Chapter 10

Euclidean algorithm

The Euclidean algorithm is one of the oldest numerical algorithms still to be in common use. It solves the problem of computing the greatest common divisor (gcd) of two positive integers.

10.1. Euclidean algorithm by subtraction

The original version of Euclid's algorithm is based on subtraction: we recursively subtract the smaller number from the larger.

10.1: Greatest common divisor by subtraction.

```
def gcd(a, b):
    if a == b:
        return a
    if a > b:
        gcd(a - b, b)
    else:
        gcd(a, b - a)
```

Let's estimate this algorithm's time complexity (based on $n = a+b$). The number of steps can be linear, for e.g. $gcd(x, 1)$, so the time complexity is $O(n)$. This is the worst-case complexity, because the value $x + y$ decreases with every step.

10.2. Euclidean algorithm by division

Let's start by understanding the algorithm and then go on to prove its correctness. For two given numbers a and b, such that $a \geqslant b$:

- if $b \mid a$, then $gcd(a, b) = b$,
- otherwise $\gcd(a, b) = gcd(b, a \bmod b)$.

$$\begin{tikzcd}
\boxed{gcd(24,9)} \arrow[rr, bend left, "24 \bmod 9 = 6"] & = & \boxed{gcd(9,6)} \arrow[rr, bend left, "9 \bmod 6 = 3"] & = & \boxed{gcd(6,3)} \arrow[rr, bend left, "6 \bmod 3 = 0"] & = & \boxed{3}
\end{tikzcd}$$

Let's prove that $gcd(a, b) = gcd(b, r)$, where $r = a \bmod b$ and $a = b \cdot t + r$:

- Firstly, let $d = gcd(a, b)$. We get $d \mid (b \cdot t + r)$ and $d \mid b$, so $d \mid r$.
  Therefore we get $gcd(a, b) \mid gcd(b, r)$.
- Secondly, let $c = gcd(b, r)$. We get $c \mid b$ and $c \mid r$, so $c \mid a$.
  Therefore we get $gcd(b, r) \mid gcd(a, b)$.

Hence $gcd(a, b) = gcd(b, r)$. Notice that we can recursively call a function while a is not divisible by b.

10.2: Greatest common divisor by dividing.

```python
def gcd(a, b):
    if a % b == 0:
        return b
    else:
        return gcd(b, a % b)
```

Denote by (a_i, b_i) pairs of values a and b, for which the above algorithm performs i steps. Then $b_i \geqslant Fib_{i-1}$ (where Fib_i is the i-th Fibonacci number). Inductive proof:

1. for one step, $b_1 = 0$,
2. for two steps, $b \geqslant 1$,
3. for more steps, $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$, then $a_k = b_{k+1}, a_{k-1} = b_k$, $b_{k-1} = a_k \bmod b_k$, so $a_k = q \cdot b_k + b_{k-1}$ for some $q \geqslant 1$, so $b_{k+1} \geqslant b_k + b_{k-1}$.

Fibonacci numbers can be approximated by:

$$Fib_n \approx \frac{(\frac{1+\sqrt{5}}{2})^n}{\sqrt{5}} \tag{10.1}$$

Thus, the time complexity is logarithmic based on the sum of a and b — $O(\log(a + b))$.

10.3. Binary Euclidean algorithm

This algorithm finds the *gcd* using only subtraction, binary representation, shifting and parity testing. We will use a *divide and conquer* technique.

The following function calculate `gcd(a, b, res)` $= gcd(a, b, 1) \cdot res$. So to calculate $gcd(a, b)$ it suffices to call `gcd(a, b, 1)` $= gcd(a, b)$.

10.3: Greatest common divisor using binary Euclidean algorithm.

```python
def gcd(a, b, res):
    if a == b:
        return res * a
    elif (a % 2 == 0) and (b % 2 == 0):
        return gcd(a // 2, b // 2, 2 * res)
    elif (a % 2 == 0):
        return gcd(a // 2, b, res)
    elif (b % 2 == 0):
        return gcd(a, b // 2, res)
    elif a > b:
        return gcd(a - b, b, res)
    else:
        return gcd(a, b - a, res)
```

This algorithm is superior to the previous one for very large integers when it cannot be assumed that all the arithmetic operations used here can be done in a constant time. Due to the binary representation, operations are performed in linear time based on the length of the binary representation, even for very big integers. On the other hand, modulo applied in algorithm 10.2 has worse time complexity. It exceeds $O(\log n \cdot \log \log n)$, where $n = a + b$.

Denote by (a_i, b_i) pairs of values a and b, for which the above algorithm performs i steps. We have $a_{i+1} \geqslant a_i, b_{i+1} \geqslant b_i, b_1 = a_1 > 0$. In the first three cases, $a_{i+1} \cdot b_{i+1} \geqslant 2 \cdot a_i \cdot b_i$. In the fourth case, $a_{i+1} \cdot b_{i+1} \geqslant 2 \cdot a_{i-1} \cdot b_{i-1}$, because a difference of two odd numbers is an even number. By induction we get:

$$a_i \cdot b_i \geqslant 2^{\lfloor \frac{i-1}{2} \rfloor} \tag{10.2}$$

Thus, the time complexity is $O(\log(a \cdot b)) = O(\log a + b) = O(\log n)$. And for very large integers, $O((\log n)^2)$, since each arithmetic operation can be done in $O(\log n)$ time.

10.4. Least common multiple

The least common multiple (*lcm*) of two integers a and b is the smallest positive integer that is divisible by both a and b. There is the following relation:

$$lcm(a, b) = \frac{a \cdot b}{gcd(a,b)}$$

Knowing how to compute the $gcd(a, b)$ in $O(\log(a+b))$ time, we can also compute the $lcm(a, b)$ in the same time complexity.

10.5. Exercise

Problem: Michael, Mark and Matthew collect coins of consecutive face values a, b and c (each boy has only one kind of coins). The boys have to find the minimum amount of money that each of them may spend by using only their own coins.

Solution: It is easy to note that we want to find the least common multiple of the three integers, i.e. $lcm(a, b, c)$. The problem can be generalized for the *lcm* of exactly n integers. There is the following relation:

$$lcm(a_1, a_2, \ldots, a_n) = lcm(a_1, lcm(a_2, a_3, \ldots, a_n))$$

We simply find the *lcm* n times, and each step works in logarithmic time.

Every lesson will provide you with programming tasks at `http://codility.com/train`.