2017S Session03 Number Theoretic Programming

gcd(a, b) Euclid

gcd(12, 18)

see Wikipedia: 5 \* number of digits in base 10 log10

int bruteforcegcd(int a, int b) {

for i = min(a,b); i >= 2; i--) {

if (a % i == 0 && b % i == 0)

return i;

}

return 1;

}

gcd(3025, 1025)

gcd(a,b) = gcd(b, b mod a) → gcd(b, b - a)

gcd(a, 0) = a

gcd(55,34) worst case

int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

gcd(12, 18) → gcd(18, 6) → gcd(6, 18 mod 6) → gcd(6, 0)

gcd(3025,3024) → gcd(3024, 3025 mod 3024) → 1,0

gcd(3025,1) → gcd(1, 0)

gcd(3025, 1512) → gcd(1512, 1511)

gcd(3025, 1025) temp = 1025, b = 975 a = 1025

gcd(1025, 975)

gcd(975, 50)

gcd(50, 25)

gcd(25, 0)

gcd(3025, 3024)

gcd(3024, 1)

gcd(3025, 2)

gcd(2, 1)

worst case: consecutive fibonacci numbers

gcd(55,34)

O(logn)

lcm(a,b) lowest common multiple

lcm(12, 18) = 36 = 2 \* 3 \* gcd(a,b)

12=4\*3 18 = 6\*3

2\*2\*3 2\*3\*3

//O(log n)

lcm(a,b) = a \* b / gcd(a,b) // O(1 + log(n)) = O(log n)

bruteForceIsPrime(n)

for i ← 2 to

if n mod i == 0

return false

end

end

return true

end

bruteForceIsPrime(n)

for i ← 2 to

if n mod i == 0

return false

end

end

return true

end

O() Ω(1)

isPrime(n)

for i ← 2 to

if n mod i == 0

return false

end

end

return true

end

factor(n) O()

countPrimes(n) //O(n\*sqrt(n))

count ← 0  
 for i← 2 to n //O(n)

if isPrime(i) //O(sqrt(n))

count++

end

end

return count

end

primes from a = 1017 to b = a +106

compute all primes 2 .. sqrt(b)

Eratosthenes’ sieve ~2500 bp

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1

2

2 3 x 5 xy 7 x y xz 11 xy 13 x yz x 17 xy 19 xz y x 23 x z

n + n/2 + n/3 + n/5 + n/7 + n/11

//O(n log log n)

Eratosthenes(n)

primes[n] ← true [ x, x, t, t, t, t, t, t, t, t, ….]  
 for i ← 2 to n

if primes[i]

print i

for j ← i\*2 to n step i

primes[j] ← false

end

end

end

end

modifiedEratosthenes(n)

primes[n] ← true [ x, x, t, t, t, t, t, t, t, t, ….]

for i← 2\*2 to n step 2

prime[i] ← false

end  
for i ← 3 to n step 2

if primes[i]

for j ← i\*i to n step 2i 97\*97 97\*97 + 97

primes[j] ← false

end

end

print i

end

end

2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23

2,3,x,5,x,7,x,y,x,11,x,13,x,y,x,17,x,19,x,y,x,23

Big Arithmetic

all small numbers fit into a word 12314125 O(1)

big number: 124124198257192875192857192875192875912875912875912874912874912874219847191

n=4

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 9 | 9 | 9 |

+

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 9 | 9 | 9 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 9 | 9 | 9 | 8 |

= // answer is O(n+1) size

// 2n = O(n)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 | 2 | 2 |

n=4

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 9 | 9 | 9 |

\*

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 9 | 9 | 9 |

= //size = O(2n)

// O(n2+n) = O(n2) → O(n log n) using convolutions (we will do this w/ matrices)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 9 | 9 | 5 |

ab = a \* a \* a \* a \* a….

5555555555555555555552

25555555555555555555555555555555555

2100

O(2bab)

//raise x to the power n

bruteforcepower(x, n) //O(n)

prod ← 1

for i ← 1 to n

prod ← prod \* x

end

return prod

end

You can do better: power algorithm

break down the exponent into bits: 17 = 10001

x17 = x1x16

x16 = (x8)2

x8 = (x4)2

x4 = (x2)2

x2 = x \* x

pow(x,2) → x\*x

power(x, n) //O(log n)

prod← 1

while n > 0  
 if x mod 2 == 1 //bitwise test x & 1

prod ← prod \* x

end

x ← x2

n← n/2 // n >>= 1

end

end

1\*2\*3\*4 = 24 \* 5 = 120

1012!

1012! mod 10= 0

// Compute xn mod m faster than (xn) mod m

powermod(x, n, m) //O(log n)

prod← 1

while n > 0  
 if x mod 2 == 1 //bitwise test x & 1

prod ← prod \* x mod m

end

x ← x2 mod m

n← n/2 // n >>= 1

end

end

<https://locklessinc.com/articles/256bit_arithmetic/>

<https://gmplib.org/>

<https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html>

find count of primes with no digit 7 between a,b, a=1014 b = 1014 + 114276

# 

# 

# Fermat



https://primes.utm.edu/notes/proofs/FermatsLittleTheorem.html

## Fermat’s Last Theorem

cn = an + bn not true for n > 3 interesting, but not very useful. [Fermat's Room](https://en.wikipedia.org/wiki/Fermat%27s_Room)

## Little Theorem

if p is prime, then ap-1 mod p == 1

p = a number to be tested for primality

a = [2..p-1] 2 <= a < p

a=2, p = 5

ap = 25-1 mod 5 = 16 mod 5 = 1

ap-1 mod p ===1

p = 17 test by dividing 2,3,4

p = 170000000000000000001

Fermat test:

p = 17

a = 2

ap-1 mod p == 1

216 mod 17 = 65536 mod 17 = 1

p = 170000000000000000001

a = 2, 217000000000000000000 mod 170000000000000000001 == 1

a = 31, 3117000000000000000000 mod 170000000000000000001 == 1

Carmichael Number 3 \* 11 \* 17 = 561

Carmichael Numbers are “square free” 11\*11 (NO)

n! 6! = 720

compute 1012!

compute 1012! mod 10 last 108 = 0

power(x, n)

prod ← 1

while n > 0

if n % 2 != 0

prod ← prod \* x

end

x ← x2

n ← n / 2

end

return prod

end

ap-1 mod p

5! = 120

100! = >?????

100! mod 10

//x = A \* B

//x mod m = ((a mod m)(b mod m)) mod m

O(log n)

powermod(x, n, m)

prod ← 1

while n > 0

if n % 2 != 0

prod ← prod \* x **MOD m**

end

x ← x2 **MOD m**

n ← n / 2

end

end

# Fermat

https://en.wikipedia.org/wiki/Proofs\_of\_Fermat%27s\_little\_theorem

ap-1 mod p == 1 probably prime

a=2, p = 4, p-1 = 3

23 mod 4 = 8 mod 4 = 0, p=4 is definitely NOT prime

a=2, p=5, p-1 = 4, 24 mod 5 = 16 mod 5 = 1, 5 is probably prime

proof by counting necklaces (from Wikipedia)

For example, if *p* = 5 and *a* = 2, then we can use an alphabet with two symbols (say *A* and *B*), and there are 25 = 32 strings of length five:

***AAAAA***, *AAAAB*, *AAABA*, *AAABB*, *AABAA*, *AABAB*, *AABBA*, *AABBB*,

*ABAAA*, *ABAAB*, *ABABA*, *ABABB*, *ABBAA*, *ABBAB*, *ABBBA*, *ABBBB*,

*BAAAA*, *BAAAB*, *BAABA*, *BAABB*, *BABAA*, *BABAB*, *BABBA*, *BABBB*,

*BBAAA*, *BBAAB*, *BBABA*, *BBABB*, *BBBAA*, *BBBAB*, *BBBBA*, ***BBBBB***.

ap - a mod p == 0 In other words, ap - a is always divisible by p

ap == a mod p

divide both sides by a

ap-1 = 1 mod p

if not 1, definitely not prime!

What about if it IS 1???

fermat(p, k)

for i ← 1 to k

a← random(2, p-1)

if powermod(a, p-1, p) != 1

return false

end

end

return true (probably)

end

False positives: Carmichael Numbers

561=3\*11\*17

The only way to find that 561 is not prime is to find the actual number that comprises it. So fermat(561) will fail for witness 3,11 or 17 because that is one of the factors

This is equivalent to trial division. It will fair for any other factor.

1105

# Miller Rabin

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

n=221 11011101

221-192=29

n-1 11011100 1101110000000000

d=55

write *n* − 1 as 2*r*·*d* with *d* odd by factoring powers of 2 from *n* − 1  
WitnessLoop: **repeat** *k* times:  
 pick a random integer *a* in the range [2, *n* − 2]  
 *x* ← *ad* mod *n*  
 **if** *x* = 1 or *x* = *n* − 1 **then**  
 **continue** WitnessLoop  
 **repeat** *r* − 1 times:  
 *x* ← *x*2 mod *n*  
 **if** *x* = 1 **then**  
 **return** *composite*  
 **if** *x* = *n* − 1 **then**  
 **continue** WitnessLoop  
 **return** *composite*  
**return** *probably prime*

# Agrawal–Kayal–Saxena

2002 “Deterministically prove that n is prime WITHOUT TRIAL DIVISION

12741872641872641 8746b1827461827461872648171

RSA: P, Q both prime

N = P \* Q N has factors ONLY P,Q

# High Performance Eratosthenes

Back to Eratosthenes for a minute...

2, 3, 4, 5, 6, 7, 8, …

char isPrime[n]; // or byte for java

bool\* isPrime = new bool[n];

bool isPrime[n]; // c++

boolean isPrime[] = new boolean[n]; // java

A OR B = if either = 1 → 1

A AND B → both must be 1 → 1

a&b a|b

A B A AND B A OR B

0 0 0 0

0 1 0 1

1 0 0 1

1 1 1 1

00001000100000000000000000x0000

0000000000000000000000000000001 = 1

test if bit x is true

00001000100000000000000000x0000

0000000000000000000000000010000 1 << 4 0x8000000LL

-------------------------------

00000000000000000000000000x0000 either 0 or not 0 depending only on x

1 & x = x

00001000100000000000000000000000

00000000000000000000000001000000 << ??

00001000100000000000000000000000

00000000100000000000000000000000

00001000100000000000000000000000

11111111011111111111111111111111 1 << pos

00001000000000000000000000000000

50 → how many longs do you need 1

100 → 2

1 → 1 (1 + 63) / 64

64 → (64 + 63) = 127 / 64 → 1

65 → (65 + 63) = 128/ 64 → 2

class Bitvec {  
private:

uint64\_t\* bits;  
public:

Bitvec(size\_t size, bool initial) {

const int NUM\_WORDS = (size + 63) / 64; //109 / 64

uint64\_t v = (uint64\_t)-1L;  
 bits = new uint64\_t[NUM\_WORDS];

for (int i = 0; i < NUM\_WORDS; ++i)  
 bits[i] = v;  
 }

void set(uint64\_t i) {  
 bits[i / 64] |= (1 << (i % 64)) // /64 ⇒ >>6 // mod 2n & 0x3F  
 }

void clear(uint64\_t i) {  
 bits[i / 64] &= ~(1 << (i % 64))  
 }  
 bool test(uint64\_t i) {  
 return (bits[i / 64] & (1 << (i % 64)) != 0;  
 }

void toggle(uint64\_t i) {  
 bits[i/64] ^= (1 << (i % 64));  
 }  
};

xx110101010101010101010101010101

xxx1 1 1 0 1 1 0 ….

3 5 7 9 11 13 15

Don’t represent even numbers!

Storage requirements for Eratosthenes’ sieve

n = 109

with each boolean as 1 byte 109 byte

stored as bits 125Mb

no even numbers 62.5Mb

xx1101010101010101010101010101010101

Operations count

first mark every number prime n

mark all even numbers NOT prime n/2

mark all n mod 3 NOT prime n/3

* n/5 + n/7 + n/11 + n/13 + ...

1110110110 1000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000

# Prime Number Wheel

for (int i = 3; i < n; i+= 2)

2, 3

2\*3 = 6

6 7 8 9 10 11 12 13 14 15 16 17

0 x 0 0 0 x 0 x 0 0 0 x

18 19 20 21 22 23 24 25 26 27 28 29

0 x 0 0 0 x 0 x 0 0 0 x

for (int i = 6; i < n; i += 6)

check(i+1); check(i+5);

2,3,5 = 30

7 8 9 10 11 12 13 14 15 16 17 18 19

x 0 0 0 x 0 x 0 0 0 x 0 x

20 21 22 23 24 25 26 27 28 29 30 31 32

0 0 0 x 0 0 0 0 0 x 0 x 0

33 34 35 36

0 0 0 0

37 38 39 40 41 42 43 44 45 46 47 48 49

x x x x x

50 51 51 53 54 55 56 57 58 59 60 61 62

... 66

2x3x5x7 = 210

lcm(105,64) = 105\*64

lcm(210, 64) = 105 words

2 x 3 x 5 x 7 = 210 105 \* 64

Bitvector wordsize = 64 = 26

bits[0] bits[1] bits[2] … bits[104]

111011011xxxxxxxxxxxxx

bits[105] bits[106] bits[107]

=bits[0] =bits[1] …

mark every multiple of 11, 13, 17

index = p / 2 = p >> 1

1111111111111111111111111111111111111111111111111111111111111111

1110110111111111111111111111111111111111111111111111111111111111

bits[0] .. bits[104]

sieve[0…. 100000000]

countPrimes(long a, long b)

for (long i = a; i <= b; i++) {

bool primes[int(i-a)]

for (i = 6; i < n; i+= 6

if (check(i+1)) check(i+5)

# Encryption

Symmetric Key Encryption

In order to send a secret message, both sides must share a secret

Alice → Bob

key key

example: key = 0x55

message = “ABCDE” = 41 42 43 44 45

XOR

41 42 43 44 45

55 55 55 55 55

--------------------

14 17 16 11 10 to reverse, xor with the same key

55 55 55 55 55

--------------------

41 42 43 44 45 same answers again

public key encryption: keypub keypriv

# RSA

Rivest, Shamir, Adelman (Like the book, Cormen Leiserson ***Rivest*** and Stein)

1295879285719827519287519287519287519285719287591287591287591825791287519281

n = pq // O(n2) , p,q = 4096 bits = 4096 / 64 words

p ~ 4096 bits ~ q

n = 8192 bits

sqrt(28192) = 24000

b bits

what is the complexity to pick prime number p, q?

do

pick random number ←

check if it’s prime O(log 2b) = O(b)

while not prime

n = p \* q //O(b2) O(b log b)

p = 61, q = 53

n = p \* q = 3233

e = to encrypt choose number from 1 < e < = 17

d = to decrypt = modular multiplicative inverse of e = 413

public key = (n, e)

private key = (n, d)

message m = 65

c = me mod n 6517 mod 3233 = 2790

c = 2790 is the coded message

cd mod n → m 2790413 mod 3233 = 65

RSA is vulnerable to plaintext attack. If we know the text being sent, we can reverse engineer the key.

Therefore, we only send random numbers, using RSA to send the key that is then used to encrypt the message

RSA is used to exchange random keys  
AES-256 symmetric encryption

# Diffie-Hellman key exchange

https:

One of the two parties makes up two prime numbers g, p.

client: make up a random number a and computes gamod p, sends to the server

server make up a random number b, computes gbmod p, sends to client.

Both can now compute shared value c = gabmod p and use this value to encrypt with

Shared secret can then use assymetric encryption that is faster

server decrypts → key

Use symmetric encryption: current AES-256

Bruce Schneier Cryptography Addison-Wesley

digital signature

redaction validation

David Kahn The Codebreakers