Programming Challenges (GB21802)

Week 8 - Mathematics

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: June 6, 2021)

Version 2021.1

Math Problems: Lecture Outline

Every computer program requires some amount of mathematics. So what does **"Math Problems"** mean in Programming Challenges?

Here we describe two kinds of problems as **"Math Problems"**:

The Challenge is The Implementation of Mathematical Concepts

- Problems with Big Numbers (above variable limits)
- Problems with Geometry (next lecture!)

The Challenge Requires Mathematical Planning Before Programming

In this case, it is sometimes possible to solve the entire problem in paper and quickly implement a solution to the problem.

- Number Theory (primality testing, factorization, rings)
- Combinatorics (sequences, counting, recurrences)

Math Problems Part I: Large Numbers

Some programming challenges, in particular challenges involving combinatoric analysis, require operations on very large numbers.

In this section, we will review some ways to deal with these numbers:

- "BigNum" libraries;
- Modulo Operations;

Dealing with Large Numbers

In this lecture, we call "Large Numbers" (also sometimes **Bignum**) integers that do not fit in the standard variable types in programming languages (ex: long, long long, unsigned long, etc).

This is very common in problems and algorithms involving factorials. For example: $25! = 15511210043330985984000000 > 10^{26}$.

BigNum in Different Languages

- **C++ STL** does not have native support to bignum. You have to program yourself;
  - unsigned long long: $2^{64} < 10^{20}$
- **Java** has the "BigInteger" class, which contains several useful operations on large numbers;
- **Python** handles BigNums natively, so a special class is not necessary;

Sum and Division using Java's "Big Integer" class

```
import java.util.Scanner; import java.math.BigInteger;
class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt();
      if (N == 0 && F == 0) break;
      BigInteger sum = BigInteger.ZERO;                 // Bignum Constant
      for (int i = 0; i < N; i++) {
        BigInteger V = sc.nextBigInteger();             // Bignum I/O
        sum = sum.add(V); }                             // Bignum Addition
      System.out.println(
        "Total " + sum + ": Division: "
        + sum.divide(BigInteger.valueOf(F)) + "\n" );}  // Bignum division.
  }
}
```

Useful functions in Java.math.BigInteger

Besides dealing with arbitrarily large numbers, the BigInteger class also has some other useful mathematical functions:

Algebraic functions

BigInteger.add(), .subtract(), .multiply(), .divide(), .pow(), .mod(), .remainder()

Changing Number Base

```
BI = BigInteger(10); System.println(BI.toString(2))
// Result: 1010
```

Probabilistic Primality Test

```
isPrime = BI.isProbablePrime(int certainty)
// Chance of being correct is 1 - (1/2)^certainty
```

Other functions

BigInteger.gcd(BI) BigInteger.modPow(BI exponent, BI m)

Modulo Arithmetic

Another way to operate in very large numbers is to use **Modulo Arithmetic**.

For some problems, the final result is small (modulo n) but the intermediate results are too large. In these cases, we can use modulo arithmetic to avoid storing these large intermediate results.

Modulo Arithmetic Reminder

1. $(a + b)\%s = ((a\%s) + (b\%s) + s)\%s$
2. $(a * b)\%s = ((a\%s) * (b\%s))\%s$
3. $(a^n)\%s = ((a^{n/2}\%s) * (a^{n/2}\%s) * (a^{n\%2}\%s))\%s$

Example Problem: 10176, Ocean Deep! Make it Small

Problem summary

Your receive as input a **large binary number** (up to 100 digits). You need to calculate if the number is divisible by 131071 (a prime number).

- Problem: Input and store a large n, and calculate $n\%131071$.
- Two approaches:
  - Use a BigNum data structure to store n, and calculate.
  - Use modulo arithmetic to calculate the result **without** BigNum.

Part II: Number Theory

Number Theory studies the relationships between **integer numbers.**

It is a large and fascinating field of study, but for the purposes of programming contests, in this lecture we will focus on three topics:

- **Primality**: How to decide if a number is prime;
- **Division and Remainders**: The division relationship between integers;
- **Sequences**: Recurrence relations between sets of numbers;

Primality Testing

Prime Numbers are integers (> 1) that are only divisible by 1 and by themselves: $2, 3, 5, 7, 11, 13, \ldots$

Question: How do you write a (simple) program to test if N is prime?

- Complete Search: For each $d \in 2..N - 1$, test if $N\%d == 0$.
  - This requires $O(N)$ divisions.
- Pruning the search:
  - Search only numbers between 2 and $\sqrt{N}$: $O(\sqrt{N})$
  - Search only **odd** numbers between 2,3 and $\sqrt{N}$: $O(\frac{\sqrt{N}}{2})$
  - Search only **PRIME** numbers between 2 and $\sqrt{N}$: $O(\frac{\sqrt{N}}{\ln(\sqrt{N})})$
- Can we calculate all primes between 2 and $\sqrt{N}$ easily?

Primality Testing: Finding **Sets** of primes

The Prime Number Theorem (simplified)

There are approximately $\frac{N}{\log N - 1}$ prime numbers between 1 and N

- Number of prime numbers between 1 and $\sqrt{10^6} = 168$
- Number of prime numbers between 1 and $\sqrt{10^{10}} \approx 9500$

If we have a "list of prime numbers", we can calculate primality of many large numbers very quickly.

A simple algorithm to find a list of primes is **Sieve of Eratosthenes.**

Sieve of Eratosthenes

- Initialize "Sieve" vector of size $\sqrt{N}$, all TRUE;
- Loop on Sieve. If Sieve[i] is TRUE, add *i* to prime list
- Remove **ALL** $i \times m$ multiples of *i* from Sieve;

```
def sieve(k):                   ## Find all primes up to k
   primes = []                  ## List of primes found
   sieve = [1]*(k+1)            ## all numbers start in the list
   sieve[0] = sieve[1] = 0      ## 0,1 trivially not primes
   for i in range(k+1):         ## Linear search
      if (sieve[i] == 1):       ## Found a new prime
         primes.append(i)       ## Add to prime list
         j = i*i                ## Optimization. Why not i*2?
         while (j < k+1):       ## Costs O(loglogN)
            sieve[j] = 0        ## Remove multiples from sieve
            j += i
   return primes                ## list of primes
```

Sieve of Eratosthenes: Computation Cost

- The cost of calculating the Sieve for k is $O(k \log \log k)$
- The cost of full search for N is $O(\sqrt{N}/2)$
- Why use sieve and not the full search?

Amortized Complexity

Do a complex calculation once, use result many times:

- If we are only testing **ONE PRIME**, the full search is better.
- But, if the problem requires many primes to be tested, the sieve is better.
  - If $N < k$, checking the sieve table costs $O(1)$.
  - We can pre-calculate the sieve table when initalizing the program;

When do we need to calculate multiple primes? Prime factorization!

Prime Factorization

Every natural number N can be written as a **unique multiplication of primes**[1]. Example:

$$1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$$

In other words, for N, the prime number factorization of N is:

$$N = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n}, p_i \text{ is prime}$$

(Prime) Factorization is a key issue in Cryptography, so fast factorization is an important research problem. For programming challenges, we use two simple approaches:

- **Full search**: create a list of primes (with sieve) and test if each of them divides N.
- **Divide and Conquer:** Find the smallest prime p_i from sieve that divides N. Replace N with $N|p_i$. Repeat until $p_i > \sqrt{N}$.

[1] Fundamental Theorem of Arithmetics

Prime factorization: Divide and conquer approach

This algorithm is reasonably fast if N is composed of several small prime factors.

```
vector<int> primeFactors(ll N) {
  vector<int> factors;
  long PF_idx = 0, PF = sieve[PF_idx];    // sieve is a precomputed prime list
  while (PF * PF <= N) {                  // remember, N gets smaller;
    while (N % PF == 0) {                 // Remove PF^x from N
      N /= PF; factors.push_back(PF);
    }
    PF = primes[PF_idx++];                // only consider primes!
  }
  if (N != 1) factors.push_back(N);       // special case: N is prime
  return factors;
}
```

Full Factorization

In some cases, we want to know **all** numbers that divide a certain number N.

We can calculate the full factorization of N from its prime factorization.
In fact, the full factorization of N is the set of all unique combinations of prime factors.

Example:

- $1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$
- Number of factors of 1200: $5(2^4) \times 2(3^1) \times 3(5^2) = 30$
  - $2^0 \times 3^0 \times 5^1 = 5$,
  - $2^0 \times 3^0 \times 5^2 = 25$,
  - $2^0 \times 3^1 \times 5^0 = 3$,
  - $2^0 \times 3^1 \times 5^1 = 15$,
  - $2^0 \times 3^1 \times 5^2 = 75$,
  - $\ldots$

Factorization Problem Example: 10139 – Factovisors

Problem summary

Check if m divides $n!$ $(1 \leq m, n \leq 2^{31} - 1)$

The factorial of $n \leq 2^{31} - 1$ is a HUGE number. Fortunately, it is not necessary to calculate this number at all. Consider that:

- F_m: primefactors(m)
- $F_{n!}$: $\cup$(primefactors(1), primefactors(2) $\ldots$, primefactors(n))

We can say that m divides $n!$ iff $F_m \subset F_{n!}$.

Examples:

- $m = 48, n = 6, n! = 2 \times 3 \times 4 \times 5 \times 6$
  $F_m = \{2, 2, 2, 2, 3\}, F_{n!} = \{2, 3, 2, 2, 5, 2, 3\}$
- $m = 25, n = 6, n! = 2 \times 3 \times 4 \times 5 \times 6$
  $F_m = \{5, 5\}, F_{n!} = \{2, 3, 2, 2, 5, 2, 3\}$

Extended Euclid Algorithm

For integers *a* and *b*, the **greatest common divisor** GCD(a,b) is the largest integer *d* so that $d|a$ and $d|b$. Euclid's algorithm can quickly calculate *d* for a,b ($O(\log_{10} a)$).

The **Extended Euclid's Algorithm**[2], calculate's x_0 and y_0 so that $a \times x_0 + b \times y_0 = d$.

```
int gcdExtended(int a, int b, int *x, int *y)  {
  if (a == 0) { *x = 0; *y = 1; return b; }

  int x1, y1; // To store results of recursive call
  int gcd = gcdExtended(b%a, a, &x1, &y1);

  *x = y1 - (b/a) * x1; *y = x1;      // Update x,y

  return gcd;
}
```

[2] Also called "The Pulverizer"

Extended GCD and the Diophantine Equation

One very useful property of $d = \text{GCD}(a, b)$ is that d **divides every integer combination of** a **and** b. In other words: For every $ax + by = c$, if x and y are integers, then $d|c$.[3].

We can use this property to calculate the integer solutions of the **Diophantine Equation**: $xa + yb = c$

- If $d|c$ is not true, there are no integer solutions.
- If $d|c$ is true, there are infinite integer solutions:
  - The first solution (x_0, y_0) is calculated from the extended GCD.
  - Other solutions (x_n, y_n) can be derived as: $x_n = x_0 + (b/d)n, y_n = y_0 - (a/d)n$, where n is an integer.

[3] The proof for this is very cool

Diophantine Equation Problem Example

Problem Example

With 839 yens, you want to buy Candy X and Candy Y.

- Candy X costs $x = 25$ yens.
- Candy Y costs $y = 18$ yens.

How many candies can you buy?

1. Calculate d, x_0, y_0 from extended GCD:
   - $d = 1, x_0 = -5, y_0 = 7$. This means that $25 \times (-5) + 18 \times (7) = 1$
2. Is $d|c$? **Yes.** Continue.
3. Multiply both sides of the equation by $\frac{c}{d}$:
   - $25 \times (-5 \times 839) + 18 \times (7 \times 839) = 839$
4. It is impossible to buy negative candies, so we iterate on n to find
   - $x_n = x_0 + (y/d)n$ and $y_n = y_0 - (x/d)n$
5. At $n = 234$ we find: $25 \times 17 + 18 \times 23 = 839$

Combinatoric Problems

Combinatorics is the area of mathematics that studies **countable discrete structures** (integers, sets, etc).

For our focus on competitive programming, combinatoric problems involves calculating the values of **numeric sequences.** This requires programming the **Recurrent Form** or the **Closed Form** of the sequence.

- **Recurrent Form**: The recurrent form of a sequence F calculates F_n based on its antecessor values: $F_{n-1}, F_{n-2}, \ldots$.
  - Recurrent forms are usually implemented using **Dynamic Programming** and **Memoization**;
- **Closed Form**: The closed form of a sequence F calculates F_n **without** using the antecessor values of the sequence.

Sequence Example: Triangular Numbers

Definition

Triangular Numbers is the sequence where T_n is the sum of all inegers from 1 to n.
Example:

$T_1 = 1, T_2 = 1 + 2 = 3, \ldots, T_7 = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$
Trivial, right?

- **Recurrent Form:** $T(n) = T(n-1) + n$
  - The recurrent form can be calculated with a loop or recursion;
- **Closed Form:** $T(n) = \frac{n(n+1)}{2}$
  - The closed form can be calculated at once;
  - It can be used to estimate how fast a sequence grows. In this case, T_n is $O(N^2)$

A more famous sequence: Fibonacci Numbers

Definition

The Fibonacci number F_n is the sum of the two numbers before it.

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

- Recurrent Form:
  - Starting Values: $F_0 = 0$, $F_1 = 1$
  - Recurrence: $F_n = F_{n-1} + F_{n-2}$
- Be careful when implementing recurrences with multiple terms;
  - If using recursive functions, **memoization/DP** is necessary to avoid wasted calculation;
  - In general, each term in a recurrence requires a starting value;

Bonus: Fibonacci Facts

Closed Form for the Fibonacci Numbers:

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

The second term of the closed form tends to 0 when n is large!

Pisano's period

The last digits of the Fibonacci sequence repeat with a fixed period!

```
 Digits          | Period       || Digits          | Period
 last digit      | 60 numbers   || last 3 digits | 1500 numbers
 last 2 digits | 300 numbers || last 4 digits | 15000 numbers
F(6)   =                              8
F(66)  =                  27777890035288
F(366) =  1380356 ... 8899086435571688
```

Binomial Coefficient

Definition

Binomial Coefficients are the set of numbers that correspond to the expansion of a binomial:

- $B_3 = (a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + b^3 = \{1, 3, 3, 1\}$
- $B_5 = (a+b)^5 = 1a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + 1b^5 = \{1, 5, 10, 10, 5, 1\}$

Many times, we are interested in the k-th number of the n-binomial, written as $C(n, k)$ or ${}^{n}C_k$. Example: $C(5, 2) = 10$.

Binomial Coefficient

Interpretation and Recurrent Form

The common interpretation of $C(n, k)$ is "I have to select A or B n times, how many different ways can I choose A k times?"

- How many binary strings with n digits have k ones?
- How many paths exist

Using this definition, we can define the recurrent form of the Binomial:

- I have to choose A k times out of n
  - If I choose A $k-1$ times out of $n-1$, I choose A again.
  - If I choose A k times out of $n-1$, I choose B.
- $C(n, k) = C(n-1, k-1) + C(n-1, k)$
- Don't forget to use DP to implement this!

Pascal's Triangle

The recurrent form of the binomials:

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

Can also be observed by laying out the numbers:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

Closed Form for the Binomial

The closed form for $C(n, k)$ is:

$$C(n, k) = \frac{n!}{(n-k)!k!}$$

Be careful! As you remember, the value of $n!$ can become very big, very fast. It might be better to calculate the binomial using the recurrent form, to avoid overflow.

The Catalan Numbers

Motivating Problem

Given n pairs of parenthesis, how many different balanced expressions can you create?

- n = 0: . = 1
- n = 1: () = 1
- n = 2: ()(), (()) = 2
- n = 3: ((())), ()(()), (())(), (()()), ()()() = 5
- n = 4: 14
- n = 5: 42

This sequence is known as the **Catalan Numbers**, and it appears in several recursive combinatory problems.

The Catalan Numbers

Recurrent Form

The **Recurrent form** of the catalan number can be derived from the parenthesis definition:

- If we define c_k as an expression with k parenthesis, we can break it down into: $c_k = (c_a)c_b$, where $k = a + b + 1$.
- Varying the values of *a* and *b*, and counting all possible variations gives us the recurrent form:
- $c_{n+1} = \sum_{i=0}^{n} c_i c_{n-i}$

Closed Form and Usage

The closed form of the Catalan Numbers is:

$$c_n = \frac{1}{n+1} C(2n, n)$$

Be careful of calculating factorials in $C(2n, n)$

Other uses of Catalan Numbers

- Number of ways you can triangulate a poligon with $n+2$ sides;
- Number of monotonic paths on an *nxn* grid that do not pass above the diagonal.
- Number of distinct binary trees with n vertices
- Etc...

Class Summary

In this lecture, we discussed challenges in math-focused problems:

- Large Integers and Log Operations;
- Number Theory:
  - Primality Testing and Prime Number Sieve;
  - Factorization;
  - Diaphantyne Equation and Linear Combinations;
- Common Combinatorics Sequences in Programming Challenges;

Next Week we will discuss geometry problems!

Problems for this Week

- Ocean Deep! - Make it Shallow!!
- Sum of Consecutive Prime Numbers
- Divisibility of Factors
- How Many Trees?
- Triangle Counting
- Self-Describing sequence
- Marbles

10176 – Ocean Deep! – Make it Shallow!!

Discussed in the Lecture

Outline

You receive many binary numbers (up to 100 digits), and you must determine if each number is divisible by 131071. Example:

- 0 – YES (0)
- 1010101 – NO (85)

- You can use some bignum library;
- Or you can use mod division too;

Sum of Consecutive Primes

Outline

For a number $N \leq 10000$, determine how many different ways you can write N as a sum of consecutive primes $(p_i + p_{i+1} + \ldots + p_{i+k})$.

- You have to solve for many numbers, but the primes are always the same, so you should pre-calculate the primes.
- Remember that the primes are consecutive, so you should be able to search without backtracking.

Divisibility of Factors

Outline

Given N and d, count how many factors of $N!$ are divisible by d.

- Hint 1: You don't need to calculate $N!$, just the factorization of $N!$
- Hint 2: Think about the relationship between **Prime Factorization** and **Divisibility**

How Many Trees?

Outline

Given a number of nodes with increasing labels, how many **Binary Search Trees** can you make?

- Easy combinatoric problem. Which sequence describes this situation?
- Note that the output might be a large integer.

Triangle Counting

Outline

Given an integer N, how many triangles can you make by choosing three **different** sizes $\leq N$?

Example: $N = 5$, triangles: 2,3,4; 2,4,5; 3,4,5;

- Note that testing all pairs can be too slow for large N
- You should try to find the recurrence on paper first;
  - When you add a new n in the end, how many new triangles can you make with n?

Self Describing Sequence

Outline

In the **self describing sequence**, the value $f(n)$ indicates how many times n appears in the sequence. For example, the first few numbers are:

n :	1	2	3	4	5	6	7	8	9	10	11	12
f(n):	1	2	2	3	3	4	4	4	5	5	5	6

Given a value of $n \leq 2 \times 10^9$, calculate $f(n)$.

- To calculate $f(n)$, is it necessary to calculate every value between $f(1)$ and $f(n-1)$?
- Can we skip some values?

Marbles

Outline

You have n marbles to put in boxes. Box 1 fits n_1 marbles and costs c_1. Box 2 fits n_2 marbles and costs c_2. What is the minimum cost to put all n marbles in boxes?

- This is equivalent to the "candies" problem, but you also have to think about cost.
- Remember, that there are multiple linear combinations that satisfy $n = b_1 n_1 + b_2 n_2$.
- After you calculate one pair b_1, b_2, how do you find other pairs with possibly smaller cost?

About these Slides

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I