Arbitrary Precision Arithmetic

Branden Xia

Jan 21, 2026

Why?

Builtin integers in most programming languages have fixed sizes (e.g. `short`, `int`, `long` in C and C++; `i32`, `i64` in Rust). This is since hardware has finite memory.

However, since there's only a finite number of possible combinations of bits in these fixed-size integers, they can only represent a limited range of values.

`0x00`	`0x00`	`0x00`	`0x43`

Table 1: An typical 32-bit (4 bytes) integer in memory (little endian)

What about in Python?

What about in Python?

Python use floating point numbers for all the numbers beyond the fixed size, but, again, since it has finite precision, it can only represent a limited number of digits accurately.

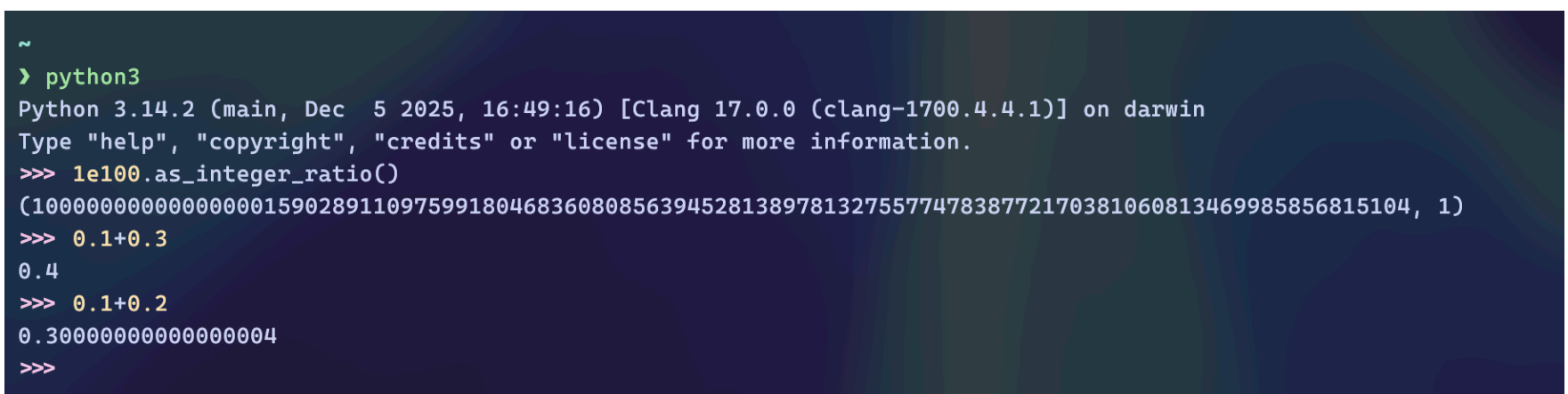

Figure 1: Demonstration of Python's number representation limits

Solution?

Solution: Dynamic-sized Numbers

The solution is intuitive: if precision of fixed-size numbers is limited, let's just increase its size dynamically if currently given memory storage is not enough.

0xFF	0xFF	0xFF	0xFF

↓

0x01	0x00	0x00	0x00	0x00

Table 2: Dynamically increase storage size after overflow

How to represent a number?

How to represent a number?

When we write a number like "2417", we normally assume it's under base 10, which means:

$$2417_{10} = 2 \cdot 10^3 + 4 \cdot 10^2 + 1 \cdot 10^1 + 7 \cdot 10^0$$

We can do the same thing in computer by storing each digit as a number. A dynamically-sized number would just be an array of integers.

Base?

Base 10 is what we normally work with, but is it efficient?

Base?

What will type of each element in the array be?

Consider the smallest unit of memory that we can directly manipulate in computer: a byte.

$$1 \text{ byte} = 8 \text{ bits}$$

$$2^8 = 16$$

$$\text{base } 10 \text{ waste} : 1 - \log_{16} 10 \approx 16.95\%$$

$$\text{base } 2^8 \text{ waste} : 1 - \log_{16} 16 \approx 0\%$$

Arithmetic

Addition

Addition works similarly as the vertical addition we do on paper. However, there's still some optimization we can do.

$$\begin{array}{r} 9\ 6\ 2 \\ +\quad 9\ 3 \\ \hline 1\ 0\ 5\ 5 \end{array}$$

Figure 2: Vertical addition [1]

Addition (Naive)

```
1 for i from 1 to max(length(a), length(b)) − 1 do
2     a[i] ← a[i] + b[i]
3     if a[i] ≥ base do
4         a[i] ← a[i] − base
5         a[i + 1] ← a[i + 1] + 1
```

Addition (Naive)

1 **for** i from 1 to $\max(\text{length}(a), \text{length}(b)) - 1$ **do**
2 $\quad a[i] \leftarrow a[i] + b[i]$
3 $\quad$ **if** $a[i] \geq \text{base}$ **do** $\quad \leftarrow$ branch here
4 $\quad\quad a[i] \leftarrow a[i] - \text{base}$
5 $\quad\quad a[i+1] \leftarrow a[i+1] + 1$

Addition (Branch-less)

1 $i \leftarrow 1$, carry $\leftarrow 0$
2 **while** $i \leq \text{length}(b)$ **do**
3 | // hardware builtin overflow in practice (e.g. x86 ADD with carry)
4 | $a[i] \leftarrow a[i] + b[i] + \text{carry}$
5 | carry $\leftarrow a[i] \div$ base (integer division)
6 **while** $i \leq \text{length}(a)$ **do** // same as above
7 | $a[i] \leftarrow a[i] + \text{carry}$
8 | carry $\leftarrow a[i] \div$ base (integer division)

Subtraction

```
318  FORCE_INLINE auto sub_usize_borrow(usize a, usize b, unsigned char borrow_in,
  1  |                                   usize *out) → unsigned char {
  2  #if defined(_MSC_VER) && defined(_M_X64)
  3  |  return _subborrow_u64(borrow_in, a, b, out);
  4  #elif defined(__x86_64__) && defined(__ADX__)
  5  |  return _subborrow_u64(borrow_in, a, b, (unsigned long long *)out);
  6  #else
  7  |  usize res = a - b - borrow_in;
  8  |  bool borrow_out = (borrow_in ? (a ≤ b) : (a < b));
  9  |  *out = res;
 10  |  return borrow_out ? 1 : 0;
 11  #endif
 12  }
```

Figure 3: A single-digit subtraction with borrow

```cpp
template <traits::bignum_concept T>
auto sub_bignum(T &a, const T &b) {
  auto a_size = a.size(), b_size = b.size();

  auto a_ptr = a.data();
  const auto b_ptr = b.data();
  unsigned char borrow = 0;
  std::size_t i = 0;

  for (; i < b_size; ++i)
    borrow = sub_usize_borrow(a: a_ptr[i], b: b_ptr[i], borrow_in: borrow, out: &a_ptr[i]);

  for (; i < a_size && borrow; ++i)
    borrow = sub_usize_borrow(a: a_ptr[i], b: 0, borrow_in: borrow, out: &a_ptr[i]);

  while (a_size > 0 && a_ptr[a_size - 1] == 0)
    --a_size;
}
```

Figure 4: Full multi-digit subtraction with borrow

Multiplication (Naive)

The naive way to do multiplication is similar to the vertical multiplication we do on paper.

```
1 for i from 0 to length(a) − 1 do
2     carry ← 0
3     for j from 0 to length(b) − 1 do
4         tmp ← a[i] · b[j] + c[i + j] + carry
5         c[i + j] ← tmp mod base
6         carry ← tmp ÷ base
```

```
7   | while carry > 0 do
8   |   | tmp ← c[i + length(b)] + carry
9   |   | c[i + length(b)] ← tmp mod base
10  |   | carry ← tmp ÷ base
```

Obviously, this has a time complexity of $\mathbb{O}(n^2)$, which is not so efficient for really big numbers.

Multiplication (Karatsuba)

Consider two integers x and y, both in base b, with n digits (possibly with prefixed zero). Fix m so that $0 < m < n$, we have:

$$x = \alpha_1 b^m + \beta_1$$
$$y = \alpha_2 b^m + \beta_2$$
$$x \times y = c_1 b^{2m} + c_2 b^m + c_3$$

Substituting the first two equations into the third, we have:

$$(\alpha_1 b^m + \beta_1)(\alpha_2 b^m + \beta_2) = c_1 b^{2m} + c_2 b^m + c_3$$

Expanding it gives:

$$\begin{cases} c_1 = \alpha_1 \alpha_2 \\ c_2 = \alpha_1 \beta_2 + \alpha_2 \beta_1 \\ c_3 = \beta_1 \beta_2 \end{cases}$$

Notice that:

$$c_2 = (\alpha_1 + \alpha_2)(\beta_1 + \beta_2) - c_1 - c_3$$

Therefore, we just need those three components (smaller problems) to solve the multiplication.

Let's estimate the complexity of this algorithm:

Let $m = \lceil \frac{n}{2} \rceil$. Since x and y have length of n, we know that the multiplication c_1, c_3, and $(\alpha_1 + \alpha_2)(\beta_1 + \beta_2)$ all have length $\leq m$.

Therefore, complexity of one single recursion is:

$$\text{rec} : n \Rightarrow 3 \text{ fix}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathbb{O}(n)$$

Since problem of length n is converted to length $\frac{n}{2}$, number of recursions needed is $\lceil \log_2 n \rceil$.

From bottom up, for the kth level, there will be $3^{\lceil \log_2 n \rceil - k}$ problems.

Thus, the total complexity will be:

$$\lim_{n\to\infty} \sum_{k=0}^{\lceil \log_2 n \rceil} 3^{\lceil \log_2 n \rceil - k} \mathbb{O}\left(2^k\right) = \lim_{n\to\infty} 3^{\log_2 n} \sum_{k=0}^{\log_2 n} 3^{-k} \mathbb{O}\left(2^k\right)$$

$$= \mathbb{O}\left(\lim_{n\to\infty} 3^{\log_2 n} \sum_{k=0}^{\log_2 n} 3^{-k} \cdot 2^k \right)$$

$$\sim \mathbb{O}\left(3^{\log_2 n} \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)^k \right)$$

$$
\begin{aligned}
\text{prev} &= \mathbb{O}\left(3^{\log_2 n} \cdot 3\right) \\
&\sim \mathbb{O}\left(3^{\log_2 n}\right) \\
&= \mathbb{O}\left(n^{\log_2 3}\right)
\end{aligned}
$$

Multiplication (FFT)

For any number in base b, it can be considered as a polynomial where the variable is b. Therefore, we can optimize it with FFT to get a time complexity of $\mathbb{O}(n \log n)$.

Multiplication

To get a general purpose multiplication, we can combine all three methods:

1. Naive way for smaller numbers since it's cache-line friendly
2. Karatsuba approach for average numbers
3. FFT for really big ($n > 10^{10^5}$) numbers

Bibliography

[1] O. W. Team, “□□□□□ - OI Wiki.” [Online]. Available: https://oi-wiki.org/math/bignum/