GB20602 - Programming Challenges

Week 8 - Mathematics

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: June 11, 2022)

Version 2021.1

Lecture Outline

- Primality testing
- Extended GCD and Diaphantine Equation
- Sequences (Fibonacci, Binomial, Catalan)

Part I – Primality

Primality Testing

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 (reducing) the Complete Search:
  - Search only from 2 to $\sqrt{N}$: $O(\sqrt{N})$
  - Search only 2, and **odd** numbers from 3 to $\sqrt{N}$: $O(\frac{\sqrt{N}}{2})$
  - Search only **prime** numbers from 2 to $\sqrt{N}$: $O(\frac{\sqrt{N}}{\ln(\sqrt{N})})$
- How do we quickly calculate a set of small prime numbers?

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 $\sqrt{10^{10}} \approx 9500$

With a list of small prime numbers, we can test the primality of large numbers quickly.

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

Sieve of Eratosthenes

1. Initialize Vector "sieve" of size $\sqrt{N}$, all TRUE; Loop on Vector.
2. If sieve[i] is TRUE, add i to prime list;
3. Set all multiples of i, sieve[$i * m$] to FALSE;

```
def sieve(k):                   ## Find all primes up to k
   primes = []                  ## List of primes found
   sieve = [1]*(k+1)            ## set all elements of "sieve" to true;
   sieve[0] = sieve[1] = 0      ## 0,1 trivially not primes
   for i in range(k+1):         ## Loops on the sieve;
      if (sieve[i] == 1):       ## Found a new prime
         primes.append(i)       ## Add to prime list
         j = i*i                ## Remove multiples of i (Quiz: 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:

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

Part II – GCD

Modulo Arithmetic

Modulo Arithmetic is a way to operate in very large number without using bignum.

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

Modulo Arithmetic Reminder

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

Example Problem

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.

Modular Inverse

The **Modular Inverse** of a is the number a^{-1} so that $a \times a^{-1} \equiv 1 \mod n$.

How do we find $a^{-1} \mod n$?

- If n is prime, then $b^{-1} \equiv b^{n-2} \mod n$
- If n is not prime, but $\gcd(n, b) = 1$, then $b^{-1} \equiv b^{\Phi(n)-1} \mod n$

We can use the extended GCD to calculate this.

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 extEuclid(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; // GCD, xa + by = d;
}
```

[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: $25 \times 17 + 18 \times 23 = 839$

Extended GCD to calculate modular inverse

Let's calculate x so that $b \times x \equiv 1 \mod n$.

This is equivalent to $bx = 1 + ny \rightarrow bx - ny = 1$, for any y. We feed these values to the extended GCD.

```
int mod(int a, int m) { return ((a%m) + m)%m; }

int modInverse(int b, int m) {
  int x, y;
  int d = extEuclid(b, m, x, y);
  if (d != 1) return -1;  // inverse only exists if gcd(b,m) = 1;

  // b*x + m*y == 1, so apply (mod m) to x to obtain b^-1
  return mod(x, m);
}
```

Part III – Sequences

Sequences

Some programming challenges involves the calculation of well known number sequences.

We usually focus this calculation on two forms:

- **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**;
- **Closed Form**: The closed form of a sequence F calculates F_n **without** using the antecessor values of the sequence.
  - Formula for F(n);

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 ${}^nC_k$. 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...

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