Representing Numbers as Bit Patterns

The goal is to describe how numbers are represented as **bit patterns** in computers.

Information and Bits

- **Information** = choice between different possibilities.
- Simplest situation: choice between two possibilities, 0 and 1. This choice is called a **bit**.
- Relevant for computers because two-part choices are easiest to represent physically (1 = current, 0 = no current).
- Larger collection of information: use multiple bits, e.g., $01101011000110010101101 1\ldots$.
- 8 bits (= 1 byte): choice between $2^8 = 256$ possibilities.

Bit Patterns

Bit patterns must be interpreted to have meaning. $01101011 = ?$

A system is needed to specify the meaning of different bit patterns. Such systems exist for:

- Numbers (integers, floating-point numbers)
- Letters
- Pixels (image files)
- Amplitude (audio files)
- Computer instructions (program)

Number Systems

Decimal System

```
$4532 = 4 \cdot 1000 + 5 \cdot 100 + 3 \cdot 10 + 2 \cdot 1 = 4 \cdot 10^3 + 5
```

- Base: 10
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (because $10 \cdot 10^i = 10^{i+1}$)

Septal System

```
$4532_7 = 4 \cdot 7^3 + 5 \cdot 7^2 + 3 \cdot 7^1 + 2 \cdot 7^0 = 4 \cdot 343
```

- Base: 7
- Digits: 0, 1, 2, 3, 4, 5, 6 (because $7 \cdot 7^i = 7^{i+1}$)

Binary System

```
$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 8 +
```

- Base: 2
- Digits: 0, 1 (because $2 \cdot 2^i = 2^{i+1}$)
- Relevant for computers because two-part choices are easiest to represent physically (1 = current, 0 = no current).
- The binary system is also called the **binary number system**.
- It gives a natural interpretation of bit patterns as non-negative integers.

Hexadecimal System

Also used in computer science is the 16-number system:

```
$4A3F_{16} = 4 \cdot 16^3 + 10 \cdot 16^2 + 3 \cdot 16^1 + 15 \cdot 16^0 = 4 \
```

- Base: 16
- Digits: 0, 1, 2, 3, ..., 9, A (=10), B (=11), ..., F (=15) (because $16 \cdot 16^i = 16^{i+1}$)

Hexadecimal Notation

The 16-number system can also be used as a simple/short way to describe **bit strings**. Group bits into groups of 4 (i.e. 16 different possibilities):

```
0110 1010 1110 01...
```

Use the 16 digits to describe these possibilities:

0111	7	0110	6
0101	5	0100	4
0011	3	0010	2
0001	1	0000	0
1111	F	1110	E
1101	D	1100	C
1011	B	1010	A
1001	9	1000	8

```
0110 1010 1110 01... = 6AE...
```

Addition

Addition works the same in all number systems, just with the base exchanged.

Decimal System

```
  1 1 1 1
 5432 +
96781 =
102213
```

Binary System

```
 1 1 1 1
 11102 +
111002 =
1010102
```

Subtraction, multiplication, division also work the same. E.g.

```
1010_2 \cdot 1110_2 = 10001100_2 \text{ (Check: } 10 \cdot 14 = 140)
```

```
1101011_2 : 101_2 = 10101_2, \text{ remainder } 10_2 \text{ (Check: } 107 : 5
```

Converting Between Number Systems

From Other Bases

Use the definition of number systems.

```
$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 8 +
$4532_7 = 4 \cdot 7^3 + 5 \cdot 7^2 + 3 \cdot 7^1 + 2 \cdot 7^0 = 4 \cdot 343
```

To Other Bases

Use repeated integer division. Remember how integer division works:

Integer division	Quotient	Rest	As equation
31: 7	4	3	$31 = 7 \cdot 4 + 3$
25: 2	12	1	$25 = 2 \cdot 12 + 1$

Conversion to Binary Number System

The following algorithm finds the digits from right to left in the binary representation of a positive integer N:

1. X = N
2. As long as X > 0, repeat:
    - Next digit = remainder by integer division X : 2
    - X = quotient by integer division X : 2

Example: N = 25

Integer division	Quotient	Rest
25: 2	12	1
12: 2	6	0
6: 2	3	0
3: 2	1	1
1: 2	0	1

Why Does it Work?

```
$25 = 2 \cdot 12 + 1 = 2(2 \cdot 6 + 0) + 1 = 2(2(2 \cdot 3 + 0) + 0) + 1 = 2
```

Note that the last division is always 1: 2 (with quotient 0 and remainder 1). Because X becomes 1 at some point, as an integer division by 2 constantly makes X smaller, but cannot get from integer ≥ 2 to integer ≤ 0.

Representation of Integers

Number representations (almost always) use a fixed number of bits (so operations can be implemented efficiently).

k bits = 2^k different bit patterns

Positive integers: the binary number system provides a natural representation.

0111	7	0110	6
0101	5	0100	4
0011	3	0010	2
0001	1	0000	0
1111	15	1110	14
1101	13	1100	12
1011	11	1010	10
1001	9	1000	8

How should these 2^k bit patterns be distributed if we want to represent all integers, both negative and positive?

Two's Complement

One possible representation of both negative and positive integers is as follows:

0111	7	0110	6
0101	5	0100	4
0011	3	0010	2
0001	1	0000	0
1111	-1	1110	-2
1101	-3	1100	-4
1011	-5	1010	-6
1001	-7	1000	-8

This is called **two's complement** (for reasons that are not relevant here). It can also be described as the highest digit counting $-(2^{k-1})$ instead of 2^{k-1}:

```
$1101_2 = 1 \cdot (-(2^3)) + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot
```

Two's Complement Properties

The **two's complement** representation has many good properties:

1. The sign can be seen by the first bit.
2. A simple method to change the sign exists:
    - Copy bits from right to left, up to and including the first 1-bit. The rest of the bits are inverted (i.e. 1 is set to 0 and vice versa).
    - (Example: $6 = 0110 \rightarrow 1010 = -6$)
3. The common method of addition also works for negative numbers. No extra logic circuits for these (saves transistors on the CPU).
4. Subtraction can be done by reversing the sign and adding. No logic circuits for subtraction (saves transistors on the CPU).

Two's complement is therefore often chosen as the representation for integers. In Java, the type `int` is an integer in two's complement (k = 32). In Python, this is also the basic type for integers.

Representations of Decimal Numbers

Number representations (almost always) use a fixed number of bits.

k bits = 2^k different bit patterns

How to use k bits to describe decimal numbers?

From the decimal system are known

- Fixed decimal point (45.32)
- Floating decimal point (−6.87 · 10−6)

These can easily be repeated in the total system (base 2). See next pages.

In computers, floating decimal point (with base 2) is most often used. To understand these, you must first understand fixed decimal point (with base 2).

In Java, the types `float` (k = 32) and `double` (k = 64) are floating-point decimal numbers. In Python, the type `float` is the same (k = 64).

Fixed Decimal Point

Decimal System:

```
$45.32 = 4 \cdot 10 + 5 \cdot 1 + 3 \cdot 1/10 + 2 \cdot 1/100 = 4 \cdot 10^1
```

The binary number system:

```
$10110.111_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdo
```

Floating Decimal Point

Decimal system: get the comma to stand after the first digit $! = 0$.

```
2340000.0 = 2.34 \cdot 10^6
0.000456 = 4.56 \cdot 10^{-4}
-0.0987 = -9.87 \cdot 10^{-2}
```

- Sign: plus
- Exponent: 6
- Mantissa: 2.34
- Sign: plus
- Exponent: −4
- Mantissa: 4.56
- Sign: minus
- Exponent: −2
- Mantissa: 9.87

Total system: get the comma to stand after the first digit ! $= 0$ (is always 1).

```
101100.0_2 = 1.011_2 \cdot 2^5
-0.01101_2 = -1.101_2 \cdot 2^{-2}
```

A fixed number of bits is set aside for each of: sign, exponent, mantissa. For k = 8 we choose: 1, 3 and 4 bits. The exponent can be positive or negative, we use two's complement for it. The mantissa is filled with 0's to the right if necessary. Example: for -0.01101_2

- Sign: 1 (1 for negative number, 0 for positive)
- Exponent: 110 (-2 in two's complement (3 bits))
- Mantissa bits: (1.)1010 (first bit is not written, as it is always 1)

Limitations

Integers and decimal numbers are infinite sets of numbers. If a fixed number of (k) bits is allocated, a finite number (2^k) of different bit patterns is obtained.

Not all numbers can be represented!

Evidenced e.g. by

- Overflow
  - `maxInt + maxInt = ?`
- Rounding errors
- Large number `x` + very small number `y` = same large number `x`
- `(x + y) + z != x + (y + z)` if e.g. `x + y` cannot be represented exactly.

In practice, problems are rarely experienced due to a large number of bits in the number representations.

Alternatively, there are programming libraries that implement e.g. arbitrarily large integers (using a variable number of bits, as well as a loss of efficiency). This happens automatically in Python for the type `int`.