algorithms and imperative programming (i)

Available at JTANG.DEV/RESOURCES

complexity measures

$\mathbf{O}(f)$ denotes a set of functions:
$\{g : \mathbb{N} \to \mathbb{N} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+, \forall n > n_0, g(n) \leq c \cdot f(n)\}$
$\boldsymbol{\Omega(f)}$ denotes a set of functions:
$\{g : \mathbb{N} \to \mathbb{N} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
$\boldsymbol{\Theta(f)}$ denotes $\boldsymbol{O(f) \cap \Omega(f)}$

euclid's algorithm

```
Algorithm EuclidGCD(a, b):
  Input: Non-negative integers a and b
  Output: gcd(a, b)

  if b = 0 then
    return a
  return EuclidGCD(b, a mod b)
end
```

correctness
Let $d = gcd(a, b)$ and $c = gcd(b, a - rb)$.
We need to show that $gcd(a, b) = gcd(b, a - rb)$, so $d = c$.
By definition of d, we have the number $\frac{(a-rb)}{d} = \frac{a}{d} - r\frac{b}{d}$ is an integer as $d|a$ and $d|b$ and we have also shown $d|a - rb$ hence $d \leq c$.
Now by definition of c, $\frac{a-rb}{c} = \frac{a}{c} - r\frac{b}{c}$ shows that $c|a$ as we know $r\frac{b}{c}$ is an integer and $\frac{a-rb}{c}$ is an integer, so we have $c \leq d$.

complexity
After the first call, the first argument is always larger than the second one. Denote a_i as the first argument of the ith recursive call of EuclidGCD. It is clear that the second argument of a recursive call is equal to a_{i+1} and we also have

$$a_{i+2} = a_i \bmod a_{i+1}$$

which implies the sequence a_i is strictly decreasing. We claim that

$$a_{i+2} < \frac{1}{2}a_i$$

case 1: $a_{i+1} \leq \frac{1}{2}a_i$, since the sequence of a_i's is strictly decreasing, we have

$$a_{i+2} < a_{i+1} \leq \frac{1}{2}a_i$$

case 2: $a_{i+1} > \frac{1}{2}a_i$, in this case $a_{i+2} = a_i \bmod a_{i+1}$, so we have

$$a_{i+2} = a_i \bmod a_{i+1} = a_i - a_{i+1} < \frac{1}{2}a_i$$

Thus the size of the first argument to the EuclidGCD method decreases by half with every other recursive call. Hence we have $\boldsymbol{O(log \max(a, b))}$.

modular arithmetic

```
Algorithm pow1(a, b, k):
  Input: Integers a, b, k
  Output: a^b mod k

  s = 1
  for i from 1 to b
    s = s * a mod k
  return s
end
```

The number of operations performed here is clearly $O(b)$, therefore the time complexity is $O(2^n)$ as the size of b is $\log_2 b$.

```
Algorithm pow2(a, b, k):
  Input: Integers a, b, k
  Output: a^b mod k

  d = a, e = b, s = 1
  until e = 0
    if e is odd
      s = s * d mod k
    d = d * d mod k
    e = floor(e / 2)
  return s
end
```

The number of operations performed here is proportional to the number of times e $(= b)$ can be halved before reaching 0, i.e. at most $\lceil log_2 b \rceil$. It follows that this algorithm has running time in $O(n)$.

primitive roots
We say that g is a **primitive root** with respect to p means that $\mathbb{Z}_p = \{1, 2, \cdots, p-1\} = \langle g \rangle = \{g^i \bmod p \,|\, i \in \mathbb{Z}\}$

```
Algorithm dl(y, g, p):
  Input: Integers y, g, p
  Output: x such that y = g^x mod p

  a = y mod p
  for x from 1 to p - 1
    b = pow2(g, x, p)
    if a = b
      return x
  end
end
```

The number of loop iterations is $O(p)$ and in each iteration, the pow2 call is $O(x)$. So the total number of operations is bounded by $O(px)$ but $x < p$ so this is also bounded by $O(p^2)$ which is $O(4^n)$ as the size of p is $log_2 p$.

El Gamal with private key x
public key (p, g, y) with $y = g^x \mod p$
cipher (a, b) with $a = g^k \mod p$ and $b = My^k \mod p$
message $M = b/(a^x) \mod p = b(a^x)^{-1} \mod p$