Basic Number theory

Today..

- Simple primality test – in $O(\sqrt{n})$
- Sieve of Eratosthenes: Prime generation and Prime factorization
- Modular operations
- Euclidean algorithm: Greatest Common Divisor

What is a prime number?

- **A prime number** is a positive integer greater than 1 that has no positive divisors other than 1 and itself.
  - 17 is prime.
  - 169 is not prime, since its divisors are 1, ***13*** and 169.
  - 1 is not prime.
  - -17 is not prime.

What is a primality test?

- Given an positive integer N, check whether N is prime or not.
- Examples:
  - $N = 17$: N is prime.
  - $N = 169$: N isn't prime.
  - $N = 1$: N isn't prime.
- Prime numbers have lots of properties, so it is important to know whether a number is a prime or not.

How to check whether N is prime?

- Just use the definition:

```
bool is_prime (int N) {
  if(N <= 1) return false; // prime number should be greater than 1.
  for(int i = 2; i < N; i++) {
    // check whether i divides N or not.
    if(N % i == 0) return false;
  }
  return true;
}
```

- It takes $O(N)$ time.

How to check whether N is prime? (cont.)

- However, we don't need to check all divisors.
- If N is not prime, we can write $N = x \cdot y$, where $2 \leq x \leq y$.
  - Idea: $x \leq \sqrt{N}$
  - Proof: If $x > \sqrt{N}$, $y > x > \sqrt{N}$. So $x \cdot y > \sqrt{N} \cdot \sqrt{N} = N$, a contradiction.
- So, if N is not prime, there is ***at least one divisor*** $\leq \sqrt{N}$.
- It is sufficient to check $2 \leq i \leq \sqrt{N}$.

How to check whether N is prime? (cont.)

- Just change the constraint:

```
bool is_prime (int N) {
  if(N <= 1) return false;
  // instead of i ≤ √N, use i² ≤ N, to avoid doubles.
  for(int i = 2; i * i <= N; i++) {
    if(N % i == 0) return false;
  }
  return true;
}
```

What is "Sieve of Eratosthenes"?

- Sometimes, we want to know which integers under N are prime, and which are not.
  - Ex) $N = 14$: 2, 3, 5, 7, 11 and 13 are prime. 1, 4, 6, 8, 9, 10, 12 and 14 are composite.
- If we use the $O(\sqrt{n})$ primality test, the time complexity is:

$$\sum_{i=1}^{n} \sqrt{i} \approx \int_{1}^{n} \sqrt{x}dx = \left[\frac{2}{3}x^{\frac{3}{2}}\right]_{1}^{n} = \frac{2}{3}(n\sqrt{n} - 1) = O(n\sqrt{n})$$

- It seems good, but we can improve more!

What is "Sieve of Eratosthenes"? (cont.)

- **Sieve of Eratosthenes** is an algorithm for finding *all prime numbers* up to any given limit N.
- We are going to explain the algorithm by showing an example: $N = 50$.

Sieve of Eratosthenes

1. Create a list of all integers from 2 to N.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

2. Initially, let $p = 2$, the smallest prime number.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

Idea: For any prime p, all multiples of p larger than p i.e. $2p, 3p, \cdots, np$ are composite.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

3. So, mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

3. So, mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

4. Now, find the smallest number greater than p which is ***not*** marked. We know $p = 3$.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

5. Mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

5. Mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

6. Now, find the smallest number greater than p which is ***not*** marked. We know $p = 5$.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

7. Mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

8. Now, find the smallest number greater than p which is ***not*** marked. We know $p = 7$.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

9. Mark every multiple of p.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

- Now, all multiples of 2, 3, 5, 7 are marked. Since $\sqrt{N} \approx 7.07$, all unmarked cells are guaranteed to be prime.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Sieve of Eratosthenes (cont.)

- Therefore, we can consider all unmarked numbers as *primes*.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Implementation of Sieve of Eratosthenes

```
const int MAXN = 100000;
bool t[MAXN + 1]; // initialize to ‘false’.
t[1] = true;
for(int p = 2; p <= MAXN; p++) {
  // if p is marked, p is composite.
  if(t[p]) continue;
  // otherwise, mark all multiples of p.
  for(int i = 2*p; i <= MAXN; i += p) t[i] = true;
}
```

Time complexity of Sieve of Eratosthenes

- Suppose we mark all multiples of i (even multiples of composite numbers!) Then, the number of iterations is about

$$\sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \approx \int_{1}^{n} \frac{n}{x} dx = n \log n$$

- Actually, it is proven that the time complexity is $O(n \log \log n)$, which is really fast.

Prime factorizing with Sieve of Eratosthenes

- We can easily factorize any integer below N with some changes of the algorithm.
- Idea: When we mark $p \cdot n$, we know "p is a prime divisor of $p \cdot n$"!
  - Example: When $p = 3$, we mark $6, 9, 12, 15, 18, \cdots$
  - During this process, we know $6, 9, 12, 15, 18, \cdots$ have 3 as a prime divisor.
- So instead of just 'marking', write the 'prime divisor'!

Prime factorizing with Sieve of Eratosthenes (cont.)

```
const int MAXN = 100000;
int w[MAXN + 1];// initialize to 0.
// if w[n] != 0, w[n] is the largest prime divisor of n.
for(int p = 2; p <= MAXN; p++) {
  if(w[p] != 0) continue;
  // if p has a prime divisor less than p, p is composite.
  w[p] = p;
  // otherwise, p is a prime divisor of p. mark all multiples of p.
  for(int i = 2*p; i <= MAXN; i += p) w[i] = p;
}
```

Prime factorizing with Sieve of Eratosthenes (cont.)

```
int n = 150;
while(n > 1) {
  printf("%d ", w[n]);
  n /= w[n];
}
```

Result: 5 5 3 2

What is a ‘modulo operation’?

- The ***modulo*** operation **”a modulo b”** finds the remainder after division of a by b.
  - "a" is called the *dividend*, "b" is called the *divisor*.
  - Example: $19 \bmod 5 = 4$, since $19 = 5 \cdot 3 + 4$
  - In C++, it is denoted by **“`a % b`“.**
- In math, the *remainder* is defined by the Euclidean division.

$$a = bq + r \ (q, r \text{ are integers, } 0 \leq r < |b|)$$

- So the ‘remainder’ can bedetermined even if a is negative.
  - Example: $(-19) \bmod 5 = 1$, since $-19 = 5 \cdot (-4) + 1$

Modular arithmetic

- We only consider when M is *positive*.
- Addition, subtraction and multiplication:

$$(a+b) \bmod M = \big((a \bmod M) + (b \bmod M)\big) \bmod M$$
$$(a-b) \bmod M = \big((a \bmod M) - (b \bmod M)\big) \bmod M$$
$$(a\times b) \bmod M = \big((a \bmod M) \times (b \bmod M)\big) \bmod M$$

- We omit the proof. You can prove by letting $a = M \cdot q_1 + r_1$, $b = M \cdot q_2 + r_2$.

Caution: Modulo of negative numbers

- When we run the following code:

```
int x = (-19) % 5;
printf("(-19) mod 5 = %d\n", x);
```

- The result is:

```
(-19) mod 5 = -4
```

- ..which is different from the result by definition of *remainder*.

$$(-19) \bmod 5 = 1$$

Caution: Modulo of negative numbers (cont.)

- Note that $(-1) + 5 = 4$. So if the dividend is negative, we can add the divisor to the result of the modulo operation.

`int x = (-19) % 5;` `printf("(-19) mod 5 = %d\n", x);`	`(-19) mod 5 = -4`	$(-4) + \|5\| = 1$

- Without casework, we can do it like this:

```
(a % b + b) % b
```

Applications of modular arithmetic

- Example: We would like to calculate $1209321 \times 819281912 \times 6598313 \times 121231$ modulo 100. How?
  - Method 1. $1209321 \times 819281912 \times 6598313 \times 121231 = 79254067743022838207924 6856$. So the answer is 56.
  - Method 2. It is sufficient to calculate $21 \times 12 \times 13 \times 31$.
    - $(21 \times 12) \bmod 100 = 252 \bmod 100 = 52$
    - $(52 \times 13) \bmod 100 = 676 \bmod 100 = 76$
    - $(76 \times 31) \bmod 100 = 2356 \bmod 100 = 56$
    - So the answer is 56.

Applications of modular arithmetic (cont.)

- We can know the results of addition/multiplication modulo M by **only considering integers between $\mathbf{0}$ and $\boldsymbol{M-1}$. (inclusive)**
- So if M is sufficiently small, we can only use built-in integer types.
  - If $M \approx 10^9$, we can use `int` for addition and `long long` for multiplication modulo M.
  - If $M \approx 10^{18}$, we can use `long long` for addition modulo M.

Applications of modular arithmetic (cont.)

- In some problems, authors ask us to compute **the answer modulo P *(mostly prime)***, because the answer is quite big and authors don't want to use super-large integers.
  - In most counting problems, the number of ways are very large.
  - Ex) $\binom{1000}{500} \approx 2.702 \times 10^{299}$, so it is impossible to represent in a `long long`-integer type. However, by some computation, we can easily find that $\binom{1000}{500} \bmod 1{,}000{,}000{,}007 = 159{,}835{,}829$.
  - We will discuss this next time.

Modular division?

- However it is impossible to know the result of division by:

$$(a \div b) \bmod M = \big((a \bmod M) \div (b \bmod M)\big) \bmod M$$

- Example:
  - $a = 75, b = 25, M = 5$
  - $\frac{a}{b} \bmod M = 3 \bmod 5 = 3$
  - $\frac{a \bmod M}{b \bmod M} = \frac{0}{0}$ (undefined)

Modular division? (cont.)

- However, for some dividend a and divisor b, we can define the *modular inverse* $\boldsymbol{a^{-1}}$ such that:

$$(a^{-1} \cdot a) \bmod b = 1$$

- We won't cover about this in this course.

Greatest common divisor

- ***Greatest common divisor(GCD)*** of two or more positive integers:
    - = the largest positive integer that divides each of the integers.
- Examples:
    - $\gcd(15,30) = 15$
    - $\gcd(18,27) = 9$
    - $\gcd(30,54,42) = 6$
- First, we are considering about GCD of ***two*** positive integers.

How to calculate GCD?

- Using prime factorization:

$$48 = 2 \times 2 \times 2 \times 2 \times 3 = 2^3 \cdot 3^1 \cdot 5^0$$

$$180 = 2 \times 2 \times 3 \times 3 \times 5 = 2^2 \cdot 3^2 \cdot 5^1$$

- GCD is,

$$2^{\min(3,2)} \cdot 3^{\min(1,2)} \cdot 5^{\min(1,2)} = 2^2 \cdot 3$$

- Prime factorization is quite hard, and it is too complicated for this problem.

Euclidean algorithm

- GCD has some properties. We are going to use
$$\gcd(a, b) = \gcd(a - b, b)$$
- When $a > b$.
- Proof: By proving the following..
  - $\gcd(a, b) \leq \gcd(a - b, b)$
  - $\gcd(a - b, b) \leq \gcd(a, b)$

Euclidean algorithm (cont.)

- $\gcd(a,b) \le \gcd(a-b,b)$
  - By definition: $g = \gcd(a,b)$ divides both a and b.
  - Let $a = g \cdot x$, $b = g \cdot y$
  - g divides $a-b$ too, since $a-b = g \cdot (x-y)$
  - g is a common divisor of $a-b$ and b.
  - $\gcd(a,b) \le \gcd(a-b,b)$

Euclidean algorithm (cont.)

- $\gcd(a-b,b) \leq \gcd(a,b)$
  - By definition: $g' = \gcd(a-b,b)$ divides both $a-b$ and b.
  - Let $a-b = g' \cdot x'$, $b = g' \cdot y'$
  - g' divides $(a-b)+b = a$ too, since $(a-b)+b = g \cdot (x'+y')$
  - g' is a common divisor of a and b.
  - $\gcd(a-b,b) \leq \gcd(a,b)$

Euclidean algorithm (cont.)

- Since $\gcd(a, b) = \gcd(a - b, b)$ holds, this is also true:
$$\gcd(a, b) = \gcd(a \bmod b, b)$$
- because $a \bmod b = a - q \cdot b$ where $q = \lfloor a/b \rfloor$.
- When $a \bmod b = 0$, $\gcd(a, b) = b = \gcd(0, b)$. So we define $\gcd(0, n) = n$ for all $n > 0$.
- Also, since $a > b$ and $b > a \bmod b$, let's write
$$\gcd(a, b) = \gcd(b, a \bmod b)$$
- instead.

Euclidean algorithm (cont.)

- In short:

$$\gcd(a,b) = \begin{cases} a, & b = 0 \\ \gcd(b, a \bmod b), & b \neq 0 \end{cases}$$

- If $a < b$, $\gcd(a,b) = \gcd(b, a \bmod b) = \gcd(b,a)$. So when $a < b$, this recurrence swaps them such that $a' > b'$ holds.

Euclidean algorithm (cont.)

- Example: Calculate the GCD of 48 and 180.
  - $\gcd(48,180)$
    $= \gcd(180,48)$ ∵ $180 = 48 \times 3 + \mathbf{36}$
    $= \gcd(48,36)$ ∵ $48 = 36 \times 1 + \mathbf{12}$
    $= \gcd(36,12)$ ∵ $36 = 12 \times 3 + \mathbf{0}$
    $= \gcd(12,0)$
    $= 12$

Implementation of Euclidean algorithm

```
int gcd (int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

- If you use large integers, `long long`-type is required.

Time complexity of Euclidean algorithm

- WLOG, suppose $a > b$. When $a < b$, only one step is more needed.
- Claim: When $a > b$, $(a \bmod b) < a/2$.
  - If $b \leq a/2$: it is obviously true.
  - If $b > a/2 \leftrightarrow 1 < a/b < 2$: $a \bmod b = a - \lfloor a/b \rfloor \cdot b = a - b < a - \frac{a}{2} = \frac{a}{2}$.
- Suppose two steps happened:

$$(a, b) \to (b, a \bmod b) \to \big(a \bmod b, b \bmod (a \bmod b)\big)$$

- Since $a \bmod b \leq a/2$, **a decreased by at least half** in two steps!

Time complexity of Euclidean algorithm (cont.)

- Euclidean algorithm stops when $b = 0$.
- Since $a > b$, when $a = 1$, the algorithm stops since $b = 0$.
- When two steps happened, a becomes at most $\left(\frac{1}{2}\right) a$.
- When $2k$ steps happened, a becomes at most $\left(\frac{1}{2}\right)^k a$.
- So, $k \leq \log_2 a$ holds, so only about $2 \log_2 a$ steps are needed.
- The time complexity of Euclidean algorithm is $\underline{\boldsymbol{O(\log \max(a, b))}}$!

Time complexity of Euclidean algorithm (cont.)

- This implies we can compute GCD of ***any two integers*** representable by ***built-in integer*** type (`long long` or `int`) **very fast**.
  - ..because these types uses 64 and 32 bits, respectively.

GCD of three or more integers?

- Now, we can use this property:

$$\gcd(a, b, c) = \gcd(\gcd(a, b)\,, c)$$

- So we can compute the GCD one by one. Just think of "GCD" as a binary operator.