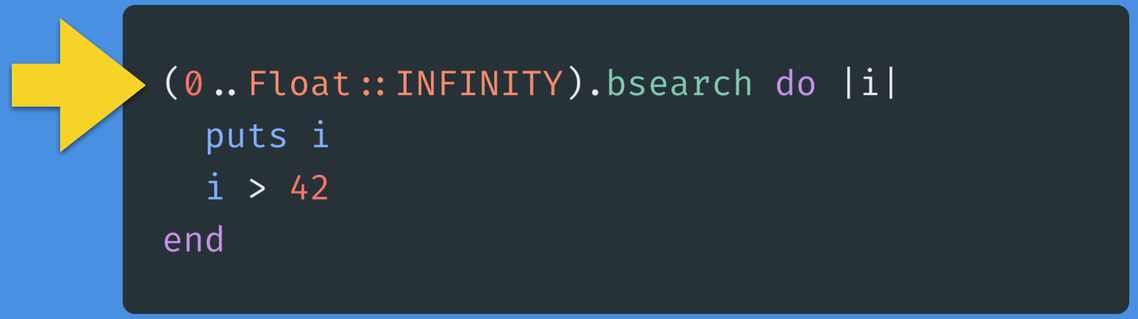
Ruby Core Committer
Senior Developer, Shopify

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?

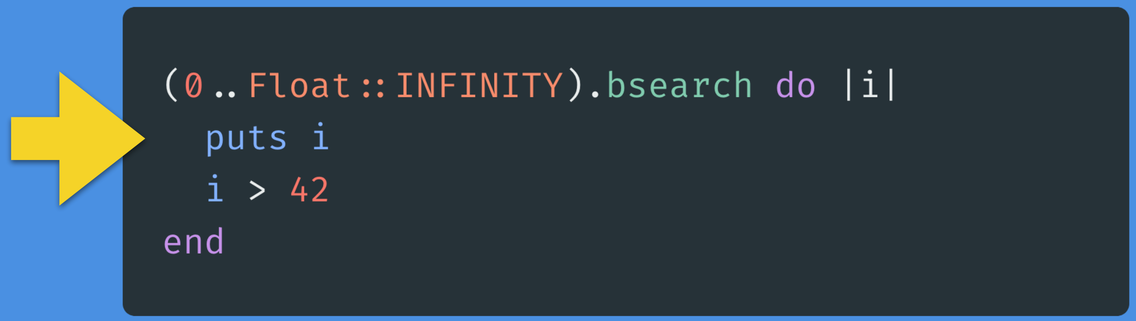
```
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?



```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?



```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?

```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

Why is 1.5 the first number Ruby's binary search inspects when searching between 0 and infinity?

```
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

```
1.5
1.6759759912428246e+154
1.5921412270130977e+77
4.8915590244884904e+38
2.7093655358260904e+19
6375342080.0
97792.0
383.0
23.96875
95.8125
47.921875
31.96484375
39.92578125
43.923828125
41.9248046875
42.92431640625
42.424560546875
42.1746826171875
42.04974365234375
...
```

Linear Search

Linear Search

Searching: 15

[12, 59, 48, 57, 10, 56, 11, 15, 74, 70]

Linear Search

Searching: 15

[12, 59, 48, 57, 10, 56, 11, 15, 74, 70]

Linear Search

Searching: 15

[12, 59, 48, 57, 10, 56, 11, 15, 74, 70]

Binary Search

Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

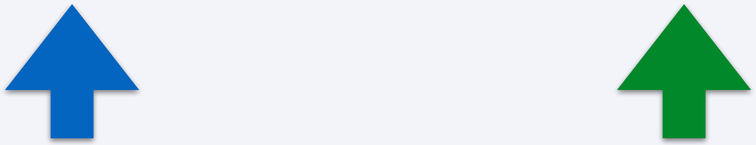
Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]



Binary Search

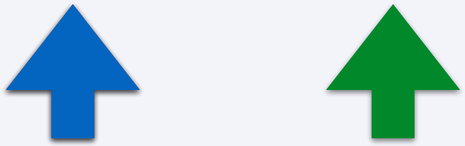
Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Binary Search

Searching: 15

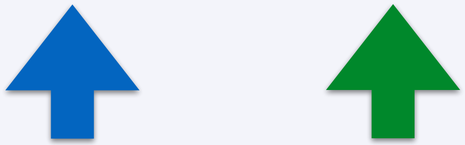
[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]



Binary Search

Searching: 15

[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]



Binary Search

Searching: 15

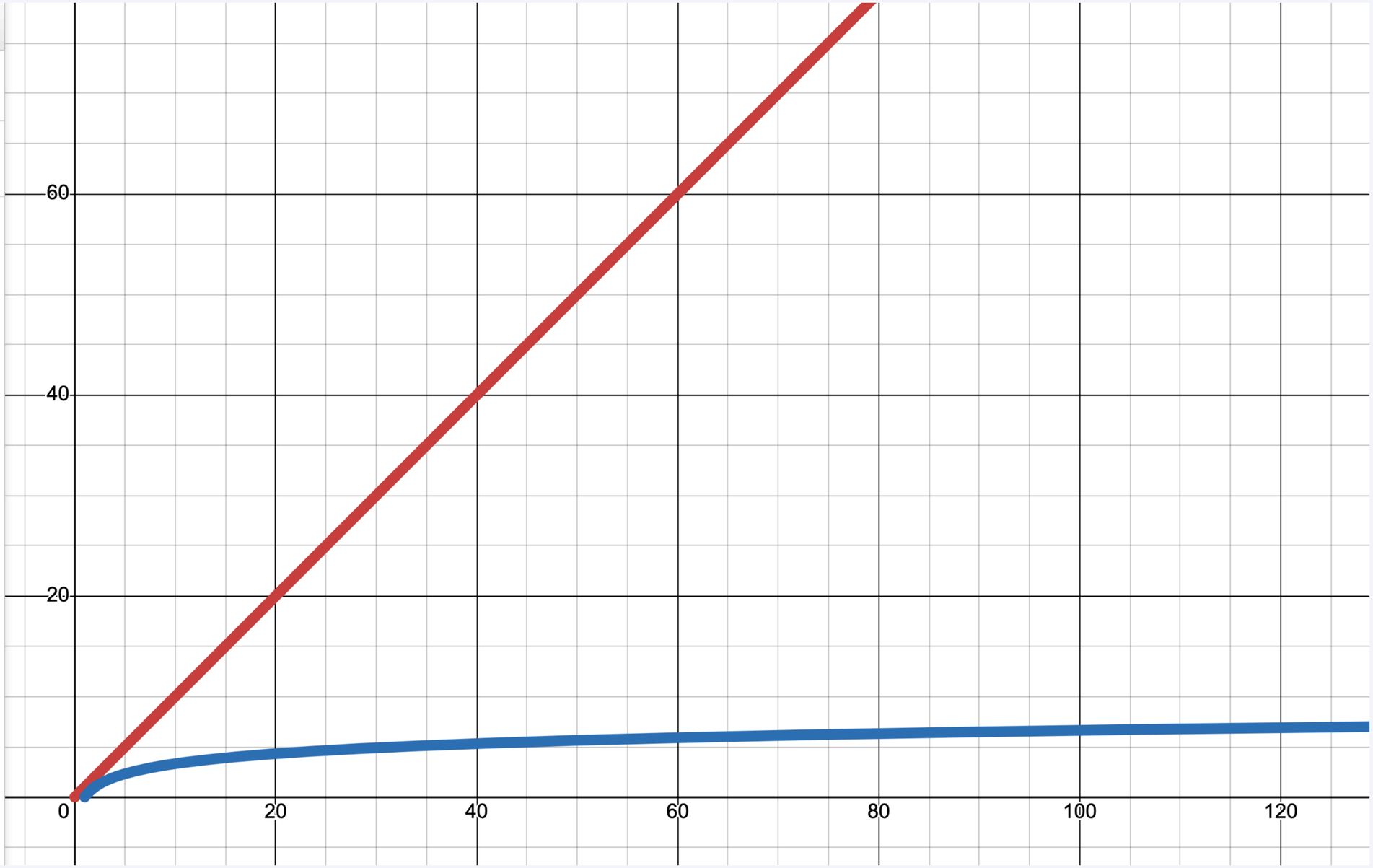
[10, 11, 12, 15, 48, 56, 57, 59, 70, 74]

Why use binary search over linear search?

1 $x\{x \geq 0\}$

2 $\log_2(x)\{x \geq 1\}$

3



How can you binary search over a float range?

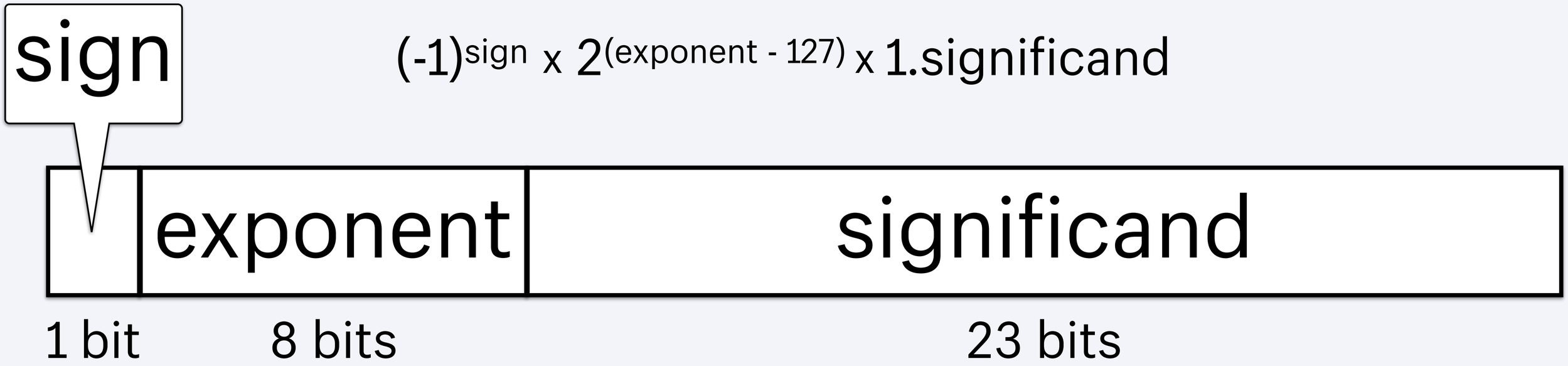
How floating point numbers work


sign
exponent
significand
1 bit
8 bits
23 bits


sign
exponent
significand
1 bit
8 bits
23 bits


sign
exponent
significand
1 bit
8 bits
23 bits

$$(-1)^{\text{sign}} \times 2^{(\text{exponent} - 127)} \times 1.\text{significand}$$


sign
(-1)sign x 2(exponent - 127) x 1.significand
exponent
significand
1 bit
8 bits
23 bits

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

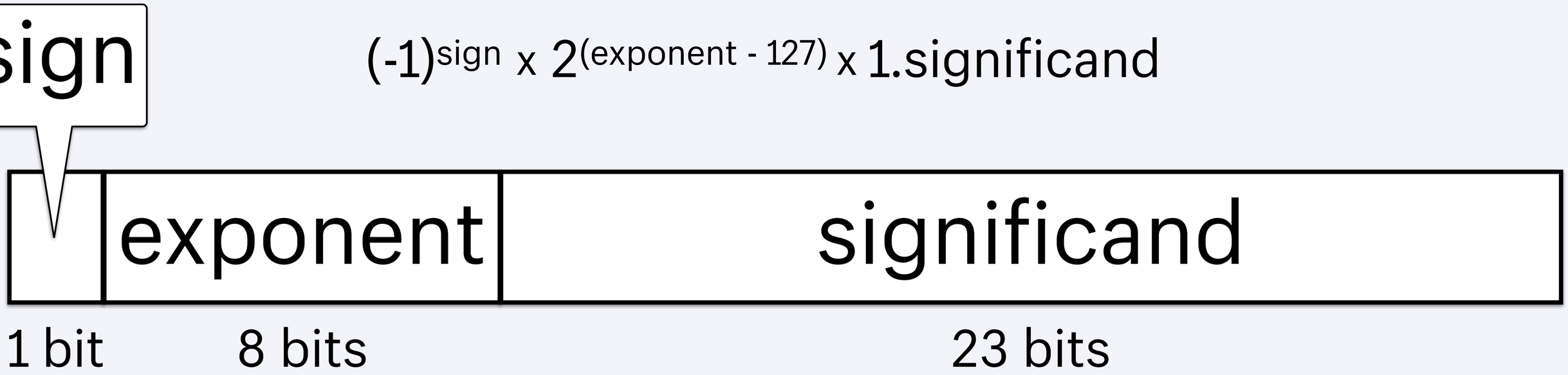
exponent	range
125	[0.25, 0.5)
126	[0.5, 1)
127	[1, 2)
128	[2, 4)
129	[4, 8)

sign

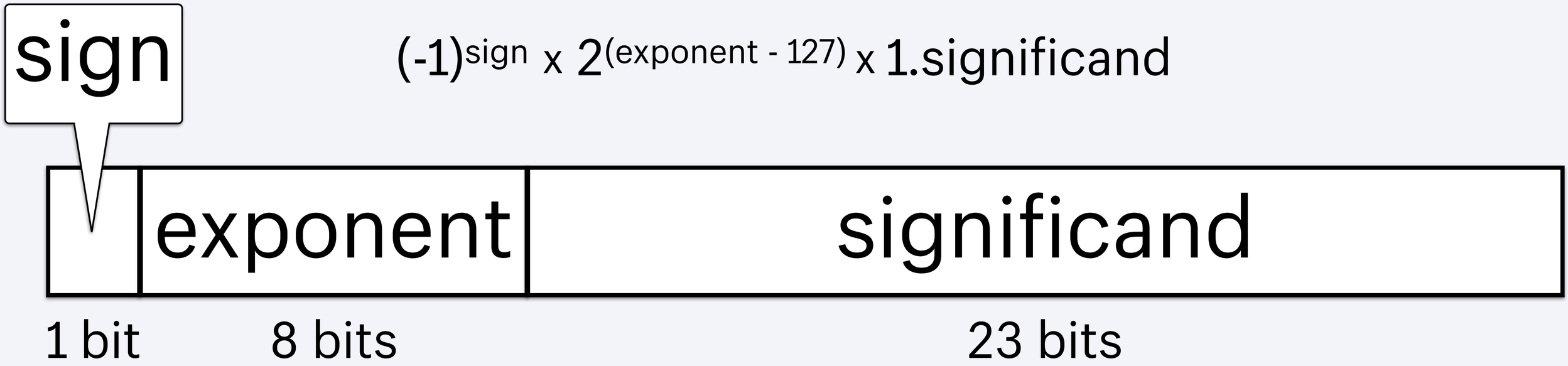
$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$



significand divides range to 2^{23} = 8,388,608 parts



significand divides range to 2^{23} = 8,388,608 parts

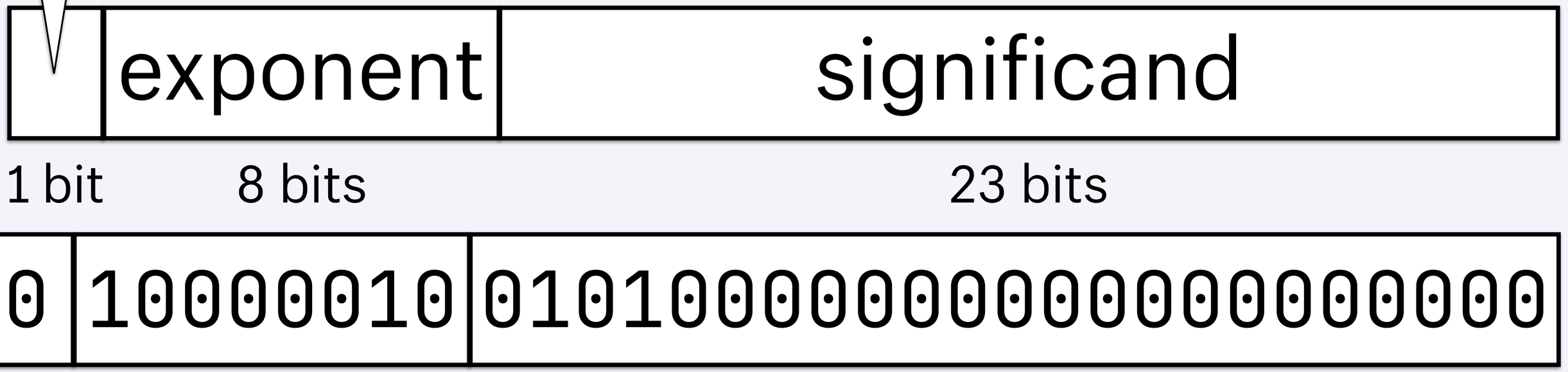
```
irb(main)> 0.1 + 0.2
⇒ 0.30000000000000004
irb(main)> 0.1 + 0.2 == 0.3
⇒ false
```

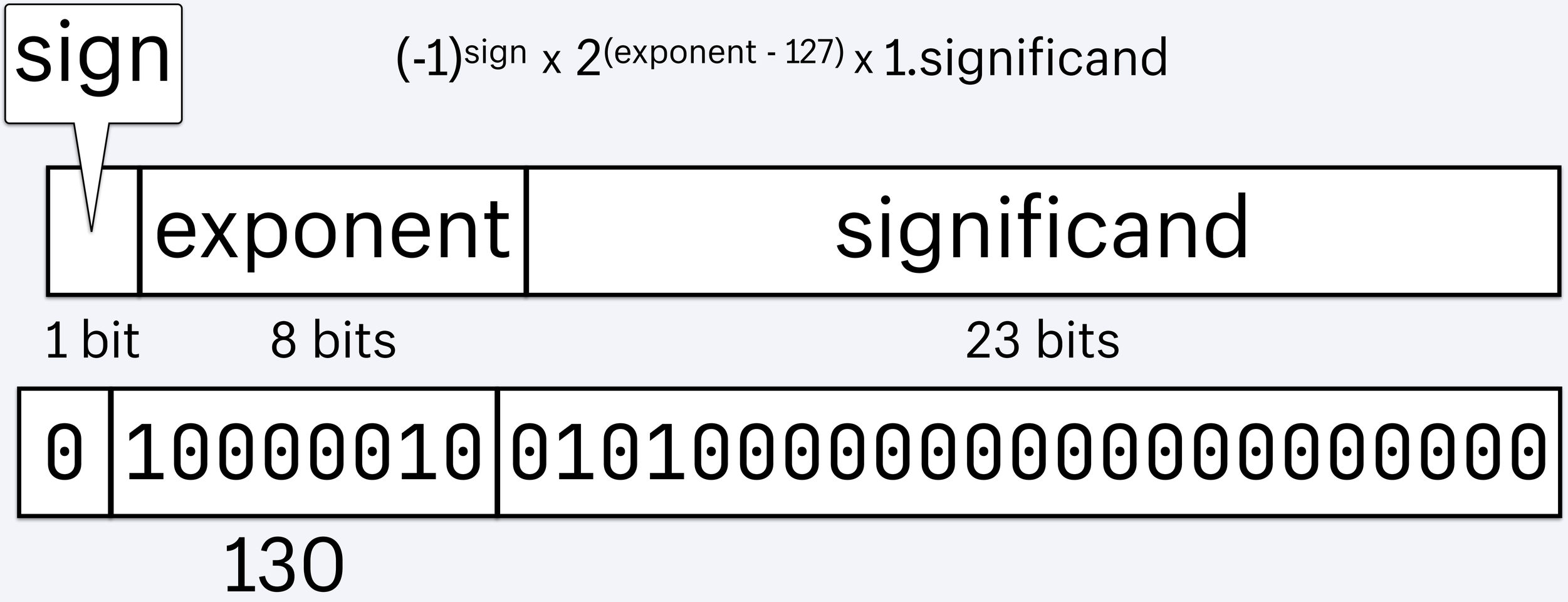
sign

$$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$$

sign

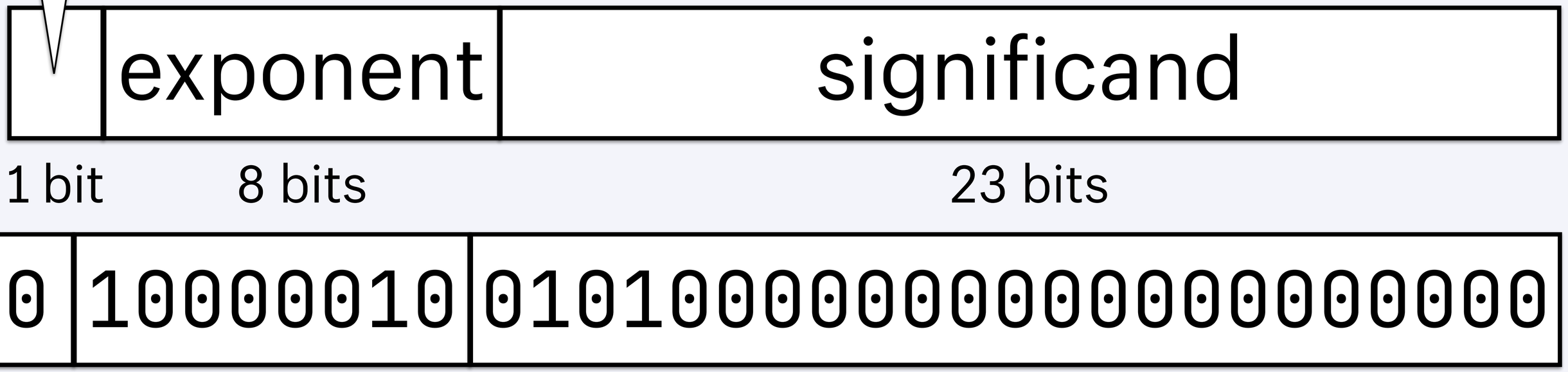
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$




sign
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$
exponent
significand
1 bit
8 bits
23 bits
0
10000010
01010000000000000000000
130

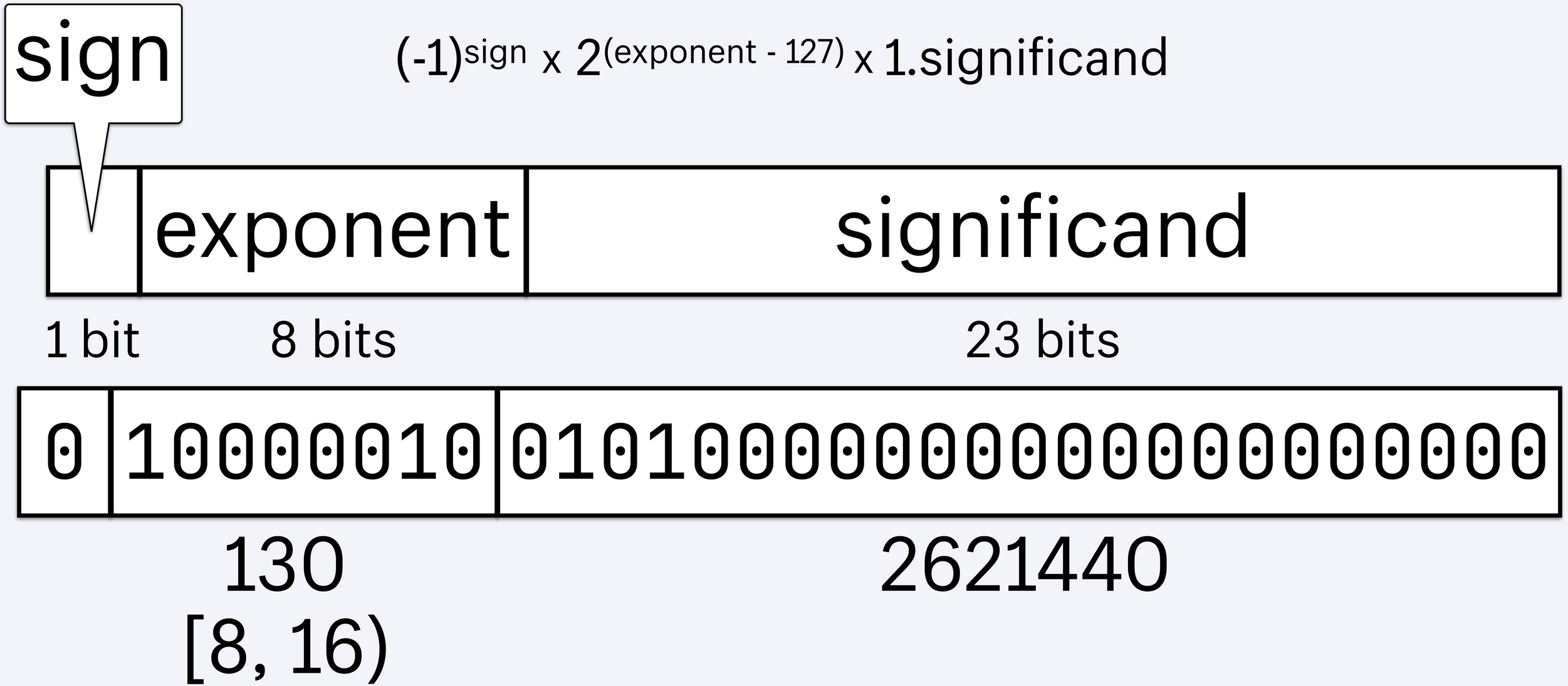
sign

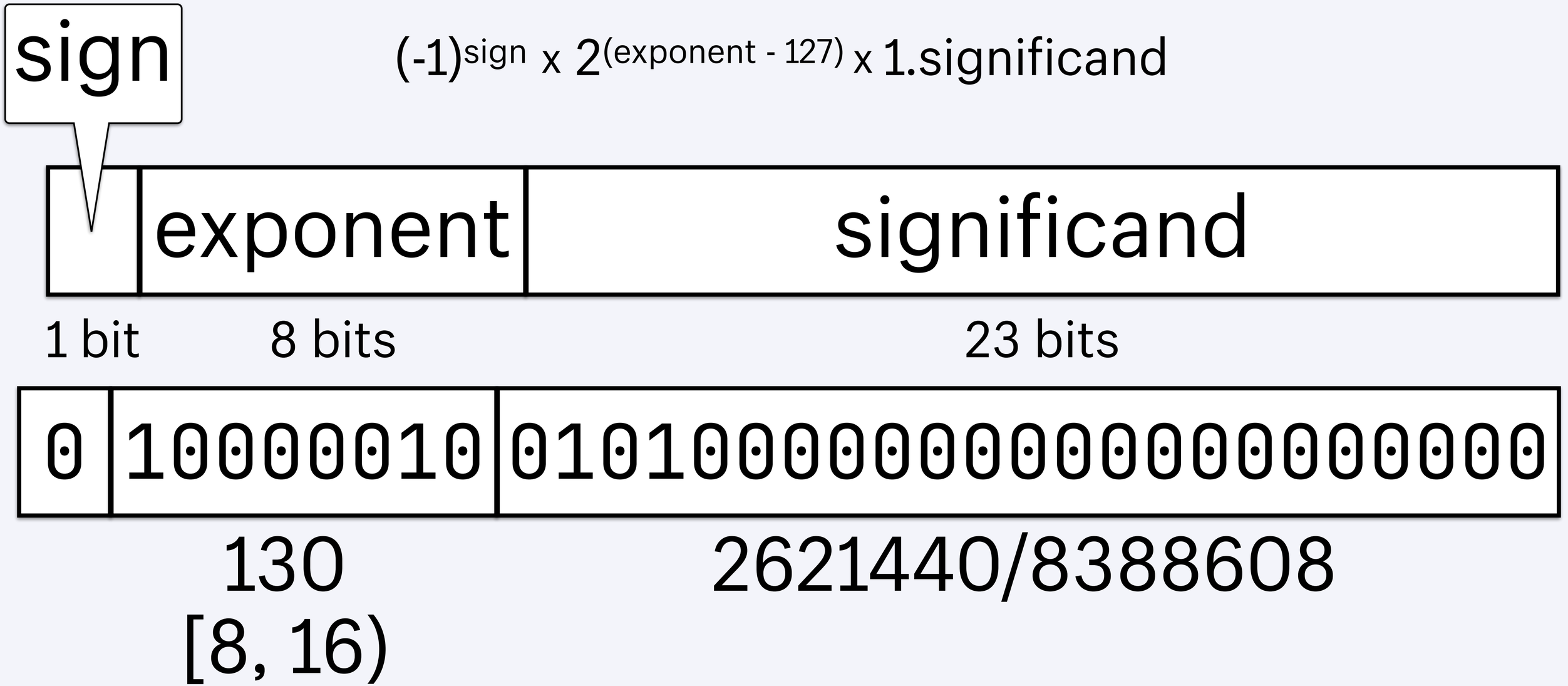
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$

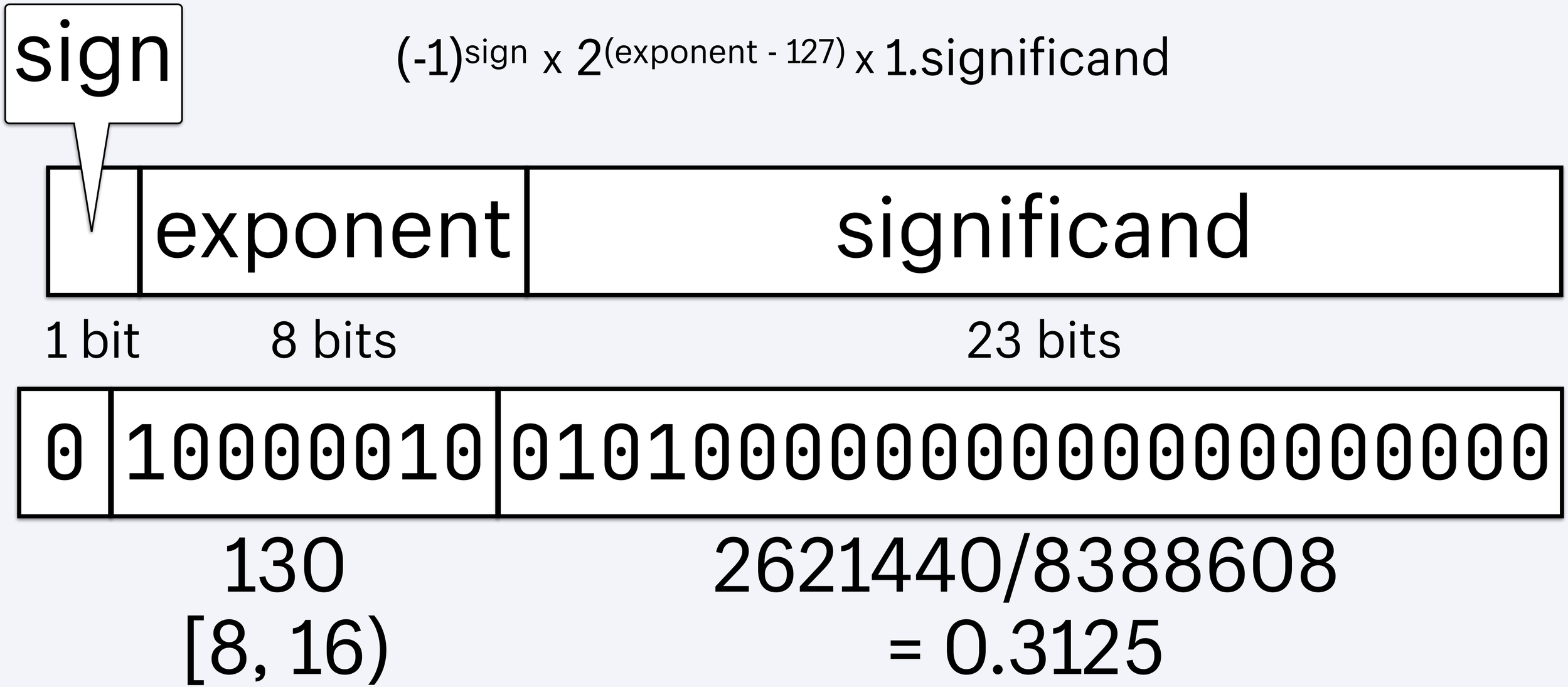


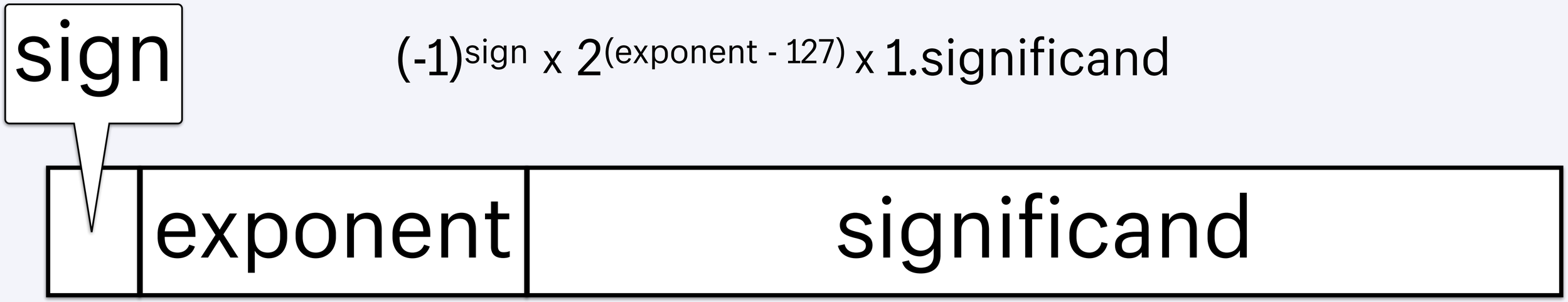
130

[8, 16)


sign
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$
exponent
significand
1 bit
8 bits
23 bits
0
10000010
01010000000000000000000
130
[8, 16)
2621440


sign
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$
exponent
significand
1 bit
8 bits
23 bits
0
10000010
01010000000000000000000
130
[8, 16)
2621440/8388608


sign
$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$
exponent
significand
1 bit
8 bits
23 bits
0
10000010
01010000000000000000000
130
[8, 16)
2621440/8388608
= 0.3125



$(-1)^{sign} \times 2^{(exponent - 127)} \times 1.significand$

0	10000010	01010000000000000000000

130

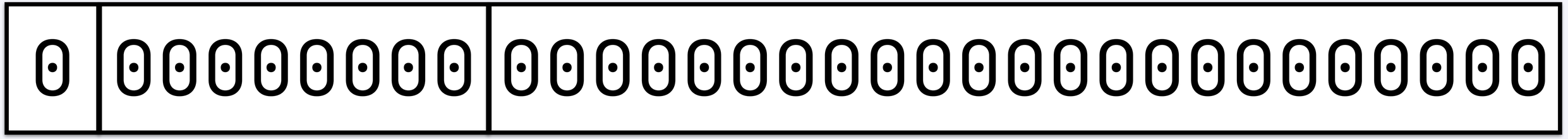
[8, 16)

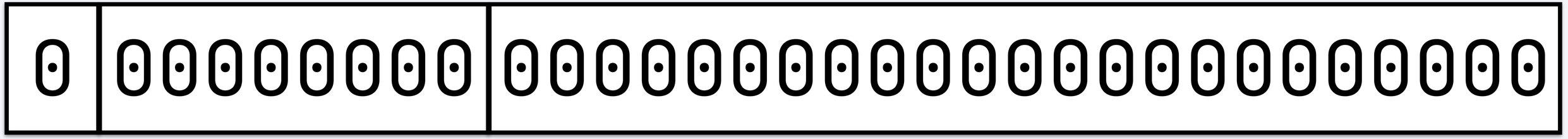
2621440/8388608

= 0.3125

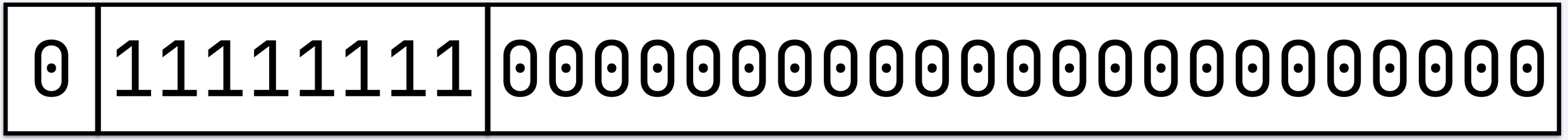
$8 + 8 \times 0.3125 = 10.5$

How does binary search and floating point numbers relate?

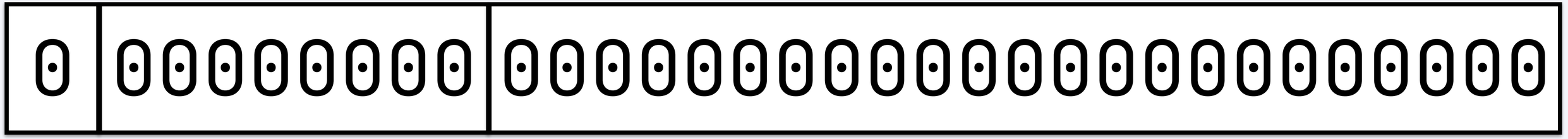

0
00000000
00000000000000000000000
0


0 00000000 00000000000000000000000

0

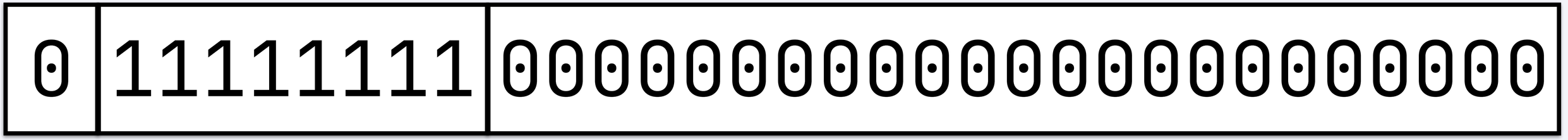

0 11111111 00000000000000000000000

infinity

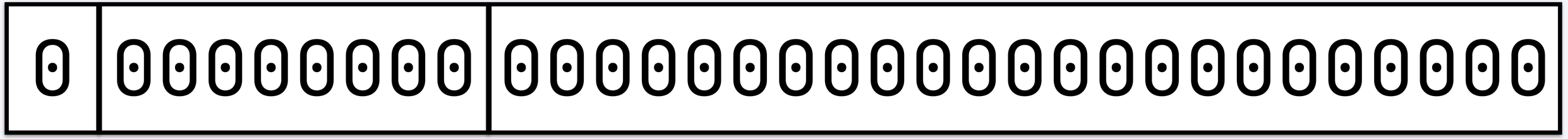

0
00000000
00000000000000000000000

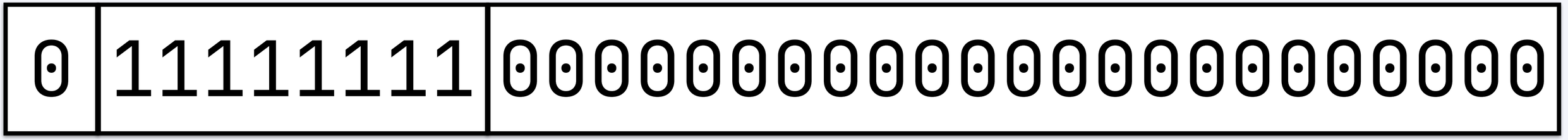
0

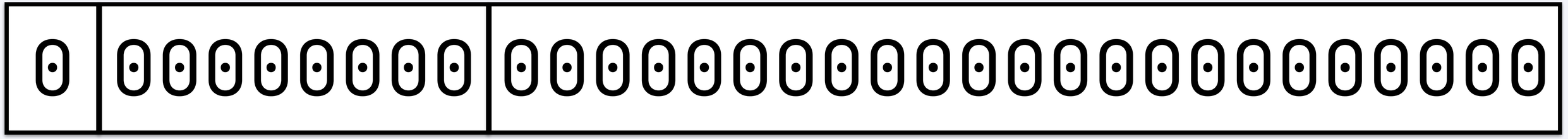
0

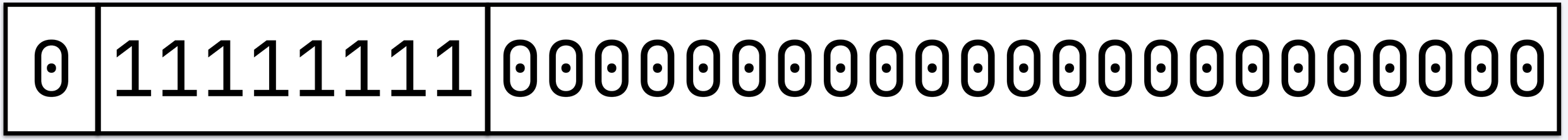

0
11111111
00000000000000000000000

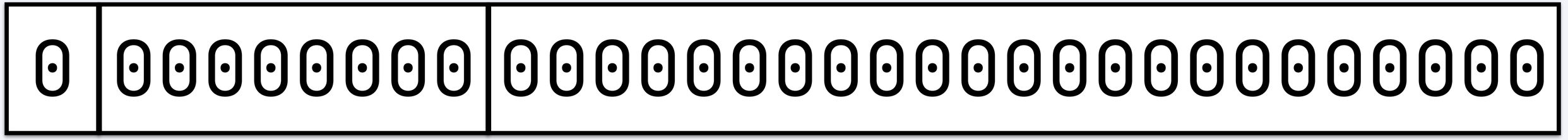
infinity

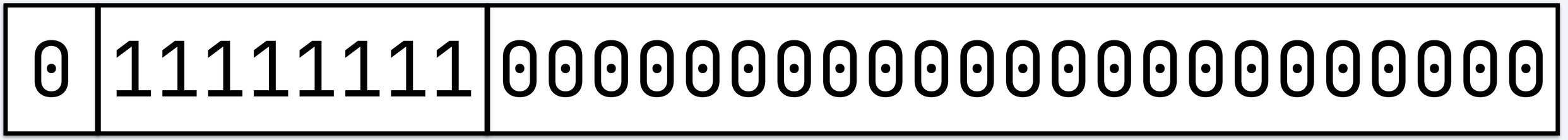

0
00000000
00000000000000000000000
0
0

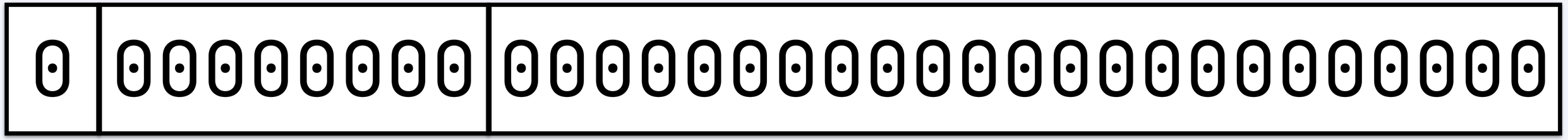

0
11111111
00000000000000000000000
infinity
2139095040


0
00000000
00000000000000000000000
0
0


0
11111111
00000000000000000000000
infinity
2139095040

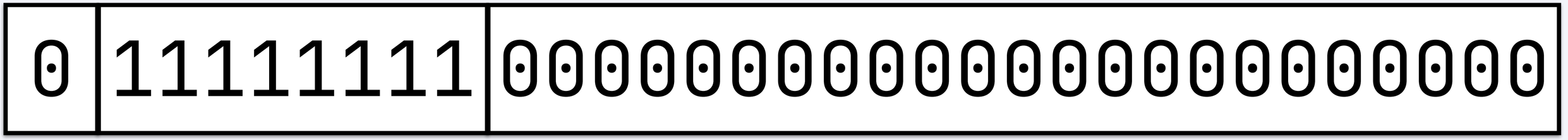

0 00000000 00000000000000000000000
0
0


0 11111111 00000000000000000000000
infinity
2139095040



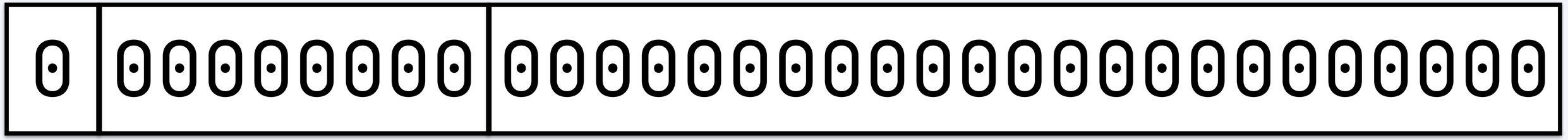
0

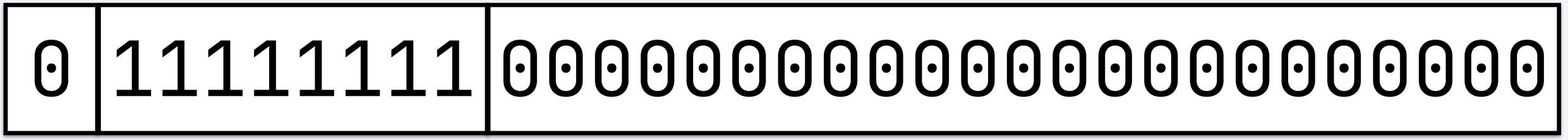
0



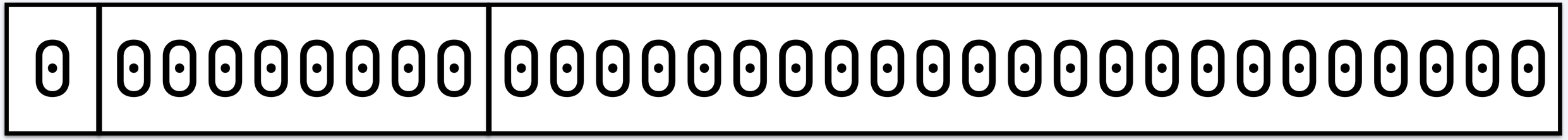
infinity

2139095040


0 00000000 00000000000000000000000
0
0

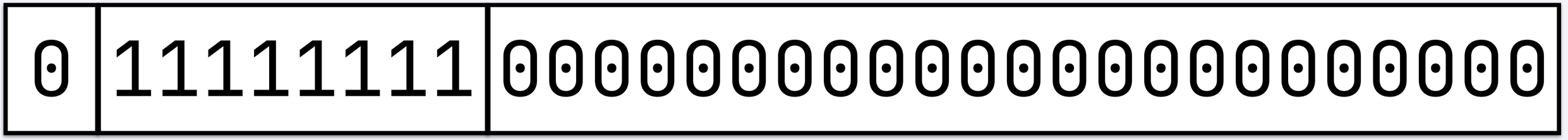

0 11111111 00000000000000000000000
infinity
2139095040

1069547520



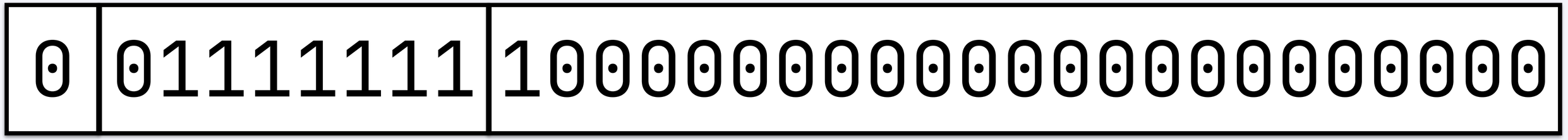
0

0

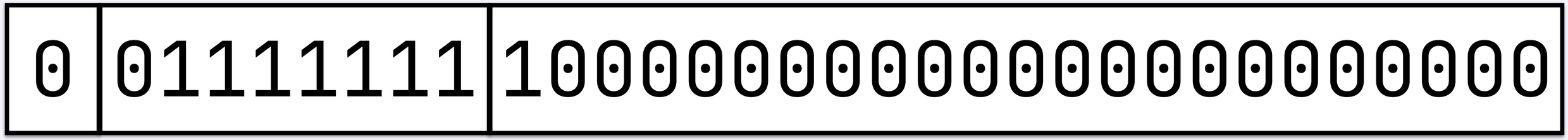


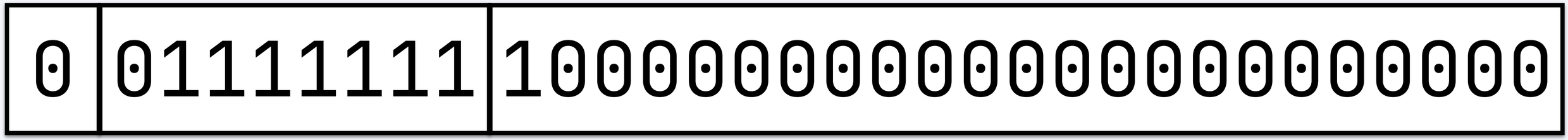
infinity

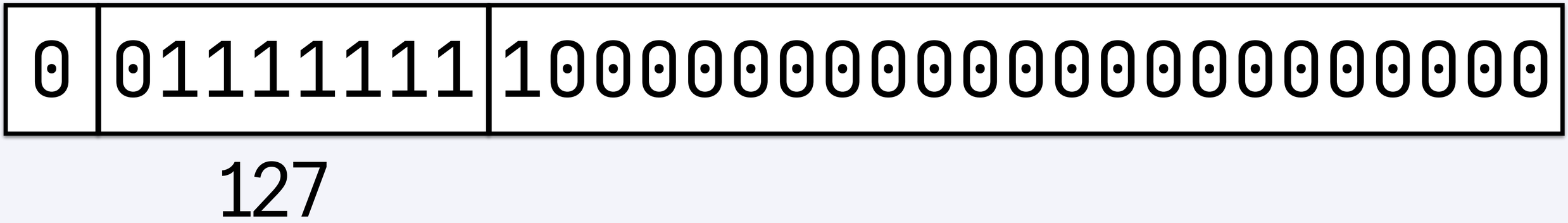
2139095040

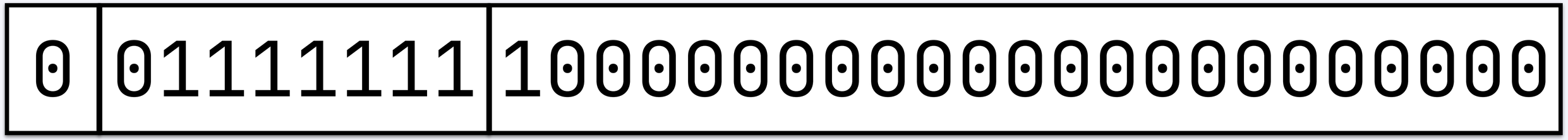


1069547520


0
01111111
10000000000000000000000

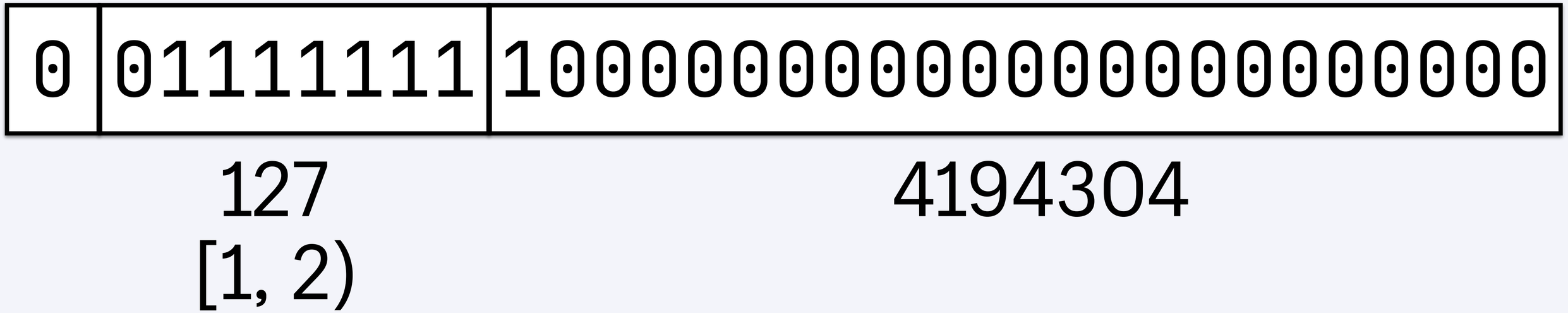

0
01111111
10000000000000000000000

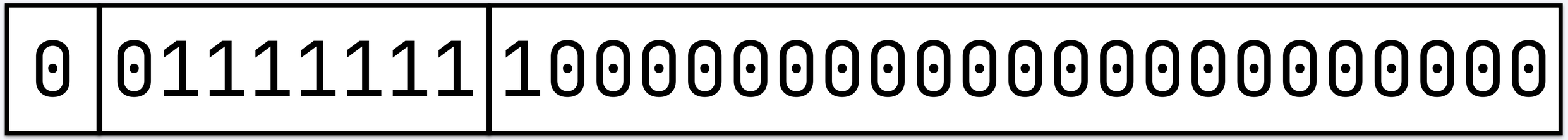

0
01111111
10000000000000000000000
127



127

[1, 2)

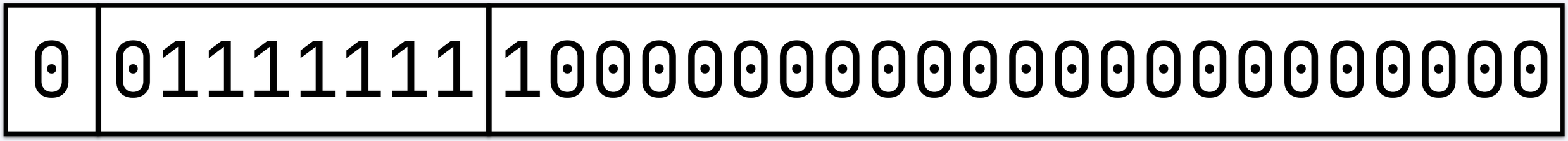

0
01111111
10000000000000000000000
127
[1, 2)
4194304



127

[1, 2)

4194304/8388608

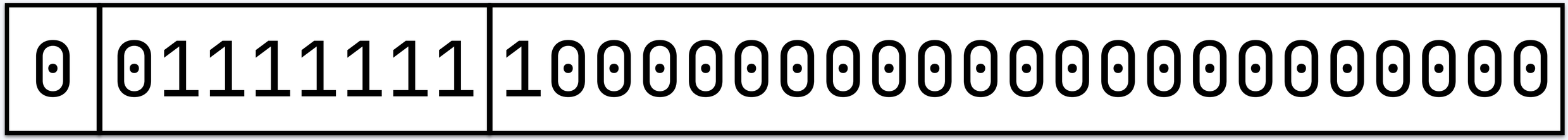


127

[1, 2)

4194304/8388608

= 0.5



127
[1, 2)

4194304/8388608
= 0.5

1 + 1 x 0.5 = 1.5

Recap

Recap

1. Read range endpoints (0 and infinity) as integers (0 and 2139095040)

Recap

1. Read range endpoints (0 and infinity) as integers (0 and 2139095040)
2. Found integer midpoint (1069547520)

Recap

1. Read range endpoints (0 and infinity) as integers (0 and 2139095040)
2. Found integer midpoint (1069547520)
3. Read integer midpoint back as a float (1.5)

```ruby
(0..Float::INFINITY).bsearch do |i|
  puts i
  i > 42
end
```

```
1.5
1.6759759912428246e+154
1.5921412270130977e+77
4.8915590244884904e+38
2.7093655358260904e+19
6375342080.0
97792.0
383.0
23.96875
95.8125
47.921875
31.96484375
39.92578125
43.923828125
41.9248046875
42.92431640625
42.424560546875
42.1746826171875
42.04974365234375
...
```

Why does this even work?

Binary search requires sorted lists


sign
exponent
significand
1 bit
8 bits
23 bits


sign
exponent
significand
1 bit
8 bits
23 bits


sign
exponent
significand
1 bit
8 bits
23 bits

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /*  ...  */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /*  ...  */
}
```

```c
union int64_double {
    int64_t i;
    double d;
};

static int64_t
double_as_int64(double d)
{
    union int64_double convert;
    convert.d = fabs(d);
    return d < 0 ? -convert.i : convert.i;
}

static VALUE
range_bsearch(VALUE range)
{
    VALUE beg = RANGE_BEG(range);
    VALUE end = RANGE_END(range);

    /* ... */

    if (RB_FLOAT_TYPE_P(beg) || RB_FLOAT_TYPE_P(end)) {
        int64_t low  = double_as_int64(RFLOAT_VALUE(rb_Float(beg)));
        int64_t high = double_as_int64(RFLOAT_VALUE(rb_Float(end)));
        int64_t mid, org_high;
        BSEARCH(int64_as_double_to_num);
    }

    /* ... */
}
```

float.exposed

half | bfloat | **float** | double

Value

1.5

Bit Pattern

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Sign

0

Exponent

127

Significand

4194304

Raw Hexadecimal Integer Value

0x3fc00000

Raw Decimal Integer Value

1069547520

Hexadecimal Form ("%a")

0x1.8p+0

Position within Significand–Exponent Range

float.exposed

half | bfloat | **float** | double

Value

1.5

Bit Pattern

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Sign

0

Exponent

127

Significand

4194304

Raw Hexadecimal Integer Value

0x3fc00000

Raw Decimal Integer Value

1069547520

Hexadecimal Form ("%a")

0x1.8p+0

Position within Significand–Exponent Range

float.exposed

half | bfloat | **float** | double

Value

1.5

Bit Pattern

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Sign

0

Exponent

127

Significand

4194304

Raw Hexadecimal Integer Value

0x3fc00000

Raw Decimal Integer Value

1069547520

Hexadecimal Form ("%a")

0x1.8p+0

Position within Significand–Exponent Range

float.exposed

half | bfloat | **float** | double

Value

1.5

Bit Pattern

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Sign

0

Exponent

127

Significand

4194304

Raw Hexadecimal Integer Value

0x3fc00000

Raw Decimal Integer Value

1069547520

Hexadecimal Form ("%a")

0x1.8p+0

Position within Significand–Exponent Range

float.exposed

half | bfloat | **float** | double

Value

1.5

Bit Pattern

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Sign

0

Raw Hexadecimal Integer Value

0x3fc00000

Exponent

127

Raw Decimal Integer Value

1069547520

Significand

4194304

Hexadecimal Form ("%a")

0x1.8p+0

Position within Significand–Exponent Range

Thank You

blog.peterzhu.ca/ruby-range-bsearch/

@peterzhu2118

peter@peterzhu.ca