(Computational) number theory

Number Theory

- Number theory is the part of mathematics devoted to the study of the integers and their properties.
- Number theory has a long history.
  - E.g.: Chinese Remainder Theorem
    1700 years old
- For a long time, it had been regarded as pure mathematics and useless.
  - G. H. Hardy (prominent British mathematician):
    Pure mathematics is “beautiful” and “useless”.
    Applied mathematics is “trivial”, “ugly”, and “dull”
- However, number theory has found numerous applications in computer science in recent decades.


孫子算經卷上

Why now?

- We have not needed any number theory or "advanced math" until now
  - Practical private-key cryptography is based on stream ciphers, block ciphers, and hash functions
  - Lots of interesting and non-trivial crypto can be done without any number theory!

Why now?

- Reason 1: Culmination of “top-down” approach
  - For most cryptography, we ultimately need to assume some problem is hard
  - The “lowest-level” assumptions we can make are about hardness of certain problems in number theory

Why now?

- Reason 2: The public-key setting
  - Public-key encryption *requires* number theory (in some sense)

Our goal

- Cover basic number theory quickly!

- Cover the minimum needed for all the applications we will study
  - Some facts stated without proof
  - Can take entire class(es) devoted to this material

Computational number theory

- We will be interested in the computational difficulty of various problems
  - Different from most of mathematics!

- The *representation* of mathematical objects is crucial for understanding the computational efficiency of working with them

Computational number theory

- Measure running times of algorithms in terms of the *input lengths* involved
  - For integer x, we have $\|x\| = O(\log x)$, $x = O(2^{\|x\|})$
- An algorithm taking input x and running in time O(x) is an *exponential* time algorithm
  - Efficient algorithms run in time poly($\|x\|$)

Computational number theory

- Our goal: classify various problems as either "easy" or "hard"
  - I.e., polynomial-time algorithms known or not

- We will not focus on optimizations, although these are very important in practice
  - For "easy" problems: speed up cryptographic implementations
  - For "hard" problems: need to understand concrete hardness for concrete security

Representing integers

- Cryptography involves very large numbers!
- Standard (unsigned) integers (e.g., in C) are small, fixed length (e.g., 16 or 32 bits)
  - For crypto, need to work with integers that are much longer (e.g., 2000 bits)
- Solution: use an *array*
  - E.g., "bignum" = array of unsigned chars (bytes)
  - May be useful to also maintain a variable indicating the length of the array
  - Or, assume fixed length (which bounds the maximum size of a bignum)

Example: addition

- Now need to define all arithmetic operations on bignums
  - Note: all our examples assume non-negative numbers; in practice, may need to also handle negative numbers
- Assume as a subroutine a way to add two bytes modulo 2^8 (i.e., discarding overflow)

Step 1

- AddWithCarry(byte a, byte b, byte carry)

  // carry is 0 or 1

  – If a < 2^7 and b < 2^7 res=a+b+carry, carry=0

  – If a < 2^7 and b ≥ 2^7 res=a+(b-2^7)+carry

    - If res ≥ 2^7 res=res-2^7, carry=1
    - Else res=res+2^7, carry=0

  – If a ≥ 2^7 and b ≥ 2^7 res=(a-2^7)+(b-2^7)+carry, carry=1

- Note: a ≥ 2^7 iff msb(a)=1

Example: addition

- Add(bignum a, int L1, bignum b, int L2)
  - Use grade-school addition, using AddWithCarry byte-by-byte…

- Running time O(max{L1,L2}) = O(max{‖a‖,‖b‖})
  - If ‖a‖=‖b‖=n then O(n)
  - Is it possible to do better?
    - No – must read input (O(n)) and write output (O(n))

Example: multiplication

- What is the length of the result of a*b?
  - $\|ab\|=O(\log ab)=O(\log a + \log b) =O(\|a\|+\|b\|)$

- Use grade-school multiplication…
- Running time $O(\|a\|\cdot\|b\|)$
  - If $\|a\|=\|b\|=n$ then $O(n^2)$
- Is it possible to do better? $O(n \cdot \log n)$.
  - Surprisingly…yes!

Basic arithmetic operations

- As we have seen, addition / subtraction / multiplication can all be done efficiently
  - Using grade-school algorithms (or better)

- Division-with-remainder can also be done efficiently
  - Much harder to implement!

Modular arithmetic

- Notation:
  - [a mod N] is the remainder of a when divided by N
  - Note $0 \leq [a \bmod N] \leq N-1$

- $a = b \bmod N \Leftrightarrow [a \bmod N] = [b \bmod N]$

Modular Arithmetic

- **Lemma**
  For any $a, b \in \mathbf{Z}, N \in \mathbf{Z}^+$, $a \bmod \mathrm{N} = (a + bN) \bmod N$.
- **Proof**
  - By Euclid's Division Theorem, there exist unique $q, r, 0 \leq r < N$, such that
    $$a = Nq + r \qquad (1)$$
  - Similarly, there exist unique $q', r', 0 \leq r' < N$, such that
    $$a + bN = Nq' + r' \qquad (2)$$
  - Adding kN to both sides of (1):
    $$a + bN = N(q + b) + r$$
  - By the uniqueness in Division Theorem, we have
    $$r = r'$$
  - By definition of mod, $a \bmod \mathrm{N} = (a + bN) \bmod N$.

Modular arithmetic

- Note that
  [a+b mod N] = [[a mod N] + [b mod N] mod N]
  [a-b mod N] = [[a mod N] - [b mod N] mod N]
  and
  [ab mod N] = [[a mod N][b mod N] mod N]

- I.e., can reduce intermediate values
  - This can be used to speed up computations

Modular Arithmetic

- **Proof**:
  - By Euclid's Division Theorem, there exist unique q_1, q_2, s.t.

$$a = q_1 N + (a \bmod N)$$
$$b = q_2 N + (b \bmod N)$$

  - Adding these 2 equations and take modulo N

$$(a + b) \bmod N$$
$$= \big((q_1 + q_2)N + (a \bmod N) + (b \bmod N)\big) \bmod N$$

  - The theorem then follows from the previous Lemma.

Modular arithmetic

- Careful: not true for division!

- I.e., [9/3 mod 6] = [3 mod 6] = 3
  but [[9 mod 6]/[3 mod 6] mod 6] = 3/3 = 1
  - We will return to division later…

Modular arithmetic

- Modular reduction can be done efficiently
  - Use division-with-remainder
- Modular addition / subtraction / multiplication can all be done efficiently
  - We will return to division later

Exponentiation

- Compute a^b ?
  - $\|a^b\| = O(b \cdot \|a\|)$
  - Just writing down the answer takes *exponential* time!

- Instead, look at *modular* exponentiation
  - I.e., compute $[a^b \bmod N]$
  - Size of the answer $\leq \|N\|$
  - How to do it?
    - Computing a^b and then reducing modulo N will not work…

without computing a^b

Modular exponentiation

- Consider the following algorithm:

```
exp(a, b, N) {
    // assume b ≥ 0
    ans = 1;
    for (i=1, i ≤ b; i++)
       ans = [ans * a mod N];
    return ans;
  }
```

- This runs in time O(b * poly(‖a‖, ‖N‖))
  - This is an exponential-time algorithm!

Efficient modular exponentiation

- Assume $b = 2^k$ for simplicity
  - The preceding algorithm roughly corresponds to computing a*a*a*…*a
  - Better: compute $(((a^2)^2)^2\ldots)^2$
  - 2^k multiplications vs. k multiplications
    - Note $k = O(\|b\|)$

Modular Exponentiation

- How to compute

$$a^n \bmod \mathrm{N}$$

  efficiently for large n?

Repeated squaring method

  - Compute

    $a^2 \bmod \mathrm{N}$

    $a^{2^2} \bmod N = a^4 \bmod N = (a^2 \bmod N)^2 \bmod \mathrm{N}$

    $a^{2^3} \bmod N = a^8 \bmod N = (a^4 \bmod N)^2 \bmod \mathrm{N}$

    $\dots$

  - Write n in binary $n = (b_k \dots b_1 b_0)_2$
  - $a^n = a^{b_0 \cdot 1} \cdot a^{b_1 \cdot 2} \cdot a^{b_2 \cdot 2^2} \cdots \bmod N$

- Example: $n = 50 = 32 + 16 + 2 = (110010)_2$
  - $a^{50} = a^{2^1} a^{2^4} a^{2^5} \bmod \mathrm{N}$

a^2 a^{16} a^{32}

Efficient exponentiation

- Consider the following algorithm:

```
exp(a, b, N) {
     // assume b ≥ 0
     x=a, t=1;
     while (b > 0) {
        if (b odd)
          t = [t * x mod N], b = b-1;
        x = [x^2 mod N],  b = b/2; }
     return t; }
```

- Why does this work?
  - Invariant: answer is $[t \cdot x^b \bmod N]$
- Running time is polynomial in ‖a‖, ‖b‖, ‖N‖

Primes and divisibility

- Notation a | b
- If a | b then a is a *divisor* of b
- p > 1 is *prime* iff its only divisors are 1 and p
  - p is *composite* otherwise
- d = gcd(a, b) if both:
  - d | a and d | b
  - d is the largest integer with that property

Computing gcd?

- Can compute gcd(a, b) by factoring a and b and looking for common prime factors…
- **Example**

  $120 = 2^3 \cdot 3 \cdot 5 \qquad 500 = 2^2 \cdot 3^0 \cdot 5^3$

  $\gcd(120,500) = 2^{\min(3,2)} \cdot 3^{\min(1,0)} \cdot 5^{\min(1,3)} = 2^2 \cdot 3^0 \cdot 5^1 = 20$

- **Definition**

  The integers a and b are relatively prime if $\gcd(a, b) = 1$.
  - Example: 17 and 22

Computing gcd?

- Can compute gcd(a, b) by factoring a and b and looking for common prime factors…
  - This is not (known to be) efficient!

- Use *Euclidean algorithm* to compute gcd(a, b)
  - One of the earliest nontrivial algorithms!

Euclidean Algorithm

- **Lemma**
  Let $a = bq + r$, where a, b, q, and r are integers. Then $\gcd(a, b) = \gcd(b, r)$.
- **Proof**
  - Suppose that d divides both a and b. Then d divides any linear combination of a and b, including $a - bq = r$ (see first slide). Hence, any common divisor of a and b must also be any common divisor of b and r.
  - Suppose that d divides both b and r. Then d also divides the linear combination $bq + r = a$. Hence, any common divisor of a and b must also be a common divisor of a and r.
  - Therefore, $\gcd(a, b) = \gcd(b, r)$.

Euclidean algorithm

ALGORITHM B.7
The Euclidean algorithm GCD

Input: Integers a, b with $a \geq b > 0$
Output: The greatest common divisor of a and b

if b divides a
  return b
else return GCD$(b, [a \bmod b])$

gcd(500,120) = gcd(120,20) = gcd(20,0)

Euclidean Algorithm

```
gcd(a, b):
x ← a
y ← b
while y ≠ 0
      r ← x mod y
      x ← y
      y ← r
return y
```

- Examples:
- $\gcd(54, 12)$ 54=12·4+6

$= \gcd(12, 6) = 6$

- $\gcd(14, 6)$ 14=2·6+2

$= \gcd(6, 2) = 2$

Correctness of algorithm follows from previous lemma

- Termination is obvious

Proposition

- Given a, b > 0, there exist integers X, Y such that Xa + Yb = gcd(a, b)
- Moreover, d=gcd(a, b) is the *smallest* positive integer that can be written this way

- Can use the *extended Euclidean algorithm* to compute X, Y

GCDs as Linear Combinations

- **Theorem** (Bézout's Theorem)
  If a and b are positive integers, then there exist integers s and t such that $\gcd(a, b) = sa + tb$.

Examples

- $\gcd(54, 12) = (-4) \cdot 12 + 1 \cdot 54$
- $\gcd(14,6) = (-2) \cdot 6 + 1 \cdot 14$
- Instead of proving this theorem directly, we give an algorithm to find such s and t.

The Extended Euclidean Algorithm

- **Example**

  Express $\gcd(252,198)$ as a linear combination of 252 and 198.

- **Solution**
  - First find $\gcd(252,198)$
    1) $252 = 1 \cdot 198 + 54$
    2) $198 = 3 \cdot 54 + 36$
    3) $54 = 1 \cdot 36 + 18$
    4) $36 = 2 \cdot 18$
    5) $\gcd(252,198) = 18$

- Rewriting:
  - $54 = 252 - 1 \cdot 198$
  - $36 = 198 - 3 \cdot 54$
  - $18 = 54 - 1 \cdot 36$
- Substituting:

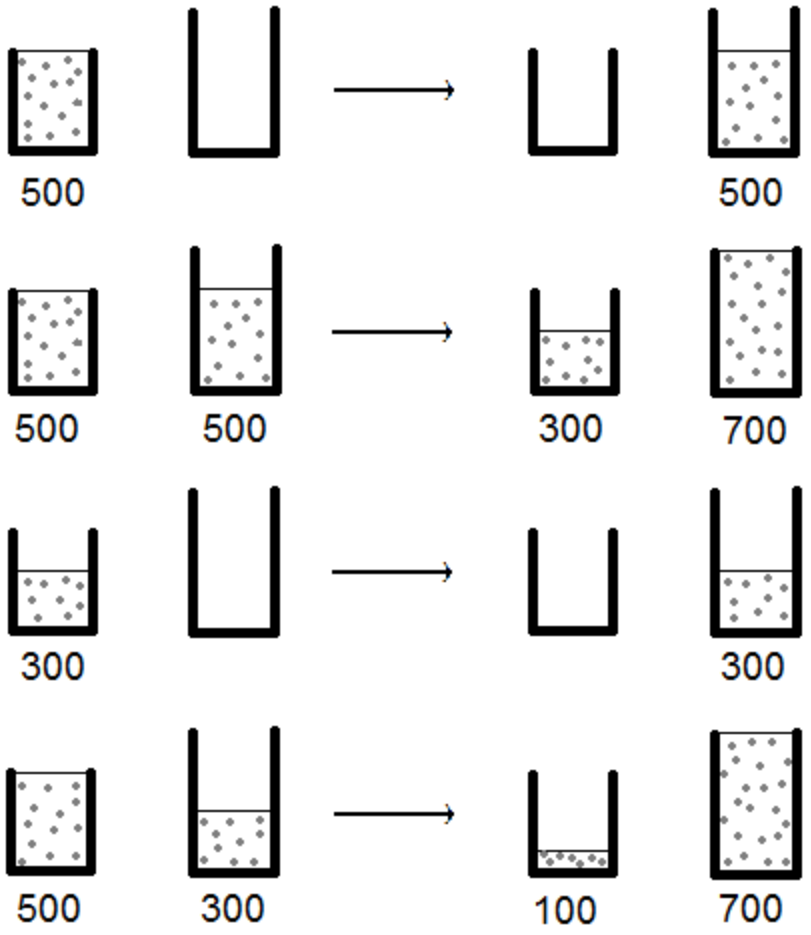
$$\begin{aligned} 18 &= 54 - 1 \cdot (198 - 3 \cdot 54) \\ &= 4 \cdot 54 - 1 \cdot 198 \\ &= 4 \cdot (252 - 1 \cdot 198) - 1 \cdot 198 \\ &= 4 \cdot 252 - 5 \cdot 198 \end{aligned}$$

Puzzle: Water Measuring

- What is the gcd of 500 and 700?
- 100
- Express $\gcd(700,500)$ as a linear combination of 700 and 500.
- Given
  - Two bottles: one has volume of 500 ml and the other one 700 ml.
  - Infinite water supply
- Goal: Get exactly 100 ml of water
- This follows exactly from

$$100 = \gcd(500,700)$$
$$= 3 \times 500 - 2 \times 700$$

- Corollary: Any multiple of the gcd can be obtained.



Group theory

Groups

- Introduce the notion of a *group*

- Provides a way to reason about objects that share the same mathematical structure
  - Not absolutely needed to understand crypto applications, but does make it conceptually easier

ℤ with addition ✓ group

Groups

takes ℤ

- An abelian *group* is a set G and a binary operation ◦ defined on G such that:
  - (**Closure**) For all g, h∈G, g◦h is in G
  - There is an identity e∈G such that e◦g=g for g∈G
  - Every g∈G has an inverse h∈G such that h◦g = g◦h = e
  - (**Associativity**) For all f, g, h∈G, f◦(g◦h) = (f◦g)◦h
  - (**Commutativity**) For all g, h∈G, g◦h = h◦g

- The *order* of a finite group G is the number of elements in G

Examples and non-examples

$\{0,1\}^*$ with concatenation. ✗ $\{0,1\}^2$ with XOR. ✓

- ✓ $\mathbb{Z}$ under addition ① ✓ ② 0 ③ $-a$ ④ ✓ ⑤ ✓
- ✗ $\mathbb{Z}$ under multiplication ① ✓ ② 1 ③ ✗
- ✓ $\mathbb{R}$ under addition ① ✓ ② 0 ③ $-a$ ④ ✓ ⑤ ✓
- ✓ $\mathbb{R}$ under multiplication ① ✓ ② 1 ③ ✗
- ✓ $\mathbb{R}\backslash\{0\}$ under multiplication ① ✓ ② 1 ③ ✓ ④ ✓ ⑤ ✓
- $\{0,1\}^*$ under concatenation ✓ empty ✗
- $\{0,1\}^n$ under bitwise XOR ✓ 0^n ✓ ✓ ✓
- 2 x 2 real, invertible matrices under mult. ✗

with AND ✗

Groups

- The group operation can be written *additively* or *multiplicatively*
  - I.e., instead of $g \circ h$, write $g+h$ or gh
  - Does *not* imply that the group operation has anything to do with (integer) addition or multiplication

- Identity denoted by 0 or 1, respectively
- Inverse of g denoted by $-g$ or g^{-1}, respectively
- Group exponentiation: $m \cdot a$ or a^m, respectively

Computations in groups

- When *computing* with groups, need to fix some representation of the group elements
  - Usually (but not always) some canonical representation
  - Usually want unique representation for each element
- Must be possible to efficiently identify elements in the group
- Must be possible to efficiently perform the group operation

  $\Rightarrow$ Group exponentiation can be computed efficiently

Useful example

- $\mathbb{Z}_N$ = {0, …, N-1} under addition modulo N
  - Identity is 0
  - Inverse of a is [-a mod N]
  - Associativity, commutativity obvious
  - Order N

Example

- What happens if we consider *multiplication* modulo N?

- {0, …, N-1} is *not* a group under this operation!
  - 0 has no inverse
  - Even if we exclude 0, there is, e.g., no inverse of 2 modulo 4

Modular inverses

- b is *invertible* modulo N if there exists an integer a such that ab = 1 mod N
  - Let $[b^{-1} \bmod N]$ denote the unique such a that lies in the range {0, …, N-1}

- Division by b modulo N is only defined when b is invertible modulo N
  - Then $[c/b \bmod N]$ is defined to be $[c\, b^{-1} \bmod N]$

Cancellation

- The “expected” cancellation rule applies for invertible elements
- I.e., if ab = cb mod N and b is invertible modulo N, then a = c mod N
  - Proof: multiply both sides by b^{-1}
- Note: this is not true if b is not invertible
  - E.g., 3*2 = 15*2 mod 8 but $3 \neq 15 \bmod 8$

Invertibility

- How to determine whether b is invertible modulo N?
- Thm: b invertible modulo N iff gcd(b, N)=1
- To find the inverse, use extended Euclidean algorithm to find X, Y with Xb + YN = 1
  - Then [X mod N] is the inverse of b modulo N
- Conclusion: can efficiently test invertibility and compute inverses!

Multiplicative Inverses

- **Definition**
  The (multiplicative) inverse of a modulo N is some b such that $ab = 1 \bmod \mathrm{N}$
- By default "inverse" means "multiplicative inverse".
- **Examples**

$\mathbf{Z}_5$:

a	1	2	3	4
a^{-1}	1	3	2	4

$\mathbf{Z}_7$:

a	1	2	3	4	5	6
a^{-1}	1	4	5	2	3	6

$\mathbf{Z}_6$:

a	1	2	3	4	5
a^{-1}	1	X	X	X	5

$\mathbf{Z}_8$:

a	1	2	3	4	5	6	7
a^{-1}	1	X	3	X	5	X	7

Multiplicative Inverses

- **Theorem**
  For any $a \in \mathbf{Z}_N$, N > 1, if $\gcd(a, N) = 1$ then a has a unique inverse in $\mathbf{Z}_N$.
- **Proof of Theorem**
  Since $\gcd(a, N) = 1$, by Bézout's Theorem, there are integers *s* and *t* such that $sa + tN = 1$.
  - Hence, $sa + tN = 1 \bmod$ N.
  - Since $tN = 0 \bmod N$, it follows that
    $$sa = 1 \pmod N$$
  - Consequently, *s* mod *N* is the inverse of *a* in $\mathbf{Z}_N$.
- **Corollary**
  For any prime p, every nonzero $a \in \mathbf{Z}_p$ has a multiplicative inverse.

Example

- Consider instead the *invertible* elements modulo N, under multiplication modulo N
- Define $\mathbb{Z}^*_N = \{0 < x < N : \gcd(x, N) = 1\}$
  - Closure
  - Identity is 1
  - Inverse of a is $[a^{-1} \bmod N]$
  - Associativity, commutativity obvious

$\phi(N)$

- $\phi(N)$ = the number of invertible elements modulo N

  $= |\{a \in \{1, \ldots, N-1\} : \gcd(a, N) = 1\}|$

  = The order of $\mathbb{Z}^*_N$

Two special cases

- If p is prime, then 1, 2, 3, …, p-1 are all invertible modulo p
  - $\phi(p) = |\mathbb{Z}^*_p| = p-1$

- If N=pq for p, q distinct primes, then the invertible elements are the integers from 1 to N-1 that are *not* multiples of p or q
  - $\phi(N) = |\mathbb{Z}^*_N| = ?$

Fermat's little theorem

- Let G be a finite group of order m. Then for any g $\in$ G, it holds that $g^m = 1$

- Useful in computing the remainders of large powers
- **Example:** Find $7^{222} \bmod 11$.

By the theorem, we know that $7^{10} \equiv 1 \pmod{11}$, and so $(7^{10})^k \equiv 1 \pmod{11}$, for any positive integer k. Therefore,

- $7^{222} = 7^{22 \cdot 10 + 2} = (7^{10})^{22} \cdot 7^2 \equiv 1^{22} \cdot 49 \equiv 5 \pmod{11}$

Examples

- In $\mathbb{Z}_N$:
  - For all $a \in \mathbb{Z}_N$, we have $N \cdot a = 0 \bmod N$

  (Note that N is not a group element!)

- In $\mathbb{Z}^*_N$:
  - For all $a \in \mathbb{Z}^*_N$, we have $a^{\phi(N)} = 1 \bmod N$
  - p prime: for all $a \in \mathbb{Z}^*_p$, we have $a^{p-1} = 1 \bmod p$

Fermat's little theorem

- Let G be a finite group of order m. Then for any $g \in G$, it holds that $g^m = 1$
  - Proof (abelian case)

Corollary

- Let G be a finite group of order m. Then for $g \in G$ and integer x, it holds that $g^x = g^{[x \bmod m]}$
  - Proof: Let $x = qm+r$. Then $g^x = g^{qm+r} = (g^m)^q g^r = g^r$

- This can be used for efficient computation…
  - …reduce the exponent modulo the order of the group before computing the exponentiation

Corollary

- Let G be a finite group of order m
- For any positive integer e, define $f_e(g)=g^e$
- Thm: If gcd(e,m)=1, then f_e is a permutation of G. Moreover, if $d = e^{-1} \bmod m$ then f_d is the inverse of f_e
  - Proof: The first part follows from the second. And $f_d(f_e(g)) = (g^e)^d = g^{ed} = g^{[ed \bmod m]} = g^1 = g$

Corollary

- Let N=pq for p, q distinct primes
  - So $|\mathbb{Z}^*_N| = \phi(N) = (p-1)(q-1)$
- If $\gcd(e, \phi(N))=1$, then $f_e(x) = [x^e \bmod N]$ is a permutation
  - In that case, let $[y^{1/e} \bmod N]$ be the *unique* $x \in \mathbb{Z}^*_N$ such that $x^e = y \bmod N$
- Moreover, if $d = e^{-1} \bmod \phi(N)$ then f_d is the inverse of f_e
  - So for any x we have $(x^e)^d = x \bmod N$
  - I.e., $[x^{1/e} \bmod N] = [x^d \bmod N]$!

Example

- Consider N=15
  - Look at table for $f_3(x)$
- N = 33
  - Take e=3, d=7, so 3rd root of 2 is…?
  - e=2; squaring is not a permutation…

- N=33

x	x^3 mod 33	x	x^3 mod 33
1	1	17	29
2	8	19	28
4	31	20	14
5	26	23	23
7	13	25	16
8	17	26	20
10	10	28	7
13	19	29	2
14	5	31	25
16	4	32	32