Application of Discrete Models

Adam Zlehovszky

October 23, 2023

1 Representation of Integers

1.1 Euclidean division

If $a, b \in \mathbb{Z}$ with $b \neq 0$ then $\exists! q, r \in \mathbb{Z}$ such that $a = qb + r$ where $0 \leq r < |b|$. This is the *Euclidean division* or *long division* of the *dividend* a with the *divisor* b. The results of the division are the *quotient* q and the *remainder* r. The standard notation for the remainder is $a \bmod b$. In algorithmic setting we use $q, r \leftarrow \textbf{divmod}\,(a, b)$.

1.2 Number systems

Let $1 < b \in \mathbb{Z}$ be the *base* of the *number system*. For each $0 \leq n \in \mathbb{Z}$ there exists a unique $1 \leq d \in \mathbb{Z}$ and a unique set of *digits* $0 \leq n_1, n_2, \ldots, n_{d-1} < b$ all integers, such that

$$n = \sum_{k=0}^{d-1} n_k b^k.$$

If $n = 0$, then $d = 1$ and $n_0 = 0$. Otherwise $d = \lfloor \log_b n \rfloor + 1$ and we can extract the digits of n with long division, since

$$\begin{aligned} n &= n_{d-1} b^{d-1} + \cdots + n_2 b^2 + n_1 b + n_0 \\ &= \left(n_{d-1} b^{d-2} + \cdots + n_2 b + n_1 \right) b + n_0 \end{aligned}$$

where the quotient $n_{d-1} b^{d-2} + \cdots + n_2 b + n_1$ is a $d-1$ digit number and n_0 is the extracted digit.

We call n_0 the *least significant digit* and n_{d-1} the *most significant digit*. The storage order of digits is called *little endian* if we start at the least significant digits and move towards the most significant one. Otherwise it is called *big endian*.

1.3 Operations on Integers

1.3.1 Addition

Let us assume that we have two unsigned integers stored as digits in a number system with base b:

$$n^{(i)} = \sum_{k=0}^{d^{(i)}-1} n_k^{(i)} b^k,$$

for $i = 1, 2$. The following algorithm computes the digits of the sum $s = n^{(1)} + n^{(2)} = \sum_{k=0}^{d^{(s)}-1} s_k b^k$:

Algorithm 1 Standard addition

```
1: procedure StandardAddition(n^(1), n^(2))
2:     d^(s) ← max(d^(1), d^(2))
3:     c ← 0
4:     for k = 0, ..., d^(s) − 1 do
5:         c, s_k ← divmod(n_k^(1) + n_k^(2) + c, b)
6:     end for
7:     return s
8: end procedure
```

In Algorithm 1 we assume that $n_k^{(i)} = 0$ if $k \geq d^{(i)}$ for $i = 1, 2$. The time complexity of the standard addition is $O\left(d^{(s)}\right)$.

1.3.2 Multiplication

Let $n^{(i)}$'s defined same as above for $i = 1, 2$. We will compute the digits of the product $p = n^{(1)} \cdot n^{(2)} = \sum_{k=0}^{d^{(p)}-1} p_k b^k$ with the naive multiplication method:

The time-complexity of Algorithm 2 is $O(d^{(1)} \cdot d^{(2)}) = O(d^2)$, where $d = \max\left(d^{(1)}, d^{(2)}\right)$.

Karatsuba's idea for faster multiplication can be demonstrated on two-digit numbers. Let

$$x = x_1 b + x_0, \text{ and } y = y_1 b + y_0$$

with $0 \leq x_i, y_i < b$ integers. Naive multiplication of x and y is

$$\begin{aligned} z = xy &= (x_1 b + x_0)(y_1 b + y_0) \\ &= x_1 y_1 b^2 + (x_1 y_0 + x_0 y_1) b + x_0 y_0 \\ &= z_1 b^2 + z_1 b + z_0. \end{aligned}$$

This is 4 multiplication and 1 addition.

Algorithm 2 Naive multiplication

procedure NAIVEMULTIPLICATION($n^{(1)}$, $n^{(2)}$)
  $d^{(p)} \leftarrow d^{(1)} + d^{(2)}$
  for $k = 0, \ldots, d^{(p)} - 1$ **do**
    $p_k \leftarrow 0$
  end for
  for $j = 0, \ldots, d^{(2)} - 1$ **do**
    $c,\ p_{i+j} \leftarrow$ **divmod** $\left(p_{i+j} + n_i^{(1)} n_j^{(2)} + c, b\right)$
    end for
    $p_{d^{(1)}+j} \leftarrow c$
  end for
  return p
end procedure

Now we can express

$$\begin{aligned} z_1 &= x_1 y_0 + x_0 y_1 \\ &= x_1 y_0 + x_0 y_1 - x_1 y_1 + x_1 y_1 - x_0 y_0 + x_0 y_0 \\ &= (x_1 + x_0)\, y_1 + (x_1 + x_0)\, y_0 - x_1 y_1 - x_0 y_0 \\ &= (x_1 + x_0)\,(y_1 + y_0) - x_1 y_1 - x_0 y_0 \\ &= (x_1 + x_0)\,(y_1 + y_0) - z_2 - z_0. \end{aligned}$$

This is 3 multiplication and 3 additions. By extending this idea to more than two digits recursively, the multiplication algorithm performs $O(d^{\log_2 3}) \approx O(d^{1.58})$ single-digit multiplication.

Fast Fourier Transform based algorithms can achieve $O(d \log d)$ complexity.

1.4 Exponentiation

We want to compute x^n for some $1 \leq n \in \mathbb{Z}$ and x that has multiplication as an operation.

Naive exponentiation By repeated multiplication, we can compute

$$x^n = \underbrace{x \cdot x \cdots x}_{n \text{ times}}.$$

This method requires $n - 1$ multiplications.

Repeated squaring If $n = 2^s$ for $0 < s \in \mathbb{Z}$, then

$$x^{(2^s)} = \left(x^2\right)^{\left(2^{s-1}\right)}.$$

This way we can compute x^n with $\log_2 n = s$ multiplications with the algorithm below:

Algorithm 3 Repeated squaring

```
1: procedure RepeatedSquaring(x, s)
2:     y ← x
3:     for k = 0, ..., s − 1 do
4:         y ← y^2
5:     end for
6:     return y
7: end procedure
```

Fast exponentiation If we write $n =$ in binary, then

$$\begin{aligned} x^n &= x^{\left(\sum_{k=0}^{d-1} n_k 2^k\right)} \\ &= \prod_{k=0}^{d-1} x^{\left(n_k 2^k\right)} \\ &= \prod_{k=0}^{d-1} x^{\left(2^k\right)^{n_k}}. \end{aligned}$$

Since $y^{n_k} = y$ if $n_k = 1$ and $y = 1$ otherwise, we arrive at the following algorithm:

Algorithm 4 Fast exponentiation

```
1: procedure FastExp(x, n)
2:     y ← 1
3:     z ← x
4:     while n > 0 do
5:         n, r ← divmod (n, 2)
6:         if r = 1 then
7:             y ← y · z
8:         end if
9:         z ← z^2
10:    end while
11:    return y
12: end procedure
```

This algorithm requires $O(\log_2 n)$ multiplication.