# GB21802 - Programming Challenges

## Week 7 - Math Problems

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-06-10,13

Last updated June 10, 2016

## Last Week Results

### Week 6 - Graph II

- From Dusk Until Dawn – 4/31
- Wormholes – 18/31
- Mice and Maze – 11/31
- Degrees of Separation – 9/31
- Avoiding your Boss – 5/31
- Arbitrage – 3/31
- Software Allocation – 4/31
- Sabotage – 1/31
- Little Red Riding Hood – 9/31
- Gopher II – 4/31

- 11 people: 0 problems;
- 8 people: 1-2 problems;
- 7 people: 3-4 problems;
- 3 people: 5-6 problems;
- 2 people: 7+ problems!

## Special Notes

**ASPS for single weighted graphs**

Apparently, this is still an open problem!

- $v$ times BFS: $O(ve + v^2)$
- A paper (2009) claims $O(v^2 \log v)$: `http://waset.org/publications/8870/all-pairs-shortest-paths-problem-for-unweighted-graphs-in-o-n2-log-n-time` (pseudocode included!)
- This is better for dense graphs ($e \to v^2$), but for sparse graphs does not make a difference;

## Math problems in programming Competitions

Math problems have a wide variety of forms, just like Graph problems. However, unlike graph problems, the programming part is easy, and the formulation is hard.

A sample of math topics in programming challenges:

- Ad-hoc: Simulation, Probability;
- Big Num: Simple problems with $n > 1000000000000000000000$
- Number Theory: Primality, Divisibility, Modulo arithmetic;
- Combinatorics: Counting, closed forms, recurrences;

In this lecture, we will just scratch the surface, focusing on examples.
Experience is the best teacher!

## Ad-hoc Maths Problems

**What is ad-hoc?**

ad-hoc means “single purpose”. In other words, you need to improvise a solution useful for only one (or few) problems.

Ad-hoc problems usually require only a bit of elementary maths and a bit of programming;

Common categories:

- Simulation (Brute Force): Execute one or more simple formulas on a set of numbers and report the result;
- Finding patterns/formulas: Similar to simulation, but trying to execute it directly will result in TLE. You have to find a function that summarizes the formula;

## Ad Hoc example: Probability problems

“What is the chance of X happening” is a common ad-hoc problem. The general formula is “prob = events of interest / total events”;

**The dice problem**

If I have *n* dice, what is the chance of rolling a total above *m*?

- Example: For $n = 3$ dice, and $m = 16$, the chance is $10/216$

- 6,6,6
- 6,6,5
- 6,5,6
- 5,6,6
- 6,5,5
- 5,6,5
- 5,5,6
- 4,6,6
- 6,4,6
- 6,6,4

## Ad Hoc example: Probabilty Problems

**The dice problem**

If I have $n$ dice, what is the chance of rolling a total above $m$?

- For $n = 0$, we have only one result: $r = 0$
- For $n = 1$, we have 6 results: $r = \{1, 2, 3, 4, 5, 6\}$
- The result for $n = i$ and $r_{n-1} = k$ is $r_n = k + \{1, 2, 3, 4, 5, 6\}$

- Can you start to see a bottom up DP here?
- With a state table (dice,result), we can count the number of dice combination above a certain value;

## Ad Hoc example: Probability Problems

### Example Code

```
int count(int dice_left, int score_left) {
   if (score_left < 1) return 1;
   if (dice_left == 0) return 0;
   if (result[dice_left][score_left] != -1)
      return result[dice_left][score_left];
   int sum = 0;
   for (int i = 0; i < 6; i++)
      sum += count(dice_left-1, score_left-(i+1))
   result[dice_left][score_left] = sum;
   return sum;
}

prob = count(n,m)/6**n;
```

## Other hints for ad hoc problems

- Calculating $\log_b a$: #import<cmath>; log(a)/log(b);
- Counting Digits: (int) floor(1+log10((double)a));
- $n^{th}$ root of $a$: pow((double) a, 1/(double) n);

## Dealing with big numbers

Many math problems require the handling of numbers much bigger than C++'s long long.

- C++ unsigned int = unsigned long = $2^{32}$ (9-10 digits)
- C++ unsigned long long = $2^{64}$ (19-20 digits)
- Factorial of 21: $>$ 20 digits!

The "DIY" approach is to transform big numbers into strings, and implement digit operations.

The programmer efficient approach is to use the JAVA BigInteger class.

## Java BigInteger Class

Java's BigInteger class handles arbitrarily big numbers: Input, Output, and many operations.

```
import java.util.Scanner;
import java.math.BigInteger;
class Main {
    public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
    int N = sc.nextInt(), F = sc.nextInt();
    if (N == 0 && F == 0) break;
    BigInteger sum = BigInteger.ZERO;
        for (int i = 0; i < N; i++) {
            BigInteger V = sc.nextBigInteger();
            sum = sum.add(V);
}}}}
```

## Java BigInteger algebraic functions

- BigInteger.add(BI): $A + BI$
- BigInteger.subtract(BI): $A - BI$
- BigInteger.multiply(BI): $A * BI$
- BigInteger.divide(BI): $A/BI$
- BigInteger.pow(int): $A^{BI}$
- BigInteger.mod(BI): $A\%BI$
- BigInteger.remainder(BI): $A - A//BI$
- BigInteger.divideAndRemainder(BI): $(A//BI, A - A//BI)$

## Problem Example - 10925 - Krakovia

**Problem Description**

Given a number of costs, and a number of friends, divide the total cost among all friends.

- Super simple problem, but requires BigInteger;
- Use this problem to familiarize yourself with writing JAVA code;

## Java BigInteger superpowers

The class BigInteger implements many other functions that might be useful in mathematical problems:

- .toString(int radix) – Outputs the Big Integer, can also be used to convert base
- .isProbablePrime(int certainty) – probabilistic primality test. The chance of the test being correct is $1 - \frac{1}{2}^{\text{certainty}}$
- .gcd(BI)
- .modPow(BI exponent, BI m)

## Number Theory

The field of Number theory studies the properties of integers and sets. Some problems in field include primality and modular arithmetic.

An understanding of number theory is important to avoid brute force attacks to certain problems, or to pre-process data in large problems.

## Number Theory: Primality

Prime numbers are numbers >= 1 that are only divisible by 1 and themselves. There is a huge use for prime numbers, including cryptography.

- Naive calculation of prime number: For each $i \in 1..N$, test i|N
- Better calculation of prime number: For each $i \in 1..\sqrt{N}$, test i|N
- Even better calculation: For each i is prime $\in 1..\sqrt{N}$, test i|N

  number of primes up to x: $\pi(x) =$ x/logx (prime number theorem)

How can we calculate all the primes between 1 and $\sqrt{N}$ fast?

`https://primes.utm.edu/howmany.html`

## Sieve of Eratosthenes

**Main Idea**

Start with a set of numbers (possible primes) between 2 and $\sqrt{N}$.
For each confirmed prime, remove all multiples of that number from the list.

```
def sieve(k):                     ## Find all primes up to k
   primes = []
   sieve = [1]*(k+1)     ## all numbers start in the list
   sieve[0] = sieve[1] = 0              ## except 0 and 1
   for i in range(k+1):                           ## O(N)
      if (sieve[i] == 1):
         primes.append(i)                ## new prime found
         j = i*i   ## why can i start from i*i, not i*2?
         while (j < k+1):                    ## O(loglogN)
            sieve[j] = 0
            j += i                         ## next multiple
   return primes
```

Complexity the sieve: $O(N \log \log N)$ (Also, this is a kind of DP!)

## Finding Prime Factors

Any natural number $N$ can be expressed as a unique set of prime numbers:

$$N = 1 p_1^{e_1} p_2^{e_2} \dots p_n^{e_n}$$

These are the Prime Factors of $N$. From this set, we can also obtain the set of Factors of $N$ (all numbers $i$ such as $i|N$.

Factorization is a key issue in cryptography

### Very Naive approach

For every $i \in 1..\sqrt{N}$, test $i|N$ and isPrime(i).

Very Costly!

### Naive approach

Calculate a list of primes, for each prime $i$, test if $i|N$.

Will not find all non-prime factors!

## Prime factorization: Divide and conquer approach

**Idea**

The prime factorization of $N$ is equal to the union of $p_i$ and the prime factorization of $N/p_i$, where $p_i$ is the smallest prime factor of $N$.

The set of all factors is composed of all combinations of the set of prime factors (including repetitions).

```
def primefactors(n):
   primes = sieve(int(np.sqrt(n))+1)
   c = 0, i = n, factors = []
   while i > 1:
      if (i%primes[c] == 0):
          i = i/primes[c]
          factors.append(primes[c])
      else:
          c = c+1
   return factors
```

## Working with Prime Factors: 10139 – Factovisors

**Problem description**

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

Factorial of 22 is already bigint! But we can break down these numbers into their factors, which are all $\leq 2^{30}$.

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

Having the factor sets, $m$ divides $n!$ if $F_m \subset F_{n!}$.

Examples:

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

## Modulo Operation

We can use modulo arithmetic to operate on very large numbers without calculating the entire number.

Remember that:

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

## Modulo Operation – UVA 10176, Ocean Deep!

**Problem summary**

Test if a binary number *n* (up to 100000 digits) is divisible by 131071

- The problem wants to know if $n \% 13107 == 0$
- But *n* is too big!
- Use the recurrence in the previous slide to break down each digit to a reasonable value.

## Euclid Algorithm and Extended Euclid Algorithm

- Euclid Algorithm gives us the greatest common divisor $D$ of $a, b$;
- Extended Euclid Algorithm also gives us $x, y$ so that $ax + by = D$;
- Both are extremely simple to code:

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

int x, y;
int egcd(int a, int b) {
  if (a==0)
    {x = 0; y = 1; return b;}         // stop condition
  int d = egcd(b%a, a);
  int tx = x;                         // gcd recurrence
  x = y - (b/a)*tx; y = tx; return d; }   // update x,y
```

## Using EGCD: The Diophantine Equation

**Problem Example (variations of this problem are common)**

You have 839 yen. Xhoco candy costs 25 yen, Yanilla candy costs 18 yen. How many candies can we buy?

The equation $xA + yB = C$ is called the Linear Diophantine Equation. It has infinite solutions if GCD(A,B)|C, but none if it does not.

The first solution $(x_0, y_0)$ can be derived from the extended GCD, and other solutons can be found from: expressed as:

- $x = x_0 + (b/d)n$
- $y = y_0 - (a/d)n$

Where $d$ is GCD(A,B) and $n$ is an integer.

## Using EGCD: The Diophantine Equation

**Problem Example (variations of this problem are common)**

You have 839 yen. Xhoco candy costs 25 yen, Yanilla candy costs 18 yen. How many candies can we buy?

- EGCD gives us: $x = -5, y = 7, d = 1$ or $25(-5) + 18(7) = 1$
- Multiply both sides by 839: $25(-4195) + 18(5873) = 839$
- So: $x_n = -4195 + 18n$ and $y_n = 5873 - 25n$
- We have to find $n$ so that both $x_n, y_n$ are $> 0$.
- $-4195 + 18n \geq 0$ and $5873 - 25n \geq 0$
- $n \geq 4195/18$ and $5873/25 \geq n$
- $4195/18 \leq n \leq 5873/25$
- $233.05 \leq n \leq 234.92$

## Combinatorics problems

Combinatorics is the branch of mathematics concerning the study of countable discrete structures.

Combinatory problems usually involving finding out the recurrence of a set (recursive function that construct the set) or the closed form of a set (formula that calculates the nth element of the set)

Depending on the recurrence or formula, the need of bignums is not uncommon either. Also, often the recurrence can be sped-up by using DP.

## Fibonacci Numbers

$$F(0) = 0, F(1) = 1, F(n > 2) = F(n-1) + F(n-2)$$

**O(N) implementation**

Recursion on F(N) with a DP table (top-down) or (better) iteration on F(n-1), F(n), F(n+1).

**O(1) implementation**

The Fibonacci has a closed form that can be approximated as:

Round($\frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$) where $\phi$ is the golden ratio $\frac{1+\sqrt{5}}{2}$.

## Funny Fibonacci facts

**Zeckendorf's theorem**

Every positive integer can be written in a unique way as a sum of one or more distinct fibonacci numbers which are not consecutive (can find this by greedy method)

**Pisano period**

The last one/two/three/four digits repeat with a period of 60/300/1500/15000

What are the last two digits of the 1211313628th Fibonacci number?

## Binomial Coefficients

${}^{n}C_{k}$ the number of ways that $n$ items can be selected $k$ at a time, also the coefficients of the $(x + y)^n$ polinomial.
A single value of the binomial can be calculated as $\frac{n!}{(n-k)!k!}$. This formula is problematic, however. Not only factorials are very large for small $n$, multiplyign factorials have a good chance of exploding.
Some tricks to avoid using bignum * If k > n-k. then exchange k and n-k. * Try to divide before multiplyign * top down dynamic programming: C(n,0) = C(n,n) = 1 C(n,k) = C(n-1,k-1) + C(n-1,k).

In this case, the recursive formula is more useful than the closed form.

## Catalan Numbers

Cat(n) = 1, 1, 2, 5, 14, 42, 132, 429, 1430

They are calculated as: Cat(n) = $\frac{({}^{2n}C_n)}{(n+1)}$, Cat(0) = 1

Or

Cat(n+1) = $\frac{(2n+2)(2n+1)}{(n+2)(n+1)}$Cat(n)

## Catalan Numbers – Uses

- Number of ways that you can match $n$ parenthesis.
- Number of ways that 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

- Math Problems
- Java's Big Integer class
- Primality
- Modulo arithmetic
- GCD and Diophantine Equations
- Combinatorics

## This Week's Problems

- Division
- What Base Is This
- Divisibility of Factors
- Triangle Counting
- Help My Brother
- Marbles
- Ocean Deep!
- Winning Streak

## To Learn More

Euler Project: Mathematical questions using computers:
`http://projecteuler.net`