Introduction to Programming Competitions

Mathematics

Dr Vincent Gramoli | Lecturer
School of Information Technologies

THE UNIVERSITY OF SYDNEY

Outline

- Combinatorics
  - Fibonacci numbers
  - Binomial coefficients
  - Catalan numbers
- Number theory
  - Prime numbers
  - GCD/LCM
  - Factorial
- Java BigInteger

Combinatorics

Combinatorics

- Combinatorics: A branch of mathematics concerning the study of countable discrete structures

- In programming contests, problem involving combinatorics usually ask the (possibly hidden) questions
  - How many [objects]?
  - Count [objects]…

- The code of the solution is usually short but finding the (usually recursive) formula requires some mathematic tricks

- Some solutions may also involve:
  - Dynamic programming (cf. lecture on the topic)
  - Java BigInteger class (cf. 1^{st} lecture)

Fibonacci Numbers

Fibonacci numbers

- Fibonacci numbers are defined as
  - fib(0) = 0, fib(1) = 1
  - fib(n) = fib(n-1) + fib(n-2)

- Example: 1, 1, 2, 3, 5, 8, 13, 21, 34…

- Fibonacci numbers grow very fast
- Some problems involving Fibonacci number requires the use of BigInteger

Fibonacci numbers

Zeckendorf's Theorem: *every positive integer can be written in a unique way as a sum of one or more Fibonacci numbers so that the sum does not include two consecutive Fibonacci numbers.*

- Given a positive integer n, such a sum can be found using a Greedy technique:
  - Take the largest Fibonacci number f lower than n
  - Reiterate with n-f instead of n until n-f = 0

Fibonacci numbers

O(1) approximation of the n^{th} Fibonacci number

- Binet's formula
  - $(p^n – (-p)^{-n})/\text{sqrt}(5)$
  - with p, the golden ratio: p = ((1+\sqrt(5)) / 2) ≃ 1.618
- Approximation
  - Take the closest integer to $(p^n – (-p)^{-n})/\text{sqrt}(5)$
  - Does not work well for large values of n

Fibonacci numbers

Pisano period. *The last one/last two/last three/last four digits of a Fibonacci number repeats with a period of 60/300/1500/15000, resp.*

Binomial Coefficients

Binomial coefficients

Binomial coefficient, ${}^{n}C_{k}$

- Number of ways that n items can be taken k at a time
- Coefficients of the algebraic expansion of powers of a binomial

Example:

- $\underline{1}x^3 + \underline{3}x^2y + \underline{3}xy^2 + \underline{1}y^3$
- The binomial coefficients of n=3 for $k \in \{0,1,2,3\}$ are 1, 3, 3, 1.

- ${}^{n}C_{k} = n!/((n-k)!k!)$
- Can be a challenge to compute when n and k are large

Binomial coefficients

Computing a single value ${}^{n}C_{k}$

- Use the formula ${}^{n}C_{k} = n!/((n-k)!k!)$
  - Can be a challenge to compute when n and k are large
- Tricks
  - Replace k by n-k because ${}^{n}C_{k} = {}^{n}C_{n-k}$
  - During computation, first divide, then multiply
  - Use BigInteger (last resort as BigInteger operations are slow)

Binomial coefficients

Computing many but not all values ${}^{n}C_{k}$ for different n and k

- Use top-down dynamic programming approach

- We can write ${}^{n}C_{k}$ and use a memo table to avoid re-computation
  - ${}^{n}C_{0} = {}^{n}C_{n} = 1$
  - ${}^{n}C_{k} = {}^{n-1}C_{k-1} + {}^{n-1}C_{k}$

Binomial coefficients

Computing all values ${}^{n}C_{k}$ for n=0 to a certain value of n

- Construct Pascal's triangle

- n=0 1
- n=1 1 1
- n=2 1 2 1
- n=3 1 3 3 1
- n=4 1 4 6 4 1
- …

Remember ${}^{n}C_{2} = O(n^2)$ as k=2 is a frequent case

Binomial coefficients

Computing all values ${}^{n}C_{k}$ for n=0 to a certain value of n

- Construct Pascal's triangle

- n=0 1
- n=1 1 1
- n=2 1 2 1
- n=3 1 3 3 1
- n=4 1 4 6 4 1
- …

Remember ${}^{n}C_{2} = O(n^2)$ as k=2 is a frequent case

Catalan Numbers

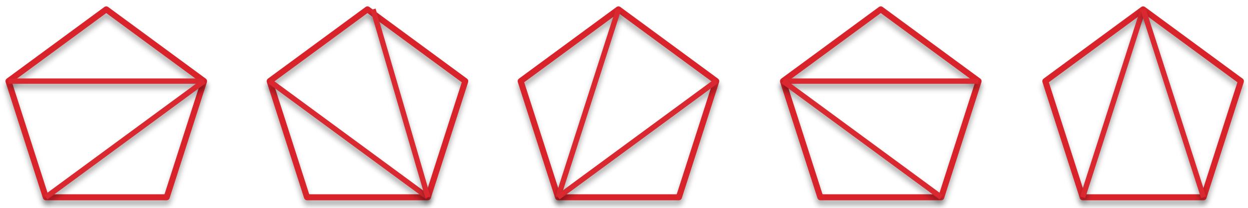
Catalan number

nth Catalan number

- Cat(n) = ${}^{2n}C_k$ / (n+1)
- Cat(0) = 1

Catalan numbers are found in various combinatorial problems:

- Cat(n) counts the number of expression containing n pairs of parentheses that are correctly matched
  - e.g., for n=3, we have ()()(), ()(()), (())(), ((())) and (()()).
- Cat(n) distinct binary trees with n vertices
- Cat(n) is the number of ways a convex polygon of n+2 sides can be triangulated



Catalan number

Compute the values of Cat(n) for several values of n

- Use bottom-up Dynamic Programming
- If we know Cat(n) we can compute Cat(n+1) using

- Cat(n) = 2n! / (n! n! (n+1))
- Cat(n+1) = (2(n+1))! / ((n+1)! (n+1)! ((n+1)+1))

  = (2n+2) (2n+1) / ((n+2) (n+1)) x Cat(n)

- If m = n+1, we have Cat(m) = 2m(2m-1) / ((m+1)m) x Cat(m-1)

Combinatorics in programming contest

- There are many other combinatorial problems in programming contests
- But they are not as frequent as Fibonacci numbers, Binomial coefficients or Catalan numbers

- In online programming contests
  - The Online Encyclopedia for Integer Sequence (OEIS) can be useful: http://oeis.org
  - Generate the output for small instances and give it to OEIS
  - If you are lucky OEIS will tell you the formula to generate larger instances

Number theory

Prime number

- A natural number starting from 2 is considered as a prime if it is only divisible by 1 or itself.

- Example: 2, 3, 5, 11, 13, 17, 19, 23, 29… and infinitely many more

Prime number

Testing whether a given natural number N is a prime

- Naïve: test if N is divisible by divisor $\in$ [2..N-1]
  - Works but takes O(N) divisions

Prime number

Improvements

1. Test if N is divisible by divisor $\in [2..\sqrt{N}]$
   - We stop when the divisor is greater than $\sqrt{N}$
   - If N is divisible by d, then $N = d \times N/d$
   - If $N/d < d$ then N/d or a prime factor of N/d would have divided N earlier
   - Therefore d and N/d cannot both be greater than $\sqrt{N}$
2. Test if N is divisible by divisor $\in [3, 5, 7.. \sqrt{N}]$
   - Only one even prime number, 2, that can be tested separately

   $\Rightarrow O(\sqrt{N}/2) = O(\sqrt{N})$
3. Test if N is divisible by prime divisors $\leq \sqrt{N}$
   - If a prime X cannot divide N, there is no point testing whether multiple of X divide N
   - Given that #primes $\leq M$ is $O(M / (\ln(M)-1))$

   $\Rightarrow O(\sqrt{N} / \ln(\sqrt{N}))$

Prime number

Sieve of Eratosthenes: Generate list of prime numbers in [0..N]

1. Initially, all numbers are potential primes
2. Exclude 0 and 1
3. Take 2 as prime and exclude all multiple of 2 (e.g., 4, 6, 8, 10…)
4. Take the next non-excluded number 3 as a prime and exclude all multiples of 3 (e.g., 9, 15…)
5. …
6. Once N is reached, you are left with all primes in [0..N]

$\Rightarrow$ N x (1/2 + 1/3 + 1/5 + 1/7 + … + 1/last prime in range $\leq$ N) operations

Using the 'sum of reciprocals of primes up to n' we end up with complexity of roughly O(N log log N)

Sieve of Eratosthenes (con't)

```
ll _sieve_size;
bitset<10000010> bs;   // 10^7 should be enough for most cases
vi primes;   // compact list of primes in form of vector<int>

void sieve(ll upperbound) { // create primes in [0..upperbound]
  _sieve_size = upperbound + 1;  // add 1 to include upperbound
  bs.set();                      // set all bits to 1
  bs[0] = bs[1] = 0;             // except index 0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i); // also add this vector
} }                          // call this method in main method
bool isPrime(ll N) {  // a good enough deterministic prime tester
  if (N <= _sieve_size) return bs[N]; // O(1) for small primes
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true; // it takes longer time if N is a large prime!
} // note: only work for N <= (last prime in vi "primes")^2
```

Prime number

Find the prime factors of a number N

- Naïve
  - Generates the list of primes (e.g., with Sieve of Eratosthenes)
  - Check which prime can actually divide integer N (without changing N)
  - This can be improved

- Divide-and-conquer strategy
  - An integer N can be expressed as N = P x N' where P is a prime factor
  - We can take out its prime factors by division until N'=1
  - We only repeat as long as $P \leq \sqrt{N}$

Prime number

Find the prime factors of a number N

› Divide-and-conquer strategy

  - Initially, `primes` is populated by Sieve of Eratosthenes

```
vi primeFactors(ll N) {   // vi, vect of int, ll is long long
  vi factors;             // vi `primes' is optional
  ll P_idx = 0, P = primes[P_idx]; // using PF = 2,3,4… is ok
  while (N != 1 && (P * P <= N)) {  // stop at sqrt(N)
    while (N % P == 0) {N /= P; factors.push_back(P);} // rem PF
    P = primes[++P_idx]; // only consider primes!
  }
  if (N != 1) factors.push_back(N); // special case if N is prime
  return factors; // if PF exceeds 32-bit int, you must change vi
}
```

Prime number

Find the prime factors of a number N

› Divide-and-conquer strategy

- Initially, `primes` is populated by Sieve of Eratosthenes
- As long as P ≤ √N and N ≠ 1

```
vi primeFactors(ll N) {   // vi, vect of int, ll is long long
  vi factors;             // vi `primes' is optional
  ll P_idx = 0, P = primes[P_idx]; // using PF = 2,3,4… is ok
  while (N != 1 && (P * P <= N)) {  // stop at sqrt(N)
    while (N % P == 0) {N /= P; factors.push_back(P);} // rem PF
    P = primes[++P_idx]; // only consider primes!
  }
  if (N != 1) factors.push_back(N); // special case if N is prime
  return factors; // if PF exceeds 32-bit int, you must change vi
}
```

Prime number

Find the prime factors of a number N

› Divide-and-conquer strategy
  - Initially, `primes` is populated by Sieve of Eratosthenes
  - As long as P ≤ √N and N ≠ 1
  - Divide N divides P then divide N by P and go on

```
vi primeFactors(ll N) {   // vi, vect of int, ll is long long
  vi factors;             // vi `primes' is optional
  ll P_idx = 0, P = primes[P_idx]; // using PF = 2,3,4… is ok
  while (N != 1 && (P * P <= N)) {  // stop at sqrt(N)
    while (N % P == 0) {N /= P; factors.push_back(P);} // rem PF
    P = primes[++P_idx]; // only consider primes!
  }
  if (N != 1) factors.push_back(N); // special case if N is prime
  return factors; // if PF exceeds 32-bit int, you must change vi
}
```

Prime number

Find the prime factors of a number N

› Divide-and-conquer strategy
  - Initially, `primes` is populated by Sieve of Eratosthenes
  - As long as P ≤ √N and N ≠ 1
  - Divide N divides P then divide N by P and go on
  - Special case: if N is prime at the end, add it to the list of primes

```
vi primeFactors(ll N) {   // vi, vect of int, ll is long long
  vi factors;             // vi `primes' is optional
  ll P_idx = 0, P = primes[P_idx]; // using PF = 2,3,4… is ok
  while (N != 1 && (P * P <= N)) {  // stop at sqrt(N)
    while (N % P == 0) {N /= P; factors.push_back(P);} // rem PF
    P = primes[++P_idx]; // only consider primes!
  }
  if (N != 1) factors.push_back(N); // special case if N is prime
  return factors; // if PF exceeds 32-bit int, you must change vi
}
```

Prime number

Count the number of prime factors of N

```
vi primeFactors(ll N) {
  ll P_idx = 0, P = primes[P_idx]; ans = 0;
  while (N != 1 && (P * P <= N)) {
    while (N % P == 0) {N /= P; ans++;}
    P = primes[++P_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

Prime number

Count the number of distinct prime factors of N

```
vi primeFactors(ll N) {
  ll P_idx = 0, P = primes[P_idx]; ans = 0;
  while (N != 1 && (P * P <= N)) {
    if (N % P == 0) { ans++; }
    while (N % P == 0) { N /= P; }
    P = primes[++P_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

Prime number

Count the number of divisors of N

- The divisors of N divide N without leaving a remainder
- If $N = a^i \times b^j \times \ldots \times c^k$ then N has $(i+1) \times (j+1) \times \ldots \times (k+1)$ divisors
- Example: $n = 60 = 2^2 \times 3^1 \times 5^1$ has 12 divisors {1,2,3,4,5,6,10,12,15,20,30,60}

```
ll numDiv(ll N) {
  ll P_idx = 0, P = primes[P_idx], ans = 1; // start from ans = 1
  while (N != 1 && (P * P <= N)) {
    ll power = 0;                         // count the power
    while (N % P == 0) { N /= P; power++; }
    ans *= (power + 1);                   // according to the formula
    PF = primes[++P_idx];
  }
  if (N != 1) ans *= 2; // (last factor has pow = 1, we add 1 to it)
  return ans;
}
```

Sum the divisors of N

- N=60 has 12 divisors whose sum is 168
- If N = $a^i \times b^j \times \ldots \times c^k$ then the sum of divisors of N is

$$(a^{i+1}-1) / (a-1) \ \times \ (b^{j+1}-1) / (b-1) \ \times \ \ldots \ \times \ (c^{k+1}-1) / (c-1)$$

```
ll numDiv(ll N) {
  ll P_idx = 0, P = primes[P_idx], ans = 1; // start from ans = 1
  while (P * P <= N) {
    ll power = 0;                     // count the power
    while (N % P == 0) { N /= P; power++; }
    ans *= ((ll)pow((double)P, power+1.0) - 1) / (P - 1);
    PF = primes[++P_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, power+1.0) - 1) / (N - 1);
  return ans;
}
```

Greatest Common Divisor

The Greatest Common Divisor (GCD) of two integers, a, b denoted by gcd(a,b) is the largest positive integer d such that d|a and d|b where x|y, means that x divides y (without leaving a remainder).

- Example: gcd(4,8) = 4, gcd(6,9) = 3

- One practical usage of GCD is to simplify fraction
  - e.g., 6/9 = (6/gcd(6,9)) / (9/gcd(6,9)) = (6/3) / (9/3) = 2/3

Greatest Common Divisor

Find the GCD of two integers

› Easy task with an effective divide and conquer Euclid algorithm

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

› The GCD is closely related to the Least Common Multiple (LCM) that is the smallest positive integer c such that a|c and b|c (e.g., lcm(6,9)=18)

- It was shown that lcm(a,b) = a x b / gcd(a,b)

```
int lcm(int a, int b) { return a * (b / gcd(a,b)); }
```

Greatest Common Divisor

Count the number of positive integers < N that are relatively prime to N

› Two integers a and b are relatively prime (coprime) if gcd(a, b) = 1

› Naïve approach
  - Count the number of positive integers < N that are relatively prime to N
  - Starts with counter = 0
  - Iterates through i ∈ [1..N-1]

Greatest Common Divisor

Count the number of positive integers < N that are relatively prime to N

- Euler’s Phi (Totien) function: $\varphi(N) = N \times \prod_P(1-1/P)$ w/ P a prime factor of N
  - Example: N=36 = 22 x 32, $\varphi(36) = 36 \times (1-1/2) \times (1-1/3) = 12$
  - There are 12 positive integers relatively prime to 36: {1,5,7,11,13,17,19,23,25,29,31,35}

```
ll EulerPhi(ll N) {
  ll P_idx = 0, P = primes[P_idx], ans = N; // start from ans = N
  while (N != 1 && (P * P <= N)) {
    if (N % P == 0) ans -= ans / P; // only count unique factor
    while (N % P == 0) N /= P;
    P = primes[++P_idx];
  }
  if (N != 1) ans -= ans / N; // last factor
  return ans;
}
```

Factorial

- Factorial of n, i.e., n! or fac(n)
  - 1 if n = 0
  - n x fac(n-1) if n > 0

- Largest factorial that can be represented in built-in data types is 20!

- Beyond this, we can work with prime factors at intermediate computation of large integers

Factorial

- Uva 10139 – Factovisors
  - Does m divides n! (where $0 \le n,m \le 2^{31}-1$)?

- Solution
  - We factorize m and n! to their prime factors
  - We check whether the prime factors of m are common to n!

- Example
  - $n = 6! = 2 \times 3 \times 4 \times 5 \times 6 = 2 \times 3 \times 2^2 \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$
  - If m = 9, then the answer is “yes” as $m = 3^2$ and $n!/m = 2^4 \times 5$
  - If m = 27, then the answer is “no” as $m = 3^3$ that does not divide n!

Java BigInteger

Java BigInteger

- When the intermediate and/or the final result of an integer-based mathematics computation cannot be stored inside the largest built-in integer data type and the given problem cannot be solved with any prime-power factorization we have no choice but to resort to BigIntger

- One way to implement BigInteger library is to store the BigInteger as a (long) string.
- For example
  - we can store 10^21 inside a string num1="1,000,000,000,000,000,000,000" without any problem whereas it is already an overflow in a 65-bit C/C++ unsigned long long (or Java long).
  - Then we can execute a digit-by-digit operation to process the BigInteger operand.

Java BigInteger

BigInteger supports the following operations

- add(BI)
- substract(BI)
- multiply(BI)
- pow(int exponent)
- divide(BI)
- remainder(BI)
- mod(BI)
- divideAndRemainder(BI)
- …

› However these are slower than the same operation on the standard 32/64-bit data type

Java BigInteger

UVa 10925 – Krakovia

- BigInteger additions to sum N large bills and
- divisions to divide the large sum to F friends.

UVa 10925 – Krakovia

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt(); // N bills,F friends
      if (N == 0 && F == 0) break;
      BigInteger sum = BigInteger.ZERO; // BigInteger ZERO
      for (int i = 0; i < N; i++) { // sum the N large bills
        BigInteger V = sc.nextBigInteger(); // read next BigInteger!
        sum = sum.add(V); // this is BigInteger addition
      }
      System.out.println("Bill #" + (caseNo++) + " costs " +
         sum + ": each friend should pay " +
        sum.divide(BigInteger.valueOf(F)));
      System.out.println(); // the line above is BigInteger division
} } }                            // divide the large sum to F friends
```

Code is short compared to if we had to write our own BigInteger routines

UVa 105515 – Basic Remains

- Given a base b and two non-negative integers p and m – both in base b, compute p%m and print the result as a base b integer.
- The base number conversion is actually a not-so-difficult mathematical problem, but this problem can be made even simpler with Java BigInteger class.
- We can construct and print a Java BigInteger instance in any base as shown below.

UVa 105515 – Basic Remains

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
      int b = sc.nextInt();
      if (b == 0) break;
      String p_str = sc.next();
      BigInteger p = new BigInteger(p_str, b); // special construc!
      String m_str = sc.next();
      BigInteger m = new BigInteger(m_str, b); // 2nd parameter base
      System.out.println((p.mod(m)).toString(b)); // output any base
} } }
```

Java BigInteger

UVa 10235 – (Probabilistic) Prime Testing

› Sieve of Eratosthenes may be too long to write

› If you just need to test whether a single (or at most several) and usually large integer is a prime there is an alternative with BI isProbablePrime

› A probabilistic prime testing function based on Miller-Rabin's algorithm

- It takes `certainty` as a parameter
- If the function returns true, then the proba that the BI is a prime is $1-(1/2)^{\texttt{certainty}}$
- For contests, `certainty` = 10 is enough as it leads to proba 0.9990…
- Larger certainty leads to higher proba but longer computation

UVa 10235 – (Probabilistic) Prime Testing

```
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  while (sc.hasNext()) {
    int N = sc.nextInt();
    BigInteger BN = BigInteger.valueOf(N);
    String R = new StringBuffer(BN.toString()).reverse().toString();
    int RN = Integer.parseInt(R);
    BigInteger BRN = BigInteger.valueOf(RN);
    System.out.printf("%d is ", N);
    if (!BN.isProbablePrime(10)) // 10 is enough for most cases
      System.out.println("not prime.");
    else if (N != RN && BRN.isProbablePrime(10))
      System.out.println("emirp.");
    else
      System.out.println("prime.");
} } }
```

UVa 10814 – Simplifying Fractions

› Reduce a large fraction to its simplest form by dividing both numerator and denominator with their GCD.

```
public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 int N = sc.nextInt();
 while (N-- > 0) { // unlike C/C++, we supply > 0 in (N-- > 0)
   BigInteger p = sc.nextBigInteger();
   String ch = sc.next(); // we ignore the division sign in input
   BigInteger q = sc.nextBigInteger();
   BigInteger gcd_pq = p.gcd(q); // wow
   System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
} } }
```

UVa 1230 – MODEX

- Compute $x^y \pmod{n}$

```java
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int c = sc.nextInt();
  while (c-- > 0) {
    BigInteger x = BigInteger.valueOf(sc.nextInt()); // valueOf conv
    BigInteger y = BigInteger.valueOf(sc.nextInt()); // simple int
    BigInteger n = BigInteger.valueOf(sc.nextInt()); // into BI
    System.out.println(x.modPow(y, n));  // it's in the library
} } }
```