Number Theory

Workshop in Competitive Programming – 234900

Agenda

- Gaussian elimination
- Fast exponentiation
- Combinatorics and probability
- GCD
- Primality tests

General Topics

Fast exponentiation, Gaussian elimination

Fast exponentiation

Fast Exponentiation

- **Goal**: Given a number A and an integer s, calculate A^s
  - s can be very large ($s > 2^{10}$)

- Naïve approach:
  - Multiply A by itself s times
  - Time complexity: $O(s)$
- Faster method:
  - Use the binary representation of s:
  $$s = 2^0 s_0 + 2^1 s_1 + \cdots + 2^t s_t$$
  - With this representation, our task becomes:
  $$A^s = A^{2^0 s_0} \cdot A^{2^1 s_1} \cdot \ldots \cdot A^{2^t s_t}$$

Fast Exponentiation – Cont.

- Our task is to calculate: $A^s = A^{2^0 s_0} \cdot A^{2^1 s_1} \cdot \ldots \cdot A^{2^t s_t}$
- A^{2^i} can be calculated iteratively: $A^{2^i} = A^{2^{i-1}} \cdot A^{2^{i-1}}$

- Time complexity:
  - In the worst case, we need t operations to calculate $A^{2^0}, \ldots, A^{2^t}$ and t multiplications to calculate A^s.
  - Overall time complexity: $O(t) = O(\log s)$ 

- This method can also be used for exponentiation $mod\ K$, and for fast exponentiation of $n \times n$ matrices.

Gaussian Elimination

Gaussian Elimination

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 2 & 2 & 8 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 0 & 0 & 0 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 0 & -2 & -3 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

- **Goal**: Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $\vec{b} \in \mathbb{R}^m$, find $\vec{x} \in \mathbb{R}^n$ such that: $A\vec{x} = \vec{b}$.
- There are three types of elementary row operations which may be performed on the rows of a matrix:
  - Type 1: **Swap** the positions of two rows.
  - Type 2: **Multiply** a row by a nonzero scalar.
  - Type 3: **Add** to one row a scalar multiple of another.
- If the matrix is associated to a system of linear equations, then these operations do not change the solution set.

Gaussian Elimination

- By combining the 3 elementary operations we can bring a system of equations into its **canonical form**, then solve it using **back substitution**.

- Refer to Wikipedia for pseudocode, and Stanford Notebook for implementation.

- Note that Gaussian elimination can be performed over any field, not just the real numbers.

Combinatorics & Probability

Fibonacci, Binomial coefficients, Catalan, Basic probability

Fibonacci Numbers

Fibonacci Numbers

- Fibonacci recurrence:
$$F(n) = F(n-1) + F(n-2)$$
- Naïve approach:
  - Compute each element from the previous two, $O(n)$.
- Can be computed in $O(\log n)$:
$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$$
  - Use fast exponentiation!
  - Similar technique can be used in other recurrences/DP problems
- Fibonacci numbers are exponential in n.
Beware of overflow!

Fibonacci Numbers - Zeckendorf

- Zeckendorf's theorem: Every integer can be written as a sum of Fibonacci numbers
- For example: $10 = 2 + 3 + 5 = 5 + 5 = 8 + 2$
- Require no two consecutive Fib. numbers
  $\Rightarrow$ unique representation
- Greedy algorithm: Add the largest possible Fib. number to the summation.

Binomial Coefficients

Binomial Coefficients

- $\binom{n}{k}$ - n choose k, number of ways to choose k elements from a set of n elements.
- $\binom{n}{k} = \frac{\mathrm{n}!}{k!(n-k)!}$
- Problem: Individual elements may be very large.
  - Cancel elements before multiplying
  - Compute using $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, $\binom{n}{0} = \binom{n}{n} = 1$
  - If many values are needed, compute the entire Pascal's triangle.

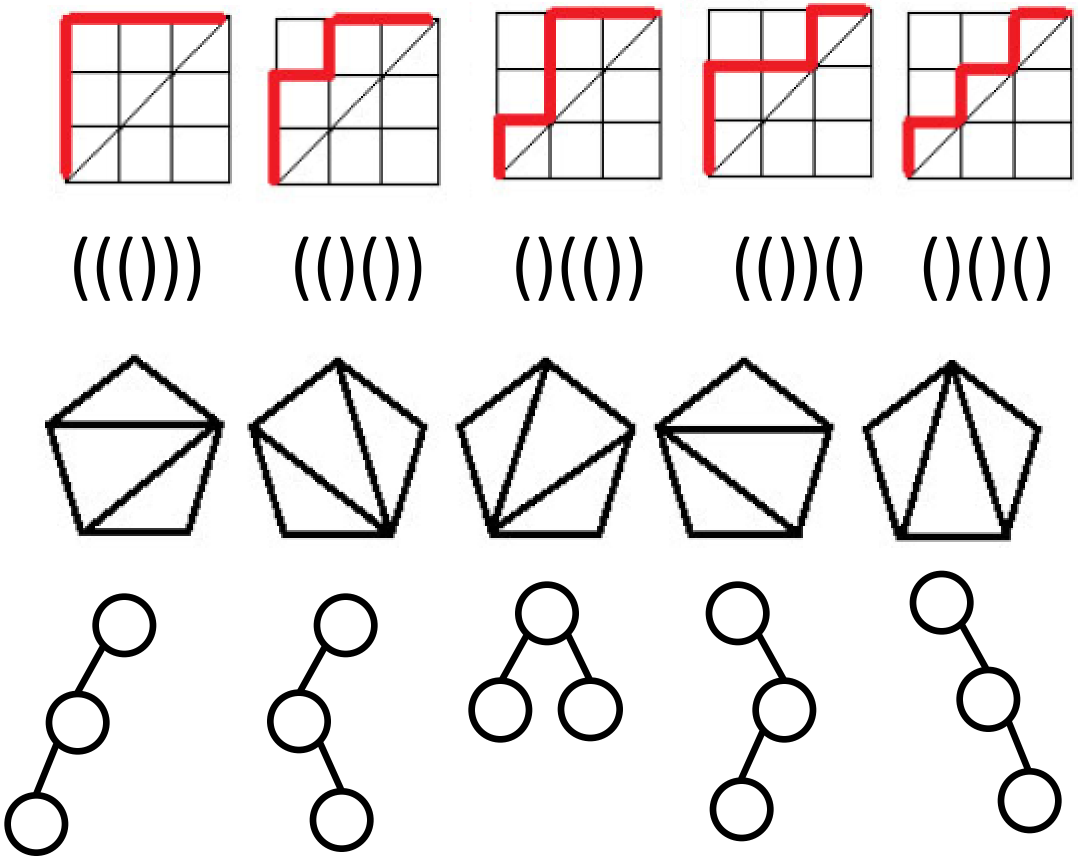
Catalan Numbers

Catalan Numbers

- Combinatorial problems which satisfies:
$$C(n+1) = \sum_{i=0}^{n} C(i)C(n-i)$$
- $C(n) = \frac{1}{n+1}\binom{2n}{n}$, $\mathrm{C}(0) = 1$
- $C(n+1) = \frac{(2n+2)(2n+1)}{(n+2)(n+1)} C(n)$
- Again, exponential in n, beware of overflow!

Catalan numbers

- Many combinatorial problems:



Number Theory

GCD, Modulo calculations, Primality testing

Greatest Common Divisor

GCD – Euclid's Algorithm

- **Goal**: Given $a, b \in \mathbb{N}$, we want to find $\gcd(a, b)$ - The largest number that divides both a and b.

- Useful property (Assuming $a \geq b$):
$$\gcd(a, b) = \gcd(a - b, b)$$
- Applying repeatedly until $a < b$ yields:
$$\gcd(a, b) = \gcd(a \, mod \, b, b)$$
- We now can swap $a \leftrightarrow b$ and repeat until $b = 0$

GCD – Euclid’s Algorithm

- We obtained the following recursive algorithm:
- function gcd(a, b) \\ assumes a >= b
  - if b = 0 return a
  - else return gcd(b, a mod b)

- Example:
  - $\mathrm{gcd}(30{,}12) = \mathrm{gcd}(12{,}6) = \mathrm{gcd}(6{,}0) = 6$

- Time complexity: $O(\log(\max(a, b)))$

LCM

- LCM – Least Common Multiple:

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

- Uses:
- Fraction operation
- Periodic prediction

Extended Euclid's Algorithm

- **Goal**: Given (a, b), find (u, v) such that:
$$au + bv = \gcd(a, b)$$

- **Application**: Solve an equation mod N:
$$ax = b \ (mod\ N)$$
  - Assuming a, N are coprime
  - Apply Extended Euclid's Algorithm to (a, N) to obtain (u, v) such that:
$$au + Nv = 1$$
  - Multiply by b, apply $mod\ N$ and obtain:
$$aub = b \ (mod\ N)$$
$$\Rightarrow \boldsymbol{x} = \boldsymbol{ub} \ (mod\ N)$$

Extended Euclid's Implementation

- An extension of the original algorithm:

```
// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}
```

Source: https://web.stanford.edu/~liszt90/acm/notebook.html#file13

Chinese Remainder Theorem

- Solving a set of modular equations:
  Chinese Remainder Theorem – [Wikipedia](#)
  [לא מדויק](#)

Primality Testing

Primality – Single Number

- **Goal**: Given a single number $N \in \mathbb{N}$, return true iff N is a prime number.

- Naïve approach: Brute force 🐌
  - Check all numbers in range $2 \ \ldots \sqrt{N}$
  - Time complexity: $O\left(\sqrt{N}\right)$
- Faster method: Miller-Rabin 🚀

Primality – Miller-Rabin

- Fermat's little theorem:
  If p is prime and p does not divides a, then
  $$a^{p-1} \equiv_p 1$$
- Fermat primality test:
  Pick a random a and check if $a^{p-1} \equiv_p 1$
- Small problem: We can draw a "bad" a, i.e. p is **not** prime but $a^{p-1} \equiv_p 1$.
  - Solution: Draw several different values of a.
- Big problem: There exists some composite numbers that pass Fermat test for **any** a (Carmichael numbers)

Primality – Miller-Rabin

- Second criterion:
  If p is prime and $x^2 \equiv_p 1$ then $x \equiv_p \pm 1$
- We want to compute a^{p-1}, write $p-1$ as $d \cdot 2^r$.
- a^{p-1}=$\left(a^d\right)^{\underbrace{2^{2^{2^{\cdots 2}}}}_{r \text{ times.}}}$
- If p is prime, and $a^{p-1} \equiv_p 1$ there must be $q < r$ for which $\left(a^d\right)^{\underbrace{2^{2^{2^{\cdots 2}}}}_{q \text{ times.}}} \equiv_p -1$

Primality – Miller-Rabin

```
bool MR(ll n, int k=5){
    if(n==1 || n==4)
        return false;
    if(n==2 || n==3)
        return true;
    ll m = n - 1;
    int r = 0;
    while (m%2 == 0){
        m/=2;
        r+=1;
    }
```

```
    while(k--){
        ll a = rand() % (n-4) + 2;
        a = powmodn(a,m,n);
        if(a==1) continue;
        int i = r;
        while(i-- && a != n-1){
            a = (a*a)%n;
            if(a == 1) return false;
        }
        if(i == -1) return false;
    }
    return true;
}
```

Primality – Miller-Rabin

- Complexity $O(k \log^3 n)$ (k is the number of repeats)
  - Can be slightly improved
- Probability of failure of Miller-Rabin (for $n > 32$) less than 4^{-k}
- Can be made deterministic using specific values of a.
- For example, for $n \leq 2^{64}$ it suffices to check only $\{2,3,5,7,11,13,17,19,23,29,31,37\}$.

Primality – Sieve of Eratosthenes

- **Goal**: Given $N \in \mathbb{N}$, find all prime numbers smaller than N.

- For each $k = 2, \ldots, \sqrt{N}$:
  - If k is not marked as “not prime”:
    - Mark k as “prime”
    - Mark $k \cdot k, (k+1) \cdot k, (k+2)k, \ldots N$ (all multiples of k up to N) as “not prime”

- Time complexity: $O(N \log \log N)$
- Space complexity: $O(N)$

Primality – Sieve of Eratosthenes

~~1~~	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~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Primality - Comparison

```
Check up to sqrt(n):
There are 2762 prime numbers below 25000
Exec time: 0.009526

Miller-Rabin:
There are 2762 prime numbers below 25000
Exec time: 0.247663

Sieve of Eratosthenes:
There are 2762 prime numbers below 25000
Exec time: 0.005514
```

Primality - Comparison

```
Check up to sqrt(n):
There are 22044 prime numbers below 250000
Exec time: 0.56401

Miller-Rabin:
There are 22044 prime numbers below 250000
Exec time: 0.429145

Sieve of Eratosthenes:
There are 22044 prime numbers below 250000
Exec time: 0.067179
```

Primality - Comparison

```
Check up to sqrt(n):
There are 13679318 prime numbers below 250000000
Exec time: 2402.77

Miller-Rabin:
There are 13679318 prime numbers below 250000000
Exec time: 250.563

Sieve of Eratosthenes:
There are 13679318 prime numbers below 250000000
Exec time: 44.3449
```

Primality - Conclusion

- To check a single prime:
  - Checking up to the root should suffice for most problems
  - If not fast enough we can use Miller-Rabin
- To generate all primes up to a number:
  - Sieve of Eratosthenes

Tips

General Tips

- Use $long\ long$ ($N \leq 2^{64}$) instead of int ($N \leq 2^{16}$)
  - Overflows can cause nasty bugs – Try to avoid them!
  - Calculate digit-by-digit if the number is still too long

- When working $mod\ N$, apply $mod\ N$ after each arithmatic operation in order to avoid overflows

- Sometimes the problem space is small enough to make brute-force possible

Competition Tips

- Team work
    - Always do something!
    - Fail your friends
    - Solve next problem
    - Write code on paper
- Notes
    - Very important!
    - Start now

Google

Event