Chapter 5

Mathematics

> *We all use math every day; to predict weather, to tell time, to handle money.*
> *Math is more than formulas or equations; it's logic, it's rationality,*
> *it's using your mind to solve the biggest mysteries we know.*
> — **TV show NUMB3RS**

5.1 Overview and Motivation

The appearance of mathematics-related problems in programming contests is not surprising since Computer Science is deeply rooted in Mathematics. The term 'computer' itself comes from the word 'compute' as computer is built primarily to help human compute numbers. Many interesting real life problems can be modeled as mathematics problems as you will frequently see in this chapter.

Recent ICPC problem sets (especially in Asia) usually contain one or two mathematics problems. Recent IOIs usually do not contain *pure* mathematics tasks, but many tasks do require mathematical insights. This chapter aims to prepare contestants in dealing with many of these mathematics problems.

We are aware that different countries have different emphasis in mathematics training in pre-University education. Thus, some contestants are familiar with the mathematical terms listed in Table 5.1. But for others, these mathematical terms do not ring any bell. Perhaps because the contestant has not learnt it before, or perhaps the term is different in the contestant's native language. In this chapter, we want to make a more level-playing field for the readers by listing as many common mathematical terminologies, definitions, problems, and algorithms that frequently appear in programming contests.

Arithmetic Progression	Geometric Progression	Polynomial
Algebra	Logarithm/Power	BigInteger
Combinatorics	Fibonacci	Golden Ratio
Binet's formula	Zeckendorf's theorem	Catalan Numbers
Factorial	Derangement	Binomial Coefficients
Number Theory	Prime Number	Sieve of Eratosthenes
Modified Sieve	Miller-Rabin's	Euler Phi
Greatest Common Divisor	Lowest Common Multiple	Extended Euclid
Linear Diophantine Equation	Cycle-Finding	Probability Theory
Game Theory	Zero-Sum Game	Decision Tree
Perfect Play	Minimax	Nim Game

Table 5.1: List of *some* mathematical terms discussed in this chapter

5.2 Ad Hoc Mathematics Problems

We start this chapter with something light: The Ad Hoc mathematics problems. These are programming contest problems that require no more than basic programming skills and some fundamental mathematics. As there are still too many problems in this category, we further divide them into sub-categories, as shown below. These problems are not placed in Section 1.4 as they are Ad Hoc problems with mathematical flavor. You can actually jump from Section 1.4 to this section if you prefer to do so. But remember that many of these problems are the easier ones. To do well in the actual programming contests, contestants must also master *the other sections* of this chapter.

- The Simpler Ones—just a few lines of code per problem to boost confidence. These problems are for those who have not solved any mathematics-related problems before.

- Mathematical Simulation (Brute Force)
  The solutions to these problems can be obtained by simulating the mathematical process. Usually, the solution requires some form of loops. Example: Given a set S of $1M$ random integers and an integer X. How many integers in S are less than X? Answer: Brute force, scan all the $1M$ integers and count how many of them are less than X. This is slightly faster than sorting the $1M$ integers first. See Section 3.2 if you need to review various (iterative) Complete Search/brute force techniques. Some mathematical problems solvable with brute force approach are also listed in that Section 3.2.

- Finding Pattern or Formula
  These problems require the problem solver to read the problem description carefully to spot the pattern or simplified formula. Attacking them directly will usually result in TLE verdict. The actual solutions are usually short and do not require loops or recursions. Example: Let set S be an infinite set of *square integers* sorted in increasing order: $\{1, 4, 9, 16, 25, \ldots\}$. Given an integer X ($1 \leq X \leq 10^{17}$), determine how many integers in S are less than X? Answer: $\lfloor \sqrt{X-1} \rfloor$.

- Grid
  These problems involve grid manipulation. The grid can be complex, but the grid follow some primitive rules. The 'trivial' 1D/2D grid are not classified here. The solution usually depends on the problem solver's creativity on finding the patterns to manipulate/navigate the grid or in converting the given one into a simpler one.

- Number Systems or Sequences
  Some Ad Hoc mathematics problems involve definitions of existing (or fictional) Number Systems or Sequences and our task is to produce either the number (sequence) within some range or the n-th one, verify if the given number (sequence) is valid according to definition, etc. Usually, following the problem description carefully is the key to solving the problem. But some harder problems require us to simplify the formula first. Some well-known examples are:

  1. Fibonacci numbers (Section 5.4.1): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  2. Factorial (Section 5.5.3): 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...
  3. Derangement (Section 9.8): 1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, ...
  4. Catalan numbers (Section 5.4.3): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...

5. Arithmetic progression series: a_1, $(a_1 + d)$, $(a_1 + 2 \times d)$, $(a_1 + 3 \times d)$, ..., e.g. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...that starts with $a_1 = 1$ and with difference of $d = 1$ between consecutive terms. The sum of the first n terms of this arithmetic progression series $S_n = \frac{n}{2} \times (2 \times a_1 + (n - 1) \times d)$.
6. Geometric progression series, e.g. a_1, $a_1 \times r$, $a_1 \times r^2$, $a_1 \times r^3$, ..., e.g. 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...that starts with $a_1 = 1$ and with common ratio $r = 2$ between consecutive terms. The sum of the first n terms of this geometric progression series $S_n = a \times \frac{1-r^n}{1-r}$.

- Logarithm, Exponentiation, Power
  These problems involve the (clever) usage of `log()` and/or `exp()` function.
  Some of the important ones are shown in the written exercises below.
- Polynomial
  These problems involve polynomial evaluation, derivation, multiplication, division, etc. We can represent a polynomial by storing the coefficients of the polynomial's terms sorted by their powers (usually in descending order). The operations on polynomial usually require some careful usage of loops.
- Base Number Variants
  These are the mathematical problems involving base number, but they are not the *standard* conversion problem that can be easily solved with Java BigInteger technique (see Section 5.3).
- Just Ad Hoc
  These are other mathematics-related problems that cannot be classified yet as one of the sub-categories above.

We suggest that the readers—especially those who are new with mathematics problems—kick start their training programme on mathematics problems by solving at least 2 or 3 problems *from each sub-category*, especially the ones that we highlight as **must try ***.

Exercise 5.2.1: What should we use in C/C++/Java to compute `log`$_b$`(a)` (base b)?

Exercise 5.2.2: What will be returned by `(int)floor(1 + log10((double)a))`?

Exercise 5.2.3: How to compute $\sqrt[n]{a}$ (the n-th root of a) in C/C++/Java?

Exercise 5.2.4*: Study the (Ruffini-)Horner's method for finding the roots of a polynomial equation $f(x) = 0$!

Exercise 5.2.5*: Given $1 < a < 10, 1 \leq n \leq 100000$, show how to compute the value of $1 \times a + 2 \times a^2 + 3 \times a^3 + \ldots + n \times a^n$ efficiently, i.e. in $O(\log n)$!

Programming Exercises related to Ad Hoc Mathematics problems:

- The Simpler Ones
  1. UVa 10055 - Hashmat the Brave Warrior (absolute function; use long long)
  2. UVa 10071 - Back to High School ... (super simple: outputs $2 \times v \times t$)
  3. UVa 10281 - Average Speed (distance = speed × time elapsed)

4. UVa 10469 - To Carry or not to Carry (super simple if you use `xor`)
5. **UVa 10773 - Back to Intermediate ... *** (several tricky cases)
6. UVa 11614 - Etruscan Warriors Never ... (find roots of a quadratic equation)
7. **UVa 11723 - Numbering Road *** (simple math)
8. UVa 11805 - Bafana Bafana (very simple $O(1)$ formula exists)
9. **UVa 11875 - Brick Game *** (get median of a sorted input)
10. *UVa 12149 - Feynman* (finding the pattern; square numbers)
11. *UVa 12502 - Three Families* (must understand the 'wording trick' first)

- Mathematical Simulation (Brute Force), Easier
    1. UVa 00100 - The 3n + 1 problem (do as asked; note that j can be $< i$)
    2. UVa 00371 - Ackermann Functions (similar to UVa 100)
    3. **UVa 00382 - Perfection *** (do trial division)
    4. UVa 00834 - Continued Fractions (do as asked)
    5. UVa 00906 - Rational Neighbor (compute c, from $d = 1$ until $\frac{a}{b} < \frac{c}{d}$)
    6. **UVa 01225 - Digit Counting *** (LA 3996, Danang07, N is small)
    7. UVa 10035 - Primary Arithmetic (count the number of carry operations)
    8. **UVa 10346 - Peter's Smoke *** (interesting simulation problem)
    9. UVa 10370 - Above Average (compute average, see how many are above it)
    10. UVa 10783 - Odd Sum (input range is very small, just brute force it)
    11. UVa 10879 - Code Refactoring (just use brute force)
    12. UVa 11150 - Cola (similar to UVa 10346, be careful with boundary cases!)
    13. UVa 11247 - Income Tax Hazard (brute force around the answer to be safe)
    14. UVa 11313 - Gourmet Games (similar to UVa 10346)
    15. UVa 11689 - Soda Surpler (similar to UVa 10346)
    16. UVa 11877 - The Coco-Cola Store (similar to UVa 10346)
    17. UVa 11934 - Magic Formula (just do plain brute-force)
    18. *UVa 12290 - Counting Game* (no '-1' in the answer)
    19. *UVa 12527 - Different Digits* (try all, check repeated digits)
- Mathematical Simulation (Brute Force), Harder
    1. *UVa 00493 - Rational Spiral* (simulate the spiral process)
    2. *UVa 00550 - Multiplying by Rotation* (rotamult property; try one by one starting from 1 digit)
    3. **UVa 00616 - Coconuts, Revisited *** (brute force up to $\sqrt{n}$, get pattern)
    4. *UVa 00697 - Jack and Jill* (requires some output formatting and basic knowledge about Physics)
    5. UVa 00846 - Steps (uses the sum of arithmetic progression formula)
    6. *UVa 10025 - The ? 1 ? 2 ? ...* (simplify the formula first, iterative)
    7. *UVa 10257 - Dick and Jane* (we can brute force the integer ages of spot, puff, and yertle; need some mathematical insights)
    8. *UVa 10624 - Super Number* (backtracking with divisibility check)
    9. **UVa 11130 - Billiard bounces *** (use billiard table reflection technique: mirror the billiard table to the right (and/or top) so that we will only deal with one straight line instead of bouncing lines)
    10. ***UVa 11254 - Consecutive Integers **** (use sum of arithmetic progression: $n = \frac{r}{2} \times (2 \times a + r - 1)$ or $a = (2 \times n + r - r^2)/(2 \times r)$; as n is given, brute force all values of r from $\sqrt{2n}$ down to 1, stop at the first valid a)
    11. UVa 11968 - In The Airport (average; fabs; if ties, choose the smaller one!)

    Also see some mathematical problems in Section 3.2.

- Finding Pattern or Formula, Easier
    1. UVa 10014 - Simple calculations (derive the required formula)
    2. UVa 10170 - The Hotel with Infinite ... (one liner formula exists)
    3. UVa 10499 - The Land of Justice (simple formula exists)
    4. UVa 10696 - f91 (very simple formula simplification)
    5. ***UVa 10751 - Chessboard **** (trival for $N = 1$ and $N = 2$; derive the formula first for $N > 2$; hint: use diagonal as much as possible)
    6. **UVa 10940 - Throwing Cards Away II *** (find pattern with brute force)
    7. UVa 11202 - The least possible effort (consider symmetry and flip)
    8. ***UVa 12004 - Bubble Sort **** (try small n; get the pattern; use long long)
    9. *UVa 12027 - Very Big Perfect Square* (sqrt trick)
- Finding Pattern or Formula, Harder
    1. *UVa 00651 - Deck* (use the given sample I/O to derive the simple formula)
    2. UVa 00913 - Joana and The Odd ... (derive the short formulas)
    3. **UVa 10161 - Ant on a Chessboard *** (involves sqrt, ceil...)
    4. *UVa 10493 - Cats, with or without Hats* (tree, derive the formula)
    5. UVa 10509 - R U Kidding Mr. ... (there are only three different cases)
    6. UVa 10666 - The Eurocup is here (analyze the binary representation of X)
    7. UVa 10693 - Traffic Volume (derive the short Physics formula)
    8. *UVa 10710 - Chinese Shuffle* (the formula is a bit hard to derive; involving modPow; see Section 5.3 or Section 9.21)
    9. *UVa 10882 - Koerner's Pub* (inclusion-exclusion principle)
    10. UVa 10970 - Big Chocolate (direct formula exists, or use DP)
    11. UVa 10994 - Simple Addition (formula simplification)
    12. **UVa 11231 - Black and White Painting *** (there is $O(1)$ formula)
    13. *UVa 11246 - K-Multiple Free Set* (derive the formula)
    14. UVa 11296 - Counting Solutions to an ... (simple formula exists)
    15. *UVa 11298 - Dissecting a Hexagon* (simple maths; derive the pattern first)
    16. *UVa 11387 - The 3-Regular Graph* (impossible for odd n or when $n = 2$; if n is a multiple of 4, consider complete graph K_4; if $n = 6 + k \times 4$, consider one 3-Regular component of 6 vertices and the rest are K_4 as in previous case)
    17. *UVa 11393 - Tri-Isomorphism* (draw several small K_n, derive the pattern)
    18. ***UVa 11718 - Fantasy of a Summation **** (convert loops to a closed form formula, use `modPow` to compute the results, see Section 5.3 and 9.21)
- Grid
    1. **UVa 00264 - Count on Cantor *** (math, grid, pattern)
    2. UVa 00808 - Bee Breeding (math, grid, similar to UVa 10182)
    3. UVa 00880 - Cantor Fractions (math, grid, similar to UVa 264)
    4. **UVa 10182 - Bee Maja *** (math, grid)
    5. **UVa 10233 - Dermuba Triangle *** (the number of items in row forms arithmetic progression series; use hypot)
    6. UVa 10620 - A Flea on a Chessboard (just simulate the jumps)
    7. UVa 10642 - Can You Solve It? (the reverse of UVa 264)
    8. *UVa 10964 - Strange Planet* (convert the coordinates to (x, y), then this problem is just about finding Euclidean distance between two coordinates)

9. *SPOJ 3944 - Bee Walk* (a grid problem)

- Number Systems or Sequences
    1. UVa 00136 - Ugly Numbers (use similar technique as UVa 443)
    2. UVa 00138 - Street Numbers (arithmetic progression formula, precalculated)
    3. UVa 00413 - Up and Down Sequences (simulate; array manipulation)
    4. **UVa 00443 - Humble Numbers *** (try all $2^i \times 3^j \times 5^k \times 7^l$, sort)
    5. UVa 00640 - Self Numbers (DP bottom up, generate the numbers, flag once)
    6. UVa 00694 - The Collatz Sequence (similar to UVa 100)
    7. UVa 00962 - Taxicab Numbers (pre-calculate the answer)
    8. UVa 00974 - Kaprekar Numbers (there are not that many Kaprekar numbers)
    9. UVa 10006 - Carmichael Numbers (non prime which has ≥ 3 prime factors)
    10. **UVa 10042 - Smith Numbers *** (prime factorization, sum the digits)
    11. *UVa 10049 - Self-describing Sequence* (enough to get past > 2G by storing only the first $700K$ numbers of the Self-describing sequence)
    12. UVa 10101 - Bangla Numbers (follow the problem description carefully)
    13. **UVa 10408 - Farey Sequences *** (first, generate `(i, j)` pairs such that `gcd(i, j) = 1`, then sort)
    14. UVa 10930 - A-Sequence (ad-hoc, follow the rules given in description)
    15. *UVa 11028 - Sum of Product* (this is 'dartboard sequence')
    16. UVa 11063 - B2 Sequences (see if a number is repeated, be careful with -ve)
    17. UVa 11461 - Square Numbers (answer is $\sqrt{b} - \sqrt{a-1}$)
    18. UVa 11660 - Look-and-Say sequences (simulate, break after j-th character)
    19. UVa 11970 - Lucky Numbers (square numbers, divisibility check, bf)

- Logarithm, Exponentiation, Power
    1. UVa 00107 - The Cat in the Hat (use logarithm, power)
    2. UVa 00113 - Power Of Cryptography (use `exp(ln(x) × y)`)
    3. UVa 00474 - Heads Tails Probability (this is just a `log` & `pow` exercise)
    4. UVa 00545 - Heads (use logarithm, power, similar to UVa 474)
    5. **UVa 00701 - Archaelogist's Dilemma *** (use log to count # of digits)
    6. *UVa 01185 - BigNumber* (number of digits of factorial, use logarithm to solve it; $\log(n!) = \log(n \times (n-1) \ldots \times 1) = \log(n) + \log(n-1) + \ldots + \log(1)$)
    7. **UVa 10916 - Factstone Benchmark *** (use logarithm, power)
    8. *UVa 11384 - Help is needed for Dexter* (finding the smallest power of two greater than n, can be solved easily using $ceil(eps + \log_2(n))$)
    9. *UVa 11556 - Best Compression Ever* (related to power of two, use long long)
    10. UVa 11636 - Hello World (uses logarithm)
    11. UVa 11666 - Logarithms (find the formula!)
    12. UVa 11714 - Blind Sorting (use decision tree model to find min and second min; eventually the solution only involves logarithm)
    13. **UVa 11847 - Cut the Silver Bar *** ($O(1)$ math formula exists: $\lfloor \log_2(n) \rfloor$)
    14. UVa 11986 - Save from Radiation ($\log_2(N+1)$; manual check for precision)
    15. *UVa 12416 - Excessive Space Remover* (the answer is $\log_2$ of the max consecutive spaces in a line)

- Polynomial
    1. *UVa 00126 - The Errant Physicist* (polynomial multiplication and tedious output formatting)
    2. UVa 00392 - Polynomial Showdown (follow the orders: output formatting)
    3. **UVa 00498 - Polly the Polynomial *** (polynomial evaluation)
    4. *UVa 10215 - The Largest/Smallest Box* (two trivial cases for smallest; derive the formula for largest which involves quadratic equation)
    5. **UVa 10268 - 498' *** (polynomial derivation; Horner's rule)
    6. UVa 10302 - Summation of Polynomials (use long double)
    7. *UVa 10326 - The Polynomial Equation* (given roots of the polynomial, reconstruct the polynomial; formatting)
    8. **UVa 10586 - Polynomial Remains *** (division; manipulate coefficients)
    9. UVa 10719 - Quotient Polynomial (polynomial division and remainder)
    10. *UVa 11692 - Rain Fall* (use algebraic manipulation to derive a quadratic equation; solve it; special case when $H < L$)
- Base Number Variants
    1. **UVa 00377 - Cowculations *** (base 4 operations)
    2. **UVa 00575 - Skew Binary *** (base modification)
    3. UVa 00636 - Squares (base number conversion up to base 99; Java BigInteger cannot be used as it is MAX_RADIX is limited to 36)
    4. UVa 10093 - An Easy Problem (try all)
    5. UVa 10677 - Base Equality (try all from r2 to r1)
    6. **UVa 10931 - Parity *** (convert decimal to binary, count number of '1's)
    7. UVa 11005 - Cheapest Base (try all possible bases from 2 to 36)
    8. UVa 11121 - Base -2 (search for the term 'negabinary')
    9. *UVa 11398 - The Base-1 Number System* (just follow the new rules)
    10. *UVa 12602 - Nice Licence Plates* (simple base conversion)
    11. *SPOJ 0739 - The Moronic Cowmpouter* (find the representation in base -2)
    12. IOI 2011 - Alphabets (practice task; use the more space-efficient base 26)
- Just Ad Hoc
    1. UVa 00276 - Egyptian Multiplication (multiplication of Egyptian hieroglyphs)
    2. UVa 00496 - Simply Subsets (set manipulation)
    3. *UVa 00613 - Numbers That Count* (analyze the number; determine the type; similar spirit with the cycle finding problem in Section 5.7)
    4. **UVa 10137 - The Trip *** (be careful with precision error)
    5. UVa 10190 - Divide, But Not Quite ... (simulate the process)
    6. *UVa 11055 - Homogeneous Square* (not classic, observation needed to avoid brute-force solution)
    7. *UVa 11241 - Humidex* (the hardest case is computing Dew point given temperature and Humidex; derive it with Algebra)
    8. **UVa 11526 - H(n) *** (brute force up to $\sqrt{n}$, find the pattern, avoid TLE)
    9. UVa 11715 - Car (physics simulation)
    10. UVa 11816 - HST (simple math, precision required)
    11. ***UVa 12036 - Stable Grid **** (use pigeon hole principle)

5.3 Java BigInteger Class

5.3.1 Basic Features

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 (Section 5.5.5) or modulo arithmetic techniques (Section 5.5.8), we have no choice but to resort to BigInteger (a.k.a bignum) libraries. An example: Compute the *precise value* of 25! (the factorial of 25). The result is 15,511,210,043,330,985,984,000,000 (26 digits). This is clearly too large to fit in 64-bit C/C++ `unsigned long long` (or Java `long`).

One way to implement BigInteger library is to store the BigInteger as a (long) string[1]. For example we can store 10^{21} inside a string `num1` = "1,000,000,000,000,000,000,000" without any problem whereas this is already overflow in a 64-bit C/C++ `unsigned long long` (or Java `long`). Then, for common mathematical operations, we can use a kind of digit by digit operations to process the two BigInteger operands. For example with `num2` = "173", we have `num1 + num2` as:

```
num1        =   1,000,000,000,000,000,000,000
num2        =                             173
              ------------------------------- +
num1 + num2 =   1,000,000,000,000,000,000,173
```

We can also compute `num1 * num2` as:

```
num1        =   1,000,000,000,000,000,000,000
num2        =                             173
               ------------------------------ *
                3,000,000,000,000,000,000,000
               70,000,000,000,000,000,000,00
              100,000,000,000,000,000,000,0
              ------------------------------- +
num1 * num2 = 173,000,000,000,000,000,000,000
```

Addition and subtraction are the two simpler operations in BigInteger. Multiplication takes a bit more programming job, as seen in the example above. Implementing efficient division and raising an integer to a certain power are more complicated. Anyway, coding these library routines in C/C++ under stressful contest environment can be a buggy affair, even if we can bring notes containing such C/C++ library in ICPC[2]. Fortunately, Java has a BigInteger class that we can use for this purpose. As of 24 May 2013, the C++ STL does not have such feature thus it is a good idea to use Java for BigInteger problems.

The Java BigInteger (we abbreviate it as `BI`) class supports the following basic integer operations: addition—`add(BI)`, subtraction—`subtract(BI)`, multiplication—`multiply(BI)`, power—`pow(int exponent)`, division—`divide(BI)`, remainder—`remainder(BI)`, modulo—`mod(BI)` (different to `remainder(BI)`), division and remainder—`divideAndRemainder(BI)`, and a few other interesting functions discussed later. All are just 'one liner'.

[1] Actually, a primitive data type also stores numbers as *limited string of bits* in computer memory. For example a 32-bit `int` data type stores an integer as 32 bits of binary string. BigInteger technique is just a generalization of this technique that uses decimal form (base 10) and longer string of digits. Note: Java BigInteger class likely uses a more efficient method than the one shown in this section.

[2] Good news for IOI contestants. IOI tasks usually do not require contestants to deal with BigInteger.

However, we need to remark that all BigInteger operations are *inherently slower* than the same operations on standard 32/64-bit integer data types. Rule of Thumb: If you can use another algorithm that only requires built-in integer data type to solve your mathematical problem, then use it instead of resorting to BigInteger.

For those who are new to Java BigInteger class, we provide the following short Java code, which is the solution for UVa 10925 - Krakovia. This problem requires BigInteger addition (to sum N large bills) and division (to divide the large sum to F friends). Observe how short and clear the code is compared to if you have to write your own BigInteger routines.

```
import java.util.Scanner;                          // inside package java.util
import java.math.BigInteger;                       // inside package java.math

class Main {                                        /* 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 has this constant
      for (int i = 0; i < N; i++) {                  // sum the N large bills
        BigInteger V = sc.nextBigInteger(); // for reading 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
  }
}
```

Source code: `ch5_01_UVa10925.java`

Exercise 5.3.1.1: Compute the last non zero digit of 25!; Can we use built-in data type?

Exercise 5.3.1.2: Check if 25! is divisible by 9317; Can we use built-in data type?

5.3.2 Bonus Features

Java BigInteger class has a few more bonus features that can be useful during programming contests—in terms of shortening the code length—compared to if we have to write these functions ourselves[3]. Java BigInteger class happens to have a built-in base number converter: The class's constructor and function `toString(int radix)`, a very good (but probabilistic) prime testing function `isProbablePrime(int certainty)`, a GCD routine `gcd(BI)`, and a modular arithmetic function `modPow(BI exponent, BI m)`. Among these bonus features, the base number converter is the most useful one, followed by the prime testing function. These bonus features are shown with four example problems from UVa online judge.

[3] A note for pure C/C++ programmers: It is good to be a *multi*-lingual programmer by switching to Java whenever it is more beneficial to do so.

Base Number Conversion

See an example below for UVa 10551 - 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[4] 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 (radix) as shown below:

```
class Main {                                  /* UVa 10551 - Basic Remains */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
      int b = sc.nextInt();
      if (b == 0) break;                        // special class's constructor!
      BigInteger p = new BigInteger(sc.next(), b);  // the second parameter
      BigInteger m = new BigInteger(sc.next(), b);            // is the base
      System.out.println((p.mod(m)).toString(b)); // can output in any base
} } }
```

Source code: ch5_02_UVa10551.java

(Probabilistic) Prime Testing

Later in Section 5.5.1, we will discuss Sieve of Eratosthenes algorithm and a deterministic prime testing algorithm that is good enough for many contest problems. However, you have to type in a few lines of C/C++/Java code to do that. If you just need to check whether a single (or at most, several[5]) and usually large integer is a prime, e.g. UVa 10235 below, there is an alternative and shorter approach with function `isProbablePrime` in Java BigInteger—a probabilistic prime testing function based on Miller-Rabin's algorithm [44, 55]. There is an important parameter of this function: `certainty`. If this function returns true, then the probability that the tested BigInteger is a prime exceeds $1 - \frac{1}{2}^{certainty}$. For typical contest problems, `certainty = 10` should be enough as $1 - (\frac{1}{2})^{10} = 0.9990234375$ is ≈ 1.0. Note that using larger value of `certainty` obviously decreases the probability of WA but doing so slows down your program and thus increases the risk of TLE. Please attempt **Exercise 5.3.2.3*** to convince yourself.

```
class Main {                                   /* UVa 10235 - Simply Emirp */
  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);
```

[4] For example, to convert 132 in base 8 (octal) into base 2 (binary), we can use base 10 (decimal) as the intermediate step: $(132)_8$ is $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$ and $(90)_{10}$ is $90 \rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0(1) = (1011010)_2$ (that is, divide by 2 until 0, then read the remainders from backwards).

[5] Note that if your aim is to generate a list of the first few million prime numbers, the Sieve of Eratosthenes algorithm shown in Section 5.5.1 should run faster than a few million calls of this: `isProbablePrime` function.

```
      BigInteger BRN = BigInteger.valueOf(RN);
      System.out.printf("%d is ", N);
      if (!BN.isProbablePrime(10)) // certainty 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.");
} } }
```

Source code: ch5_03_UVa10235.java

Greatest Common Divisor (GCD)

See an example below for UVa 10814 - Simplifying Fractions. We are asked to reduce a large fraction to its simplest form by dividing both numerator and denominator with their GCD. Also see Section 5.5.2 for more details about GCD.

```
class Main {                        /* UVa 10814 - Simplifying Fractions */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int N = sc.nextInt();
    while (N-- > 0) {   // unlike in C/C++, we have to use > 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));
} } }
```

Source code: ch5_04_UVa10814.java

Modulo Arithmetic

See an example below for UVa 1230 (LA 4104) - MODEX that computes $x^y (\bmod\ n)$. Also see Section 5.5.8 and 9.21 to see how this `modPow` function is actually computed.

```
class Main {                               /* UVa 1230 (LA 4104) - MODEX */
  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 converts
      BigInteger y = BigInteger.valueOf(sc.nextInt());    // simple integer
      BigInteger n = BigInteger.valueOf(sc.nextInt());   // into BigInteger
      System.out.println(x.modPow(y, n));           // it's in the library!
} } }
```

Source code: ch5_05_UVa1230.java

Exercise 5.3.2.1: Try solving UVa 389 using the Java BigInteger technique presented here. Can you pass the time limit? If no, is there a (slightly) better technique?

Exercise 5.3.2.2*: As of 24 May 2013, programming contest problems involving *arbitrary precision* decimal numbers (not necessarily integers) are still rare. So far, we have only identified two problems in UVa online judge that require such feature: UVa 10464 and UVa 11821. Try solving these two problems using another library: Java `BigDecimal` class! Explore: `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`.

Exercise 5.3.2.3*: Write a Java program to *empirically* determine the lowest value of parameter `certainty` so that our program can run fast and there is no *composite number* between `[2..10M]`—a typical contest problem range—is accidentally reported as prime by `isProbablePrime(certainty)`! As `isProbablePrime` uses a probabilistic algorithm, you have to repeat your experiment several times for each `certainty` value. Is `certainty = 5` good enough? What about `certainty = 10`? What about `certainty = 1000`?

Exercise 5.3.2.4*: Study and implement the Miller Rabin's algorithm (see [44, 55]) in case you have to implement it in C/C++!

Programming Exercises related to BigInteger **NOT**[6] mentioned elsewhere:

- Basic Features
  1. UVa 00424 - Integer Inquiry (BigInteger addition)
  2. UVa 00465 - Overflow (BigInteger add/multiply, compare with $2^{31} - 1$)
  3. UVa 00619 - Numerically Speaking (BigInteger)
  4. **UVa 00713 - Adding Reversed ... *** (BigInteger + StringBuffer reverse())
  5. UVa 00748 - Exponentiation (BigInteger exponentiation)
  6. UVa 01226 - Numerical surprises (LA 3997, Danang07, mod operation)
  7. UVa 10013 - Super long sums (BigInteger addition)
  8. UVa 10083 - Division (BigInteger + number theory)
  9. UVa 10106 - Product (BigInteger multiplication)
  10. UVa 10198 - Counting (recurrences, BigInteger)
  11. *UVa 10430 - Dear GOD* (BigInteger, derive formula first)
  12. *UVa 10433 - Automorphic Numbers* (BigInteger, pow, substract, mod)
  13. UVa 10494 - If We Were a Child Again (BigInteger division)
  14. UVa 10519 - Really Strange (recurrences, BigInteger)
  15. **UVa 10523 - Very Easy *** (BigInteger addition, multiplication, and power)
  16. UVa 10669 - Three powers (BigInteger is for 3^n, binary rep of set!)
  17. UVa 10925 - Krakovia (BigInteger addition and division)
  18. *UVa 10992 - The Ghost of Programmers* (input size is up to 50 digits)
  19. UVa 11448 - Who said crisis? (BigInteger subtraction)
  20. *UVa 11664 - Langton's Ant* (simple simulation involving BigInteger)
  21. UVa 11830 - Contract revision (use BigInteger string representation)
  22. **UVa 11879 - Multiple of 17 *** (BigInteger mod, divide, subtract, equals)
  23. *UVa 12143 - Stopping Doom's Day* (LA 4209, Dhaka08, formula simplification—the hard part; use BigInteger—the easy part)
  24. *UVa 12459 - Bees' ancestors* (draw the ancestor tree to see the pattern)

[6] It worth mentioning that there are *many* other programming exercises in other sections of this chapter (and also in another chapters) that also use BigInteger technique.

- Bonus Feature: Base Number Conversion
    1. UVa 00290 - Palindroms ←→ ... (also involving palindrome)
    2. **UVa 00343 - What Base Is This? *** (try all possible pair of bases)
    3. UVa 00355 - The Bases Are Loaded (basic base number conversion)
    4. **UVa 00389 - Basically Speaking *** (use Java `Integer` class)
    5. UVa 00446 - Kibbles 'n' Bits 'n' Bits ... (base number conversion)
    6. UVa 10473 - Simple Base Conversion (Decimal to Hexadecimal and vice versa; if you use C/C++, you can use `strtol`)
    7. **UVa 10551 - Basic Remains *** (also involving BigInteger mod)
    8. UVa 11185 - Ternary (Decimal to base 3)
    9. *UVa 11952 - Arithmetic* (check base 2 to 18 only; special case for base 1)
- Bonus Feature: Primality Testing
    1. *UVa 00960 - Gaussian Primes* (there is a number theory behind this)
    2. **UVa 01210 - Sum of Consecutive ... *** (LA 3399, Tokyo05, simple)
    3. **UVa 10235 - Simply Emirp *** (case analysis: not prime/prime/emirp; emirp is defined as prime number that if reversed is still a prime number)
    4. UVa 10924 - Prime Words (check if sum of letter values is a prime)
    5. **UVa 11287 - Pseudoprime Numbers *** (output yes if `!isPrime(p) + a.modPow(p, p) = a`; use Java BigInteger)
    6. *UVa 12542 - Prime Substring* (HatYai12, brute force, use isProbablePrime to test primality)
- Bonus Feature: Others
    1. **UVa 01230 - MODEX *** (LA 4104, Singapore07, BigInteger modPow)
    2. *UVa 10023 - Square root* (code Newton's method with BigInteger)
    3. UVa 10193 - All You Need Is Love (convert two binary strings S1 and S2 to decimal and check see if gcd(s1, s2) > 1)
    4. UVa 10464 - Big Big Real Numbers (solvable with Java BigDecimal class)
    5. **UVa 10814 - Simplifying Fractions *** (BigInteger gcd)
    6. **UVa 11821 - High-Precision Number *** (Java BigDecimal class)

Profile of Algorithm Inventors

Gary Lee Miller is a professor of Computer Science at Carnegie Mellon University. He is the initial inventor of Miller-Rabin primality test algorithm.

Michael Oser Rabin (born 1931) is an Israeli computer scientist. He improved Miller's idea and invented the Miller-Rabin primality test algorithm. Together with Richard Manning Karp, he also invented Rabin-Karp's string matching algorithm.

5.4 Combinatorics

Combinatorics is a branch of *discrete mathematics*[7] concerning the study of **countable** discrete structures. In programming contests, problems involving combinatorics are usually titled 'How Many [Object]', 'Count [Object]', etc, although some problem authors choose to hide this fact from their problem titles. The solution code is usually *short*, but finding the (usually recursive) formula takes some mathematical brilliance and also patience.

In ICPC[8], if such a problem exists in the given problem set, ask one team member who is strong in mathematics to derive the formula whereas the other two concentrate on *other* problems. Quickly code the usually short formula once it is obtained—interrupting whoever is currently using the computer. It is also a good idea to memorize/study the common ones like the Fibonacci-related formulas (see Section 5.4.1), Binomial Coefficients (see Section 5.4.2), and Catalan Numbers (see Section 5.4.3).

Some of these combinatorics formulas may yield overlapping subproblems that entails the need of using Dynamic Programming technique (see Section 3.5). Some computation values can also be large that entails the need of using BigInteger technique (see Section 5.3).

5.4.1 Fibonacci Numbers

Leonardo *Fibonacci*'s numbers are defined as $fib(0) = 0$, $fib(1) = 1$, and for $n \geq 2$, $fib(n) = fib(n-1) + fib(n-2)$. This generates the following familiar pattern: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. This pattern sometimes appears in contest problems which do not mention the term 'Fibonacci' at all, like in some problems in the list of programming exercises in this section (e.g. UVa 900, 10334, 10450, 10497, 10862, etc).

We usually derive the Fibonacci numbers with a 'trivial' $O(n)$ DP technique and not implement the given recurrence directly (as it is very slow). However, the $O(n)$ DP solution is *not* the fastest for all cases. Later in Section 9.21, we will show how to compute the n-th Fibonacci number (where n is large) in $O(\log n)$ time using the efficient matrix power. As a note, there is an $O(1)$ *approximation* technique to get the *n-th* Fibonacci number. We can compute the closest integer of $(\phi^n - (-\phi)^{-n})/\sqrt{5}$ (Binet's formula) where ϕ (golden ratio) is $((1+\sqrt{5})/2) \approx 1.618$. However this is not so accurate for large Fibonacci numbers.

Fibonacci numbers grow very fast and some problems involving Fibonacci have to be solved using Java BigInteger library (see Section 5.3).

Fibonacci numbers have many interesting properties. One of them is the **Zeckendorf's theorem**: Every positive integer can be written in a unique way as a sum of one or more distinct Fibonacci numbers such that the sum does not include any two consecutive Fibonacci numbers. For any given positive integer, a representation that satisfies Zeckendorf's theorem can be found by using a *Greedy* algorithm: Choose the largest possible Fibonacci number at each step. For example: 100 = 89 + 8 + 3; 77 = 55 + 21 + 1, 18 = 13 + 5, etc.

Another property is the **Pisano Period** where the last one/last two/last three/last four digit(s) of a Fibonacci number repeats with a period of 60/300/1500/15000, respectively.

Exercise 5.4.1.1: Try $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ on small n and see if this Binet's formula really produces $fib(7) = 13$, $fib(9) = 34$, $fib(11) = 89$. Now, write a simple program to find out the first value of n such that the actual value of $fib(n)$ differs from the result of this approximation formula? Is that n big enough for typical usage in programming contests?

[7] Discrete mathematics is a study of structures that are discrete (e.g. integers {0, 1, 2, . . . }, graphs/trees (vertices and edges), logic (true/false)) rather than continuous (e.g. real numbers).

[8] Note that pure combinatorics problem is rarely used in an IOI task (it can be part of a bigger task).

5.4.2 Binomial Coefficients

Another classical combinatorics problem is in finding the *coefficients* of the algebraic expansion of powers of a binomial[9]. These coefficients are also the number of ways that n items can be taken k at a time, usually written as $C(n,k)$ or ${}^nC_k$. For example, $(x+y)^3 = \mathbf{1}x^3 + \mathbf{3}x^2y + \mathbf{3}xy^2 + \mathbf{1}y^3$. The **{1, 3, 3, 1}** are the binomial coefficients of $n = 3$ with $k = \{0,1,2,3\}$ respectively. Or in other words, the number of ways that $n = 3$ items can be taken $k = \{0,1,2,3\}$ item at a time are $\{1,3,3,1\}$ different ways, respectively.

We can compute a single value of $C(n,k)$ with this formula: $C(n,k) = \frac{n!}{(n-k)! \times k!}$. However, computing $C(n,k)$ can be a challenge when n and/or k are large. There are several tricks like: Making k smaller (if $k > n-k$, then we set $k = n-k$) because ${}^nC_k = {}^n C_{(n-k)}$; during intermediate computations, we divide the numbers first before multiply it with the next number; or use BigInteger technique (last resort as BigInteger operations are slow).

If we have to compute *many but not all* values of $C(n,k)$ for different n and k, it is better to use top-down Dynamic Programming. We can write $C(n,k)$ as shown below and use a 2D memo table to avoid re-computations.

$C(n,0) = C(n,n) = 1$ // base cases.
$C(n,k) = C(n-1,k-1) + C(n-1,k)$ // take or ignore an item, $n > k > 0$.

However, if we have to compute *all* values of $C(n,k)$ from $n = 0$ up to a certain value of n, then it may be beneficial to construct the *Pascal's Triangle*, a triangular array of binomial coefficients. The leftmost and rightmost entries at each row are always 1. The inner values are the sum of two values directly above it, as shown for row $n = 4$ below. This is essentially the bottom-up version of the DP solution above.

```
n = 0           1
n = 1         1   1
n = 2       1   2   1
n = 3     1   3   3   1  <- as shown above
           \ / \ / \ /
n = 4   1   4   6   4   1 ... and so on
```

Exercise 5.4.2.1: A frequently used k for $C(n,k)$ is $k = 2$. Show that $C(n,2) = O(n^2)$.

5.4.3 Catalan Numbers

First, let's define the n-th Catalan number—written using binomial coefficients notation ${}^nC_k$ above—as: $Cat(n) = ({}^{(2\times n)}C_n)/(n+1)$; $Cat(0) = 1$. We will see its purpose below.

If we are asked to compute the values of $Cat(n)$ for *several* values of n, it may be better to compute the values using bottom-up Dynamic Programming. If we know $Cat(n)$, we can compute $Cat(n+1)$ by manipulating the formula like shown below.

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

$Cat(n+1) = \frac{(2\times(n+1))!}{(n+1)! \times (n+1)! \times ((n+1)+1)} = \frac{(2n+2)\times(2n+1)\times 2n!}{(n+1)\times n! \times (n+1) \times n! \times (n+2)} = \frac{(2n+2)\times(2n+1)\times ...[2n!]}{(n+2)\times(n+1)\times ...[n!\times n! \times (n+1)]}$.

Therefore, $Cat(n+1) = \frac{(2n+2)\times(2n+1)}{(n+2)\times(n+1)} \times Cat(n)$.

Alternatively, we can set $m = n+1$ so that we have: $Cat(m) = \frac{2m\times(2m-1)}{(m+1)\times m} \times Cat(m-1)$.

[9] Binomial is a special case of polynomial that only has two terms.

Catalan numbers are found in various combinatorial problems. Here, we list down some of the more interesting ones (there are several others, see **Exercise 5.4.4.8***). All examples below use $n = 3$ and $Cat(3) = ({}^{(2\times3)}C_3)/(3+1) = ({}^{6}C_3)/4 = 20/4 = 5$.

1. $Cat(n)$ counts the number of distinct binary trees with n vertices, e.g. for $n = 3$:

```
    *    *    *    *    *
   /    /    / \    \    \
  *    *    *   *    *    *
 /      \           /      \
*        *         *        *
```

2. $Cat(n)$ counts the number of expressions containing n pairs of parentheses which are correctly matched, e.g. for $n = 3$, we have: ()()(), ()(()), (())(), ((())), and (()()).

3. $Cat(n)$ counts the number of different ways $n + 1$ factors can be completely parenthesized, e.g. for $n = 3$ and $3 + 1 = 4$ factors: {a, b, c, d}, we have: (ab)(cd), a(b(cd)), ((ab)c)d, (a(bc))(d), and a((bc)d).

4. $Cat(n)$ counts the number of ways a convex polygon (see Section 7.3) of $n + 2$ sides can be triangulated. See Figure 5.1, left.

5. $Cat(n)$ counts the number of monotonic paths along the edges of an $n \times n$ grid, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. See Figure 5.1, right and also see Section 4.7.1.

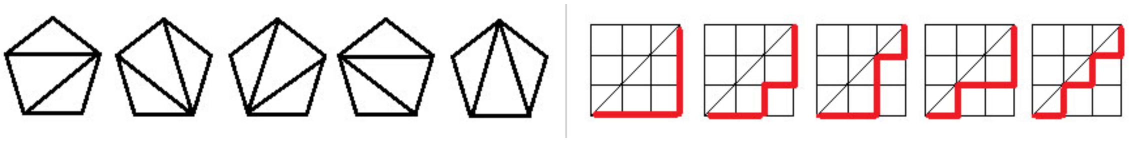

Figure 5.1: Left: Triangulation of a Convex Polygon, Right: Monotonic Paths

5.4.4 Remarks about Combinatorics in Programming Contests

There are many other combinatorial problems that may also appear in programming contests, but they are not as frequent as Fibonacci numbers, Binomial Coefficients, or Catalan numbers. Some of the more interesting ones are listed in Section 9.8.

In *online* programming contests where contestant can access the Internet, there is one more trick that may be useful. First, generate the output for small instances and then search for that sequence at OEIS (The On-Line Encyclopedia of Integer Sequences) hosted at `http://oeis.org/`. If you are lucky, OEIS can tell you the name of the sequence and/or the required general formula for the larger instances.

There are still many other counting principles and formulas, too many to be discussed in this book. We close this section by giving some written exercises to test/further improve your combinatorics skills. Note: The problems listed in this section constitute ≈ 15% of the entire problems in this chapter.

Exercise 5.4.4.1: Count the number of different possible outcomes if you roll two 6-sided dices and flip two 2-sided coins?

Exercise 5.4.4.2: How many ways to form a three digits number from {0, 1, 2, ..., 9} and each digit can only be used once? Note that 0 cannot be used as the leading digit.

Exercise 5.4.4.3: Suppose you have a 6-letters word 'FACTOR'. If we take 3-letters from this word 'FACTOR', we may have another valid English word, like 'ACT', 'CAT', 'ROT', etc. What is the maximum number of different 3-letters word that can be formed with the letters from 'FACTOR'? You do not have to care whether the 3-letters word is a valid English word or not.

Exercise 5.4.4.4: Suppose you have a 5-letters word 'BOBBY'. If we rearrange the letters, we can get another word, like 'BBBOY', 'YOBBB', etc. How many *different* permutations are possible?

Exercise 5.4.4.5: Solve UVa 11401 - Triangle Counting! This problem has a short description: "Given n rods of length 1, 2, ..., n, pick any 3 of them and build a triangle. How many distinct triangles can you make (consider triangle inequality, see Section 7.2)? $(3 \leq n \leq 1M)$ ". Note that, two triangles will be considered different if they have at least one pair of arms with different lengths. If you are lucky, you may spend only a few minutes to spot the pattern. Otherwise, this problem may end up unsolved by the time contest is over—which is a bad sign for your team.

Exercise 5.4.4.6*: Study the following terms: Burnside's Lemma, Stirling Numbers.

Exercise 5.4.4.7*: Which one is the hardest to factorize (see Section 5.5.4) assuming that n is an arbitrary large integer: $fib(n)$, $C(n,k)$ (assume that $k = n/2$), or $Cat(n)$? Why?

Exercise 5.4.4.8*: Catalan numbers $Cat(n)$ appear in some other interesting problems other than the ones shown in this section. Investigate!

Other Programming Exercises related to Combinatorics:

- Fibonacci Numbers
  1. UVa 00495 - Fibonacci Freeze (very easy with Java BigInteger)
  2. UVa 00580 - Critical Mass (related to *Tribonacci* series; Tribonacci numbers are the generalization of Fibonacci numbers; it is defined by $T_1 = 1$, $T_2 = 1$, $T_3 = 2$, and $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ for $n \geq 4$)
  3. **UVa 00763 - Fibinary Numbers *** (Zeckendorf representation, greedy, use Java BigInteger)
  4. UVa 00900 - Brick Wall Patterns (combinatorics, the pattern ≈ Fibonacci)
  5. UVa 00948 - Fibonaccimal Base (Zeckendorf representation, greedy)
  6. UVa 01258 - Nowhere Money (LA 4721, Phuket09, Fibonacci variant, Zeckendorf representation, greedy)
  7. UVa 10183 - How many Fibs? (get the number of Fibonaccis when generating them; BigInteger)
  8. **UVa 10334 - Ray Through Glasses *** (combinatorics, Java BigInteger)
  9. UVa 10450 - World Cup Noise (combinatorics, the pattern ≈ Fibonacci)
  10. UVa 10497 - Sweet Child Make Trouble (the pattern ≈ Fibonacci)

11. UVa 10579 - Fibonacci Numbers (very easy with Java BigInteger)
12. **UVa 10689 - Yet Another Number ... *** (easy if you know Pisano (a.k.a Fibonacci) period)
13. UVa 10862 - Connect the Cable Wires (the pattern ends up ≈ Fibonacci)
14. UVa 11000 - Bee (combinatorics, the pattern is similar to Fibonacci)
15. *UVa 11089 - Fi-binary Number* (the list of Fi-binary Numbers follow the Zeckendorf's theorem)
16. UVa 11161 - Help My Brother (II) (Fibonacci + median)
17. UVa 11780 - Miles 2 Km (the background problem is Fibonacci numbers)

- Binomial Coefficients:
  1. UVa 00326 - Extrapolation using a ... (difference table)
  2. UVa 00369 - Combinations (be careful with overflow issue)
  3. UVa 00485 - Pascal Triangle of Death (binomial coefficients + BigInteger)
  4. UVa 00530 - Binomial Showdown (work with doubles; optimize computation)
  5. *UVa 00911 - Multinomial Coefficients* (there is a formula for this, $result = n!/(z_1! \times z_2! \times z_3! \times ... \times zk!)$)
  6. UVa 10105 - Polynomial Coefficients ($n!/(n_1! \times n_2! \times ... \times n_k!)$; however, the derivation is complex)
  7. **UVa 10219 - Find the Ways *** (count the length of ${}^nC_k$; BigInteger)
  8. UVa 10375 - Choose and Divide (the main task is to avoid overflow)
  9. *UVa 10532 - Combination, Once Again* (modified binomial coefficient)
  10. ***UVa 10541 - Stripe **** (a good combinatorics problem; compute how many white cells are there via $Nwhite = N$ - sum of all K integers; imagine we have one more white cell at the very front, we can now determine the answer by placing black stripes after K out of $Nwhite + 1$ whites, or ${}_{Nwhite+1}C_K$ (use Java BigInteger); however, if $K > Nwhite + 1$ then the answer is 0)
  11. **UVa 11955 - Binomial Theorem *** (pure application; DP)
- Catalan Numbers
  1. **UVa 00991 - Safe Salutations *** (Catalan Numbers)
  2. **UVa 10007 - Count the Trees *** (answer is $Cat(n) \times n!$; BigInteger)
  3. UVa 10223 - How Many Nodes? (Precalculate the answers as there are only 19 Catalan Numbers $< 2^{32} - 1$)
  4. UVa 10303 - How Many Trees (generate $Cat(n)$ as shown in this section, use Java BigInteger)
  5. ***UVa 10312 - Expression Bracketing **** (the number of binary bracketing can be counted by $Cat(n)$; the total number of bracketing can be computed using *Super-Catalan* numbers)
  6. *UVa 10643 - Facing Problems With ...* ($Cat(n)$ is part of a bigger problem)
- Others, Easier
  1. UVa 11115 - Uncle Jack (N^D, use Java BigInteger)
  2. **UVa 11310 - Delivery Debacle *** (requires DP: let `dp[i]` be the number of ways the cakes can be packed for a box `2 × i`. Note that it is possible to use two 'L shaped' cakes to form a 2×3 shape)
  3. **UVa 11401 - Triangle Counting *** (spot the pattern, coding is easy)
  4. UVa 11480 - Jimmy's Balls (try all r, but simpler formula exists)
  5. **UVa 11597 - Spanning Subtree *** (uses knowledge of graph theory, the answer is very trivial)

6. UVa 11609 - Teams ($N \times 2^{N-1}$, use Java BigInteger for the modPow part)
7. *UVa 12463 - Little Nephew* (double socks & shoes to simplify the problem)

- Others, Harder
  1. *UVa 01224 - Tile Code* (derive formula from small instances)
  2. UVa 10079 - Pizza Cutting (derive the one liner formula)
  3. UVa 10359 - Tiling (derive the formula, use Java BigInteger)
  4. UVa 10733 - The Colored Cubes (Burnside's lemma)
  5. **UVa 10784 - Diagonal *** (the number of diagonals in n-gon $= n*(n-3)/2$, use it to derive the solution)
  6. UVa 10790 - How Many Points of ... (uses arithmetic progression formula)
  7. UVa 10918 - Tri Tiling (there are two related recurrences here)
  8. **UVa 11069 - A Graph Problem *** (use Dynamic Programming)
  9. UVa 11204 - Musical Instruments (only first choice matters)
  10. *UVa 11270 - Tiling Dominoes* (sequence A004003 in OEIS)
  11. ***UVa 11538 - Chess Queen **** (count along rows, columns, and diagonals)
  12. UVa 11554 - Hapless Hedonism (similar to UVa 11401)
  13. *UVa 12022 - Ordering T-shirts* (number of ways n competitors can rank in a competition, allowing for the possibility of ties, see `http://oeis.org/A000670`)

Profile of Algorithm Inventors

Leonardo Fibonacci (also known as **Leonardo Pisano**) (1170-1250) was an Italian mathematician. He published a book titled 'Liber Abaci' (Book of Abacus/Calculation) in which he discussed a problem involving the growth of a population of *rabbits* based on idealized assumptions. The solution was a sequence of numbers now known as the Fibonacci numbers.

Edouard Zeckendorf (1901-1983) was a Belgian mathematician. He is best known for his work on Fibonacci numbers and in particular for proving Zeckendorf's theorem.

Jacques Philippe Marie Binet (1786-1856) was a French mathematician. He made significant contributions to number theory. Binet's formula expressing Fibonacci numbers in closed form is named in his honor, although the same result was known earlier.

Blaise Pascal (1623-1662) was a French mathematician. One of his famous invention discussed in this book is the Pascal's triangle of binomial coefficients.

Eugène Charles Catalan (1814-1894) was a French and Belgian mathematician. He is the one who introduced the Catalan numbers to solve a combinatorial problem.

Eratosthenes of Cyrene ($\approx$ 300-200 years BC) was a Greek mathematician. He invented geography, did measurements of the circumference of earth, and invented a simple algorithm to generate prime numbers which we discuss in this book.

Leonhard Euler (1707-1783) was a Swiss mathematician. His inventions mentioned in this book are the Euler totient (Phi) function and the Euler tour/path (Graph).

Christian Goldbach (1690-1764) was a German mathematician. He is remembered today for Goldbach's conjecture that he discussed extensively with Leonhard Euler.

Diophantus of Alexandria ($\approx$ 200-300 AD) was an Alexandrian Greek mathematician. He did a lot of study in algebra. One of his works is the Linear Diophantine Equations.

5.5 Number Theory

Mastering as many topics as possible in the field of *number theory* is important as some mathematics problems become easy (or easier) if you know the theory behind the problems. Otherwise, either a plain brute force attack leads to a TLE response or you simply cannot work with the given input as it is too large without some pre-processing.

5.5.1 Prime Numbers

A natural number starting from 2: $\{2, 3, 4, 5, 6, 7, \ldots\}$ is considered as a **prime** if it is only divisible by 1 or itself. The first and the only even prime is 2. The next prime numbers are: 3, 5, 7, 11, 13, 17, 19, 23, 29, ..., and infinitely many more primes (proof in [56]). There are 25 primes in range `[0..100]`, 168 primes in `[0..1000]`, 1000 primes in `[0..7919]`, 1229 primes in `[0..10000]`, etc. Some large prime numbers are[10] $104729, 1299709, 15485863, 179424673, 2147483647, 32416190071, 112272535095293, 48112959837082048697$, etc.

Prime number is an important topic in number theory and the source for many programming problems[11]. In this section, we will discuss algorithms involving prime numbers.

Optimized Prime Testing Function

The first algorithm presented in this section is for testing whether a given natural number N is prime, i.e. `bool isPrime(N)`. The most naïve version is to test by definition, i.e. test if N is divisible by *divisor* $\in$ `[2..`N`-1]`. This works, but runs in $O(N)$—in terms of number of divisions. This is not the best way and there are several possible improvements.

The first major improvement is to test if N is divisible by a *divisor* $\in$ `[2..`$\sqrt{N}$`]`, i.e. we stop when the *divisor* is greater than $\sqrt{N}$. Reason: If N is divisible by d, then $N = d \times \frac{N}{d}$. If $\frac{N}{d}$ is smaller than d, then $\frac{N}{d}$ or a prime factor of $\frac{N}{d}$ would have divided N earlier. Therefore d and $\frac{N}{d}$ cannot *both* be greater than $\sqrt{N}$. This improvement is $O(\sqrt{N})$ which is already much faster than the previous version, but can still be improved to be twice as fast.

The second improvement is to test if N is divisible by *divisor* $\in$ `[3, 5, 7,.., `$\sqrt{N}$`]`, i.e. we only test odd numbers up to $\sqrt{N}$. This is because there is only one even prime number, i.e. number 2, which can be tested separately. This is $O(\sqrt{N}/2)$, which is also $O(\sqrt{N})$.

The third improvement[12] which is already good enough[13] for contest problems is to test if N is divisible by *prime divisors* $\leq \sqrt{N}$. This is because if a prime number X cannot divide N, then there is no point testing whether multiples of X divide N or not. This is faster than $O(\sqrt{N})$ which is about $O(\#primes \leq \sqrt{N})$. For example, there are 500 odd numbers in `[1..`$\sqrt{10^6}$`]`, but there are only 168 primes in the same range. Prime number theorem [56] says that the number of primes less than or equal to M—denoted by $\pi(M)$—is bounded by $O(M/(\ln(M) - 1))$. Therefore, the complexity of this prime testing function is about $O(\sqrt{N}/\ln(\sqrt{N}))$. The code is shown in the next discussion below.

Sieve of Eratosthenes: Generating List of Prime Numbers

If we want to generate a list of prime numbers between range `[0..`N`]`, there is a better algorithm than testing each number in the range whether it is a prime number or not. The

[10] Having a list of large random prime numbers can be good for testing as these are the numbers that are hard for algorithms like the prime testing or prime factoring algorithms.

[11] In real life, large primes are used in cryptography because it is hard to factor a number xy into $x \times y$ when both are **relatively prime** (also known as **coprime**).

[12] This is a bit recursive—testing whether a number is a prime by using another (smaller) prime number. But the reason should be obvious after reading the next section.

[13] Also see Section 5.3.2 for the Miller-Rabin's probabilistic prime testing with Java BigInteger class.

algorithm is called 'Sieve of *Eratosthenes*' invented by Eratosthenes of Alexandria.

First, this Sieve algorithm sets all numbers in the range to be 'probably prime' but set numbers 0 and 1 to be not prime. Then, it takes 2 as prime and crosses out all multiples[14] of 2 starting from $2 \times 2 = 4$, 6, 8, 10, ... until the multiple is greater than N. Then it takes the next non-crossed number 3 as a prime and crosses out all multiples of 3 starting from $3 \times 3 = 9$, 12, 15, ... Then it takes 5 and crosses out all multiples of 5 starting from $5 \times 5 = 25$, 30, 35, ... And so on .... After that, whatever left uncrossed within the range `[0..`N`]` are primes. This algorithm does approximately ($N\times$ (1/2 + 1/3 + 1/5 + 1/7 + ... + 1/last prime in range $\leq N$)) operations. Using 'sum of reciprocals of primes up to n', we end up with the time complexity of *roughly* $O(N \log \log N)$.

Since generating a list of primes $\leq 10K$ using the sieve is fast (our code below can go up to 10^7 under contest setting), we opt to use sieve for smaller primes and reserve optimized prime testing function for larger primes—see previous discussion. The code is as follows:

```
#include <bitset>       // compact STL for Sieve, better than vector<bool>!
ll _sieve_size;                  // ll is defined as: typedef long long ll;
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 list of 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);       // add this prime to the list of primes
} }                                      // 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

// inside int main()
  sieve(10000000);                  // can go up to 10^7 (need few seconds)
  printf("%d\n", isPrime(2147483647));                   // 10-digits prime
  printf("%d\n", isPrime(136117223861LL));   // not a prime, 104729*1299709
```

Source code: `ch5_06_primes.cpp/java`

5.5.2 Greatest Common Divisor & Least Common Multiple

The Greatest Common Divisor (GCD) of two integers: a, b denoted by $gcd(a, b)$, is the largest positive integer d such that $d \mid a$ and $d \mid b$ where $x \mid y$ means that x divides y. Example of GCD: $gcd(4, 8) = 4$, $gcd(6, 9) = 3$, $gcd(20, 12) = 4$. One practical usage of GCD is to simplify fractions (see UVa 10814 in Section 5.3.2), e.g. $\frac{6}{9} = \frac{6/gcd(6,9)}{9/gcd(6,9)} = \frac{6/3}{9/3} = \frac{2}{3}$.

[14] Common implementation is to start from 2 × i instead of i × i, but the difference is not that much.

Finding the GCD of two integers is an easy task with an effective Divide and Conquer *Euclid* algorithm [56, 7] which can be implemented as a one liner code (see below). Thus finding the GCD of two integers is usually not the main issue in a Math-related contest problem, but just part of a bigger solution.

The GCD is closely related to Least (or Lowest) Common Multiple (LCM). The LCM of two integers (a, b) denoted by $lcm(a, b)$, is defined as the smallest positive integer l such that $a \mid l$ and $b \mid l$. Example of LCM: $lcm(4, 8) = 8$, $lcm(6, 9) = 18$, $lcm(20, 12) = 60$. It has been shown (see [56]) that: $lcm(a, b) = a \times b/gcd(a, b)$. This can also be implemented as a one liner code (see below).

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

The GCD of more than 2 numbers, e.g. $gcd(a, b, c)$ is equal to $gcd(a, gcd(b, c))$, etc, and similarly for LCM. Both GCD and LCM algorithms run in $O(\log_{10} n)$, where $n = max(a, b)$.

Exercise 5.5.2.1: The formula for LCM is $lcm(a, b) = a \times b/gcd(a, b)$ but why do we use $a \times (b/gcd(a, b))$ instead? Hint: Try $a = 1000000000$ and $b = 8$ using 32-bit signed integers.

5.5.3 Factorial

Factorial of n, i.e. $n!$ or $fac(n)$ is defined as 1 if $n = 0$ and $n \times fac(n - 1)$ if $n > 0$. However, it is usually more convenient to work with the iterative version, i.e. $fac(n) = 2 \times 3 \times \ldots \times (n - 1) \times n$ (loop from 2 to n, skipping 1). The value of $fac(n)$ grows very fast. We can still use C/C++ `long long` (Java `long`) for up to $fac(20)$. Beyond that, we may need to use either Java BigInteger library for precise but slow computation (see Section 5.3), work with the prime factors of a factorial (see Section 5.5.5), or get the intermediate and final results modulo a smaller number (see Section 5.5.8).

5.5.4 Finding Prime Factors with Optimized Trial Divisions

In number theory, we know that a prime number N only have 1 and itself as factors but a **composite** number N, i.e. the non-primes, can be written uniquely as a multiplication of its prime factors. That is, prime numbers are multiplicative building blocks of integers (the fundamental theorem of arithmetic). For example, $N = 1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$ (the latter form is called **prime-power factorization**).

A naïve algorithm generates a list of primes (e.g. with sieve) and check which prime(s) can actually divide the integer N—without changing N. This can be improved!

A better algorithm utilizes a kind of Divide and Conquer spirit. An integer N can be expressed as: $N = PF \times N'$, where PF is a prime factor and N' is another number which is N/PF—i.e. we can reduce the size of N by taking out its prime factor PF. We can keep doing this until eventually $N' = 1$. To speed up the process even further, we utilize the divisibility property that there is no divisor greater than $\sqrt{N}$, so we only repeat the process of finding prime factors until $PF \leq \sqrt{N}$. Stopping at $\sqrt{N}$ entails a special case: If (current $PF)^2 > N$ and N is still not 1, then N is the last prime factor. The code below takes in an integer N and returns the list of prime factors.

In the worst case—when N is prime, this prime factoring algorithm with trial division requires testing all smaller primes up to $\sqrt{N}$, mathematically denoted as $O(\pi(\sqrt{N})) = O(\sqrt{N}/ln\sqrt{N})$—see the example of factoring a large composite number 136117223861 into two large prime factors: 104729×1299709 in the code below. However, if given composite numbers with lots of small prime factors, this algorithm is reasonably fast—see 142391208960 which is $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$.

```
vi primeFactors(ll N) {     // remember: vi is vector<int>, ll is long long
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx]; // primes has been populated by sieve
  while (PF * PF <= N) {             // stop at sqrt(N); N can get smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }    // remove PF
    PF = primes[++PF_idx];                           // only consider primes!
  }
  if (N != 1) factors.push_back(N);         // special case if N is a prime
  return factors;     // if N does not fit in 32-bit integer and is a prime
}                   // then ‘factors’ will have to be changed to vector<ll>

// inside int main(), assuming sieve(1000000) has been called before
  vi r = primeFactors(2147483647);        // slowest, 2147483647 is a prime
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("> %d\n", *i);

  r = primeFactors(136117223861LL);                 // slow, 104729*1299709
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("# %d\n", *i);

  r = primeFactors(142391208960LL);         // faster, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("! %d\n", *i);
```

Exercise 5.5.4.1: Examine the given code above. What is/are the value(s) of `N` that can break this piece of code? You can assume that `vi ‘primes’` contains list of prime numbers with the largest prime of 9999991 (slightly below 10 million).

Exercise 5.5.4.2: John Pollard invented a better algorithm for integer factorization. Study and implement Pollard's rho algorithm (both the original and the improvement by Richard P. Brent) [52, 3]!

5.5.5 Working with Prime Factors

Other than using the Java BigInteger technique (see Section 5.3) which is ‘slow’, we can work with the *intermediate computations* of large integers *accurately* by working with the *prime factors* of the integers instead of the actual integers themselves. Therefore, for some non-trivial number theoretic problems, we have to work with the prime factors of the input integers even if the main problem is not really about prime numbers. After all, prime factors are the building blocks of integers. Let's see the case study below.

UVa 10139 - Factovisors can be abridged as follow: “Does m divides $n!$? ($0 \leq n, m \leq 2^{31} - 1$)”. In the earlier Section 5.5.3, we mentioned that with *built-in data types*, the largest factorial that we can still compute precisely is 20!. In Section 5.3, we show that we can compute large integers with Java BigInteger technique. However, it is *very slow* to precisely compute the exact value of $n!$ for large n. The solution for this problem is to work with the prime factors of both $n!$ and m. We factorize m to its prime factors and see if it has ‘support’ in $n!$. For example, when $n = 6$, we have 6! expressed as prime power factorization:

$6! = 2 \times 3 \times 4 \times 5 \times 6 = 2 \times 3 \times (2^2) \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$. For 6!, $m_1 = 9 = 3^2$ has support—see that 3^2 is part of 6!, thus $m_1 = 9$ divides 6!. However, $m_2 = 27 = 3^3$ has *no* support—see that the largest power of 3 in 6! is just 3^2, thus $m_2 = 27$ does *not* divide 6!.

Exercise 5.5.5.1: Determine what is the GCD and LCM of $(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2)$?

5.5.6 Functions Involving Prime Factors

There are other well-known number theoretic functions involving prime factors shown below. All variants have similar time complexity with the basic prime factoring via trial division above. Interested readers can further explore Chapter 7: "Multiplicative Functions" of [56].

1. `numPF(N)`: Count the number of *prime factors* of `N`

   A simple tweak of the trial division algorithm to find prime factors shown earlier.

```
ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

2. `numDiffPF(N)`: Count the number of *different* prime factors of `N`

3. `sumPF(N)`: *Sum* the prime factors of `N`

4. `numDiv(N)`: Count the number of *divisors* of `N`

   Divisor of integer N is defined as an integer that divides N without leaving a remainder. If a number $N = a^i \times b^j \times \ldots \times c^k$, then N has $(i+1) \times (j+1) \times \ldots \times (k+1)$ divisors. For example: $N = 60 = 2^2 \times 3^1 \times 5^1$ has $(2+1) \times (1+1) \times (1+1) = 3 \times 2 \times 2 = 12$ divisors. The 12 divisors are: $\{1, \underline{\mathbf{2}}, \underline{\mathbf{3}}, 4, \underline{\mathbf{5}}, 6, 10, 12, 15, 20, 30, 60\}$. The prime factors of 12 are **highlighted**. See that N has more divisors than prime factors.

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

5. `sumDiv(N)`: *Sum* the divisors of `N`

   In the previous example, $N = 60$ has 12 divisors. The sum of these divisors is 168. This can be computed via prime factors too. If a number $N = a^i \times b^j \times \ldots \times c^k$, then the sum of divisors of N is $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times ... \times \frac{c^{k+1}-1}{c-1}$. Let's try. $N = 60 = 2^2 \times 3^1 \times 5^1$, `sumDiv(60)` $= \frac{2^{2+1}-1}{2-1} \times \frac{3^{1+1}-1}{3-1} \times \frac{5^{1+1}-1}{5-1} = \frac{7 \times 8 \times 24}{1 \times 2 \times 4} = 168$.

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;   // start from ans = 1
  while (PF * PF <= N) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);  // last
  return ans;
}
```

6. `EulerPhi(N)`: Count the number of positive integers $< N$ that are relatively prime to N. Recall: Two integers a and b are said to be relatively prime (or coprime) if $gcd(a, b) = 1$, e.g. 25 and 42. A naïve algorithm to count the number of positive integers $< N$ that are relatively prime to N starts with `counter = 0`, iterates through $i \in$ `[1..`N`-1]`, and increases the `counter` if $gcd(i, N) = 1$. This is slow for large N.

   A better algorithm is the Euler's Phi (Totient) function $\varphi(N) = N \times \prod_{PF}(1 - \frac{1}{PF})$, where PF is prime factor of N.

   For example $N = 36 = 2^2 \times 3^2$. $\varphi(36) = 36 \times (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) = 12$. Those 12 positive integers that are relatively prime to 36 are {1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35}.

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

Exercise 5.5.6.1: Implement `numDiffPF(N)` and `sumPF(N)`!
Hint: Both are similar to `numPF(N)`.

5.5.7 Modified Sieve

If the number of different prime factors has to be determined for *many* (or a *range* of) integers, then there is a better solution than calling `numDiffPF(N)` as shown in Section 5.5.6 above *many times*. The better solution is the modified sieve algorithm. Instead of finding the prime factors and then calculate the required values, we start from the prime numbers and modify the values of their multiples. The short modified sieve code is shown below:

```
memset(numDiffPF, 0, sizeof numDiffPF);
for (int i = 2; i < MAX_N; i++)
  if (numDiffPF[i] == 0)                            // i is a prime number
    for (int j = i; j < MAX_N; j += i)
      numDiffPF[j]++;              // increase the values of multiples of i
```

This modified sieve algorithm should be preferred over individual calls to `numDiffPF(N)` if the range is large. However, if we just need to compute the number of different prime factors for a single but large integer N, it may be faster to just use `numDiffPF(N)`.

Exercise 5.5.7.1: The function `EulerPhi(N)` shown in Section 5.5.6 can also be re-written using modified sieve. Please write the required code!

Exercise 5.5.7.2*: Can we write the modified sieve code for the other functions listed in Section 5.5.6 above (i.e. other than `numDiffPF(N)` and `EulerPhi(N)`) without increasing the time complexity of sieve? If we can, write the required code! If we cannot, explain why!

5.5.8 Modulo Arithmetic

Some mathematical computations in programming problems can end up having very large positive (or very small negative) intermediate/final results that are beyond the range of the largest built-in integer data type (currently the 64-bit `long long` in C++ or `long` in Java). In Section 5.3, we have shown a way to compute big integers precisely. In Section 5.5.5, we have shown another way to work with big integers via its prime factors. For some other problems, we are only interested with the result *modulo* a (usually prime) number so that the intermediate/final results always fits inside built-in integer data type. In this subsection, we discuss these types of problems.

For example in UVa 10176 - Ocean Deep! Make it shallow!!, we are asked to convert a long binary number (up to 100 digits) to decimal. A quick calculation shows that the largest possible number is $2^{100} - 1$ which is beyond the range of a 64-bit integer. However, the problem only ask if the result is divisible by 131071 (which is a prime number). So what we need to do is to convert binary to decimal digit by digit, while performing modulo 131071 operation to the intermediate result. If the final result is 0, then the *actual number in binary* (which we never compute in its entirety), is divisible by 131071.

Exercise 5.5.8.1: Which statements are valid? Note: '%' is a symbol of modulo operation.
1). `(a + b - c) % s = ((a % s) + (b % s) - (c % s) + s) % s`
2). `(a * b) % s = (a % s) * (b % s)`
3). `(a * b) % s = ((a % s) * (b % s)) % s`
4). `(a / b) % s = ((a % s) / (b % s)) % s`
5). (a^{b}) % s = ((a$^{b/2}$ % s) * (a$^{b/2}$ % s)) % s; assume that b is even.

5.5.9 Extended Euclid: Solving Linear Diophantine Equation

Motivating problem: Suppose a housewife buys apples and oranges with cost of 8.39 SGD. An apple is 25 cents. An orange is 18 cents. How many of each fruit does she buy?

This problem can be modeled as a linear equation with two variables: $25x + 18y = 839$. Since we know that both x and y must be integers, this linear equation is called the Linear *Diophantine* Equation. We can solve Linear Diophantine Equation with two variables even if we only have one equation! The solution is as follow:

Let a and b be integers with $d = gcd(a, b)$. The equation $ax + by = c$ has no integral solutions if $d \mid c$ is not true. But if $d \mid c$, then there are infinitely many integral solutions. The first solution (x_0, y_0) can be found using the *Extended Euclid* algorithm shown below and the rest can be derived from $x = x_0 + (b/d)n$, $y = y_0 - (a/d)n$, where n is an integer. Programming contest problems will usually have additional constraints to make the output finite (and unique).

```
// store x, y, and d as global variables
void extendedEuclid(int a, int b) {
  if (b == 0) { x = 1; y = 0; d = a; return; }                // base case
  extendedEuclid(b, a % b);                     // similar as the original gcd
  int x1 = y;
  int y1 = x - (a / b) * y;
  x = x1;
  y = y1;
}
```

Using `extendedEuclid`, we can solve the motivating problem shown earlier above:
The Linear Diophantine Equation with two variables $25x + 18y = 839$.

$a = 25, b = 18$
$extendedEuclid(25, 18)$ produces $x = -5, y = 7, d = 1$; or $25 \times (-5) + 18 \times 7 = 1$.

Multiply the left and right hand side of the equation above by $839/gcd(25, 18) = 839$:
$25 \times (-4195) + 18 \times 5873 = 839$.
Thus $x = -4195 + (18/1)n$ and $y = 5873 - (25/1)n$.

Since we need to have non-negative x and y (non-negative number of apples and oranges), we have two more additional constraints:
$-4195 + 18n \geq 0$ and $5873 - 25n \geq 0$, or
$4195/18 \leq n \leq 5873/25$, or
$233.05 \leq n \leq 234.92$.

The only possible integer for n is now only 234. Thus the unique solution is $x = -4195 + 18 \times 234 = 17$ and $y = 5873 - 25 \times 234 = 23$, i.e. 17 apples (of 25 cents each) and 23 oranges (of 18 cents each) of a total of 8.39 SGD.

5.5.10 Remarks about Number Theory in Programming Contests

There are many other number theoretic problems that cannot be discussed one by one in this book. Based on our experience, number theory problems frequently appear in ICPCs especially in Asia. It is therefore a good idea for one team member to specifically study number theory listed in this book and beyond.

- Prime Numbers
  1. UVa 00406 - Prime Cuts (sieve; take the middle ones)
  2. **UVa 00543 - Goldbach's Conjecture *** (sieve; complete search; Christian Goldbach's conjecture (updated by Leonhard Euler): Every even number ≥ 4 can be expressed as the sum of two prime numbers)
  3. UVa 00686 - Goldbach's Conjecture (II) (similar to UVa 543)
  4. UVa 00897 - Annagramatic Primes (sieve; just need to check digit rotations)
  5. UVa 00914 - Jumping Champion (sieve; be careful with L and $U < 2$)
  6. **UVa 10140 - Prime Distance *** (sieve; linear scan)
  7. UVa 10168 - Summation of Four Primes (backtracking with pruning)
  8. UVa 10311 - Goldbach and Euler (case analysis, brute force, see UVa 543)
  9. **UVa 10394 - Twin Primes *** (sieve; check if p and $p+2$ are both primes; if yes, they are twin primes; precalculate the result)
  10. UVa 10490 - Mr. Azad and his Son (Ad Hoc; precalculate the answers)
  11. UVa 10650 - Determinate Prime (sieve; find 3 uni-distance consecutive primes)
  12. UVa 10852 - Less Prime (sieve; $p = 1$, find the first prime number $\geq \frac{n}{2} + 1$)
  13. UVa 10948 - The Primary Problem (Goldbach's conjecture, see UVa 543)
  14. UVa 11752 - The Super Powers (try base: 2 to $\sqrt[4]{2^{64}}$, composite power, sort)
- GCD and/or LCM
  1. UVa 00106 - Fermat vs. Phytagoras (brute force; use GCD to get relatively prime triples)
  2. UVa 00332 - Rational Numbers from ... (use GCD to simplify fraction)
  3. UVa 00408 - Uniform Generator (cycle finding problem with easier solution: it is a good choice if `step` < `mod` and `GCD(step, mod) == 1`)
  4. UVa 00412 - Pi (brute force; GCD to find elements with no common factor)
  5. **UVa 10407 - Simple Division *** (subtract the set s with s[0], find gcd)
  6. **UVa 10892 - LCM Cardinality *** (number of divisor pairs of N: (m, n) such that $gcd(m, n) = 1$)
  7. UVa 11388 - GCD LCM (understand the relationship of GCD with LCM)
  8. UVa 11417 - GCD (brute force, input is small)
  9. *UVa 11774 - Doom's Day* (find pattern involving gcd with small test cases)
  10. **UVa 11827 - Maximum GCD *** (GCD of many numbers, small input)
  11. *UVa 12068 - Harmonic Mean* (involving fraction, use LCM and GCD)
- Factorial
  1. **UVa 00324 - Factorial Frequencies *** (count digits of n! up to 366!)
  2. UVa 00568 - Just the Facts (can use Java BigInteger, slow but AC)
  3. **UVa 00623 - 500 (factorial) *** (easy with Java BigInteger)
  4. UVa 10220 - I Love Big Numbers (use Java BigInteger; precalculate)
  5. UVa 10323 - Factorial. You Must ... (overflow: n>13/-odd n; underflow: n<8/-even n; PS: actually, factorial of negative number is not defined)
  6. **UVa 10338 - Mischievous Children *** (use long long to store up to 20!)
- Finding Prime Factors
  1. **UVa 00516 - Prime Land *** (problem involving prime-power factorization)

2. **UVa 00583 - Prime Factors *** (basic prime factorization problem)
3. UVa 10392 - Factoring Large Numbers (enumerate the prime factors of input)
4. **UVa 11466 - Largest Prime Divisor *** (use efficient sieve implementation to get the largest prime factors)

- Working with Prime Factors

  1. UVa 00160 - Factors and Factorials (precalc small primes as prime factors of 100! is $<$ 100)
  2. UVa 00993 - Product of digits (find divisors from 9 down to 1)
  3. UVa 10061 - How many zeros & how ... (in Decimal, '10' with 1 zero is due to factor 2×5)
  4. **UVa 10139 - Factovisors *** (discussed in this section)
  5. UVa 10484 - Divisibility of Factors (prime factors of factorial, D can be -ve)
  6. UVa 10527 - Persistent Numbers (similar to UVa 993)
  7. UVa 10622 - Perfect P-th Power (get GCD of all prime powers, special case if x is -ve)
  8. **UVa 10680 - LCM *** (use prime factors of `[1..N]` to get LCM(1,2,...,N))
  9. UVa 10780 - Again Prime? No time. (similar but different problem with UVa 10139)
  10. UVa 10791 - Minimum Sum LCM (analyze the prime factors of N)
  11. UVa 11347 - Multifactorials (prime-power factorization; `numDiv(N)`)
  12. *UVa 11395 - Sigma Function* (key hint: a square number multiplied by powers of two, i.e. $2^k \times i^2$ for $k \geq 0, i \geq 1$ has *odd* sum of divisors)
  13. **UVa 11889 - Benefit *** (LCM, involving prime factorization)

- Functions involving Prime Factors

  1. **UVa 00294 - Divisors *** (`numDiv(N)`)
  2. UVa 00884 - Factorial Factors (`numPF(N)`; precalculate)
  3. UVa 01246 - Find Terrorists (LA 4340, Amrita08, `numDiv(N)`)
  4. **UVa 10179 - Irreducible Basic ... *** (`EulerPhi(N)`)
  5. UVa 10299 - Relatives (`EulerPhi(N)`)
  6. UVa 10820 - Send A Table (`a[i] = a[i - 1] + 2 * EulerPhi(i)`)
  7. *UVa 10958 - How Many Solutions?* (`2 * numDiv(n * m * p * p) - 1`)
  8. UVa 11064 - Number Theory (`N - EulerPhi(N) - numDiv(N)`)
  9. UVa 11086 - Composite Prime (find numbers N with `numPF(N) == 2`)
  10. UVa 11226 - Reaching the fix-point (`sumPF(N)`; get length; DP)
  11. *UVa 11353 - A Different kind of Sorting* (`numPF(N)`; modified sorting)
  12. **UVa 11728 - Alternate Task *** (`sumDiv(N)`)
  13. *UVa 12005 - Find Solutions* (`numDiv(4N-3)`)

- Modified Sieve

  1. **UVa 10699 - Count the Factors *** (`numDiffPF(N)` for a range of N)
  2. **UVa 10738 - Riemann vs. Mertens *** (`numDiffPF(N)` for a range of N)
  3. ***UVa 10990 - Another New Function **** (modified sieve to compute a range of Euler Phi values; use DP to compute depth Phi values; then finally use Max 1D Range Sum DP to output the answer)
  4. UVa 11327 - Enumerating Rational ... (pre-calculate `EulerPhi(N)`)
  5. *UVa 12043 - Divisors* (`sumDiv(N)` and `numDiv(N)`; brute force)

- Modulo Arithmetic

1. UVa 00128 - Software CRC ($(a \times b) mod s = ((a mod s) * (b mod s)) mod s$)
2. **UVa 00374 - Big Mod *** (solvable with Java BigInteger `modPow`; or write your own code, see Section 9.21)
3. UVa 10127 - Ones (no factor of 2 and 5 implies that there is no trailing zero)
4. UVa 10174 - Couple-Bachelor-Spinster ... (no Spinster number)
5. **UVa 10176 - Ocean Deep; Make it ... *** (discussed in this section)
6. **UVa 10212 - The Last Non-zero Digit *** (there is a modulo arithmetic solution: multiply numbers from N down to $N - M + 1$; repeatedly use /10 to discard the trailing zero(es), and then use %1 Billion to only memorize the last few (maximum 9) non zero digits)
7. UVa 10489 - Boxes of Chocolates (keep working values small with modulo)
8. *UVa 11029 - Leading and Trailing* (combination of logarithmic trick to get the first three digits and 'big mod' trick to get the last three digits)

- Extended Euclid:

1. **UVa 10090 - Marbles *** (use solution for Linear Diophantine Equation)
2. **UVa 10104 - Euclid Problem *** (pure problem of Extended Euclid)
3. UVa 10633 - Rare Easy Problem (this problem can be modeled as Linear Diophantine Equation; let $C = N - M$ (the given input), $N = 10a + b$ (N is at least two digits, with b as the last digit), and $M = a$; this problem is now about finding the solution of the Linear Diophantine Equation: $9a + b = C$)
4. **UVa 10673 - Play with Floor and Ceil *** (uses Extended Euclid)

- Other Number Theory Problems

1. UVa 00547 - DDF (a problem about 'eventually constant' sequence)
2. UVa 00756 - Biorhythms (Chinese Remainder Theorem)
3. **UVa 10110 - Light, more light *** (check if n is a square number)
4. UVa 10922 - 2 the 9s (test divisibility by 9)
5. UVa 10929 - You can say 11 (test divisibility by 11)
6. UVa 11042 - Complex, difficult and ... (case analysis; only 4 possible outputs)
7. **UVa 11344 - The Huge One *** (read M as string, use divisibility theory of `[1..12]`)
8. **UVa 11371 - Number Theory for ... *** (the solving strategy is given)

Profile of Algorithm Inventors

John Pollard (born 1941) is a British mathematician who has invented algorithms for the factorization of large numbers (the Pollard's rho algorithm) and for the calculation of discrete logarithms (not discussed in this book).

Richard Peirce Brent (born 1946) is an Australian mathematician and computer scientist. His research interests include number theory (in particular factorization), random number generators, computer architecture, and analysis of algorithms. He has invented or co-invented various mathematics algorithms. In this book, we discuss Brent's cycle-finding algorithm (see **Exercise 5.7.1***) and Brent's improvement of the Pollard's rho algorithm (see **Exercise 5.5.4.2*** and Section 9.26).

5.6 Probability Theory

Probability Theory is a branch of mathematics dealing with the analysis of random phenomena. Although an event like an individual (fair) coin toss is random, the sequence of random events will exhibit certain statistical patterns if the event is repeated many times. This can be studied and predicted. The probability of a head appearing is 1/2 (similarly with a tail). Therefore, if we flip a (fair) coin n times, we *expect* that we see heads $n/2$ times.

In programming contests, problems involving probability are either solvable with:

- Closed-form formula. For these problems, one has to derive the required (usually $O(1)$) formula. For example, let's discuss how to derive the solution for UVa 10491 - Cows and Cars, which is a generalized version of a TV show: 'The Monty Hall problem'[15].

  You are given NCOWS number of doors with cows, NCARS number of doors with cars, and NSHOW number of doors (with cows) that are opened for you by the presenter. Now, you need to count the probability of winning a car assuming that you will always switch to another unopened door.

  The first step is to realize that there are two ways to get a car. Either you pick a cow first and then switch to a car, or you pick a car first, and then switch to another car. The probability of each case can be computed as shown below.

  In the first case, the chance of picking a cow first is (NCOWS / (NCOWS+NCARS)). Then, the chance of switching to a car is (NCARS / (NCARS+NCOWS-NSHOW-1)). Multiply these two values together to get the probability of the first case. The -1 is to account for the door that you have already chosen, as you cannot switch to it.

  The probability of the second case can be computed in a similar manner. The chance of picking a car first is (NCARS / (NCARS+NCOWS)). Then, the chance of switching to a car is ((NCARS-1) / (NCARS+NCOWS-NSHOW-1)). Both -1 accounts for the car that you have already chosen.

  Sum the probability values of these two cases together to get the final answer.

- Exploration of the search (sample) space to count number of events (usually harder to count; may deal with combinatorics—see Section 5.4, Complete Search—see Section 3.2, or Dynamic Programming—see Section 3.5) over the countable sample space (usually much simpler to count). Examples:

  - 'UVa 12024 - Hats' is a problem of n people who store their n hats in a cloakroom for an event. When the event is over, these n people take their hats back. Some take a wrong hat. Compute how likely is that *everyone* take a wrong hat?

    This problem can be solved via brute-force and pre-calculation by trying all $n!$ permutations and see how many times the required events appear over $n!$ because $n \leq 12$ in this problem. However, a more math-savvy contestant can use this Derangement (DP) formula instead: $A_n = (n-1) \times (A_{n-1} + A_{n-2})$.

  - 'UVa 10759 - Dice Throwing' has a short description: n common cubic dice are thrown. What is the probability that the sum of all thrown dices is at least x? (constraints: $1 \leq n \leq 24$, $0 \leq x < 150$).

[15] This is an interesting probability puzzle. Readers who have not heard this problem before is encouraged to do some Internet search and read the history of this problem. In the original problem, NCOWS = 2, NCARS = 1, and NSHOW = 1. The probability of staying with your original choice is $\frac{1}{3}$ and the probability of switching to another unopened door is $\frac{2}{3}$ and therefore it is always beneficial to switch.

The sample space (the denominator of the probability value) is very simple to compute. It is 6^n.

The number of events is slightly harder to compute. We need a (simple) DP because there are lots of overlapping subproblems. The state is $(dice\_left, score)$ where $dice\_left$ keeps track of the remaining dice that we can still throw (starting from n) and $score$ counts the accumulated score so far (starting from 0). DP can be used as there are only $24 \times (24 \times 6) = 3456$ distinct states for this problem.

When $dice\_left = 0$, we return 1 (event) if $score \geq x$, or return 0 otherwise; When $dice\_left > 0$, we try throwing one more dice. The outcome v for this dice can be one of six values and we move to state $(dice\_left - 1, score + v)$. We sum all the events.

One final requirement is that we have to use gcd (see Section 5.5.2) to simplify the probability fraction. In some other problems, we may be asked to output the probability value correct to a certain digit after decimal point.

Programming Exercises about Probability Theory:

1. *UVa 00542 - France '98* (divide and conquer)
2. UVa 10056 - What is the Probability? (get the closed form formula)
3. *UVa 10218 - Let's Dance* (probability and a bit of binomial coefficients)
4. UVa 10238 - Throw the Dice (similar to UVa 10759; use Java BigInteger)
5. UVa 10328 - Coin Toss (DP, 1-D state, Java BigInteger)
6. **UVa 10491 - Cows and Cars *** (discussed in this section)
7. **UVa 10759 - Dice Throwing *** (discussed in this section)
8. *UVa 10777 - God, Save me* (expected value)
9. *UVa 11021 - Tribbles* (probability)
10. ***UVa 11176 - Winning Streak **** (DP, s: (n_left, max_streak) where n_left is the number of remaining games and max_streak stores the longest consecutive wins; t: lose this game, or win the next W = [1..n_left] games and lose the (W+1)-th game; special case if W = n_left)
11. UVa 11181 - Probability (bar) Given (iterative brute force, try all possibilities)
12. *UVa 11346 - Probability* (a bit of geometry)
13. UVa 11500 - Vampires (Gambler's Ruin Problem)
14. UVa 11628 - Another lottery (p[i] = ticket[i] / total; use gcd to simplify fraction)
15. UVa 12024 - Hats (discussed in this section)
16. *UVa 12114 - Bachelor Arithmetic* (simple probability)
17. *UVa 12457 - Tennis contest* (simple expected value problem; use DP)
18. *UVa 12461 - Airplane* (brute force small n to see that the answer is very easy)

5.7 Cycle-Finding

Given a function $f : S \rightarrow S$ (that maps a natural number from a *finite set* S to another natural number in the same finite set S) and an initial value $x_0 \in N$, the sequence of **iterated function values**: $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots\}$ must eventually use the same value twice, i.e. $\exists i \neq j$ such that $x_i = x_j$. Once this happens, the sequence must then repeat the cycle of values from x_i to x_{j-1}. Let μ (the start of cycle) be the smallest index i and λ (the cycle length) be the smallest positive integer such that $x_\mu = x_{\mu+\lambda}$. The **cycle-finding** problem is defined as the problem of finding μ and λ given $f(x)$ and x_0.

For example in UVa 350 - Pseudo-Random Numbers, we are given a pseudo-random number generator $f(x) = (Z \times x + I)\%M$ with $x_0 = L$ and we want to find out the sequence length before any number is repeated (i.e. the λ). A good pseudo-random number generator should have a large λ. Otherwise, the numbers generated will not look 'random'.

Let's try this process with the sample test case $Z = 7, I = 5, M = 12, L = 4$, so we have $f(x) = (7 \times x + 5)\%12$ and $x_0 = 4$. The sequence of iterated function values is $\{4, 9, 8, 1, 0, 5, \underline{4, 9, 8, 1, 0, 5}, \ldots\}$. We have $\mu = 0$ and $\lambda = 6$ as $x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$. The sequence of iterated function values cycles from index 6 onwards.

On another test case $Z = 3, I = 1, M = 4, L = 7$, we have $f(x) = (3 \times x + 1)\%4$ and $x_0 = 7$. The sequence of iterated function values is $\{7, 2, 3, \underline{2, 3}, \ldots\}$. This time, we have $\mu = 1$ and $\lambda = 2$.

5.7.1 Solution(s) using Efficient Data Structure

A simple algorithm that will work for *many cases* of this cycle-finding problem uses an efficient data structure to store pair of information that a number x_i has been encountered at iteration i in the sequence of iterated function values. Then for x_j that is encountered later ($j > i$), we test if x_j is already stored in the data structure. If it is, it implies that $x_j = x_i$, $\mu = i$, $\lambda = j - i$. This algorithm runs in $O((\mu + \lambda) \times DS\_cost)$ where DS_cost is the cost per one data structure operation (insert/search). This algorithm requires at least $O(\mu + \lambda)$ space to store past values.

For many cycle-finding problems with rather large S (and likely large $\mu + \lambda$), we can use $O(\mu + \lambda)$ space C++ STL `map`/Java `TreeMap` to store/check the iteration indices of past values in $O(\log(\mu + \lambda))$ time. But if we just need to stop the algorithm upon encountering the *first* repeated number, we can use C++ STL `set`/Java `TreeSet` instead.

For other cycle-finding problems with relatively small S (and likely small $\mu + \lambda$), we may use the $O(|S|)$ space Direct Addressing Table to store/check the iteration indices of past values in $O(1)$ time. Here, we trade-off memory space for runtime speed.

5.7.2 Floyd's Cycle-Finding Algorithm

There is a better algorithm called Floyd's cycle-finding algorithm that runs in $O(\mu + \lambda)$ time complexity and *only* uses $O(1)$ memory space—much smaller than the simple versions shown above. This algorithm is also called 'the tortoise and hare (rabbit)' algorithm. It has three components that we describe below using the UVa 350 problem as shown above with $Z = 3, I = 1, M = 4, L = 7$.

Efficient Way to Detect a Cycle: Finding $k\lambda$

Observe that for any $i \geq \mu$, $x_i = x_{i+k\lambda}$, where $k > 0$, e.g. in Table 5.2, $x_1 = x_{1+1\times2} = x_3 = x_{1+2\times2} = x_5 = 2$, and so on. If we set $k\lambda = i$, we get $x_i = x_{i+i} = x_{2i}$. Floyd's cycle finding algorithm exploits this trick.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
Init	TH						
1		T	H				
2			$\underline{\text{T}}$		$\underline{\text{H}}$		

Table 5.2: Part 1: Finding $k\lambda$, $f(x) = (3 \times x + 1)\%4$, $x_0 = 7$

The Floyd's cycle-finding algorithm maintains two pointers called 'tortoise' (the slower one) at x_i and 'hare' (the faster one that keeps jumping around) at x_{2i}. Initially, both are at x_0. At each step of the algorithm, tortoise is moved *one step* to the right and the hare is moved *two steps* to the right[16] in the sequence. Then, the algorithm compares the sequence values at these two pointers. The smallest value of $i > 0$ for which both tortoise and hare point to equal values is the value of $k\lambda$ (multiple of λ). We will determine the actual λ from $k\lambda$ using the next two steps. In Table 5.2, when $i = 2$, we have $x_2 = x_4 = x_{2+\underline{\mathbf{2}}} = x_{2+k\lambda} = 3$. So, $k\lambda = 2$. In this example, we will see below that k is eventually 1, so $\lambda = 2$ too.

Finding μ

Next, we reset hare back to x_0 and keep tortoise at its current position. Now, we advance *both* pointers to the right one step at a time, thus maintaining the $k\lambda$ gap between the two pointers. When tortoise and hare points to the same value, we have just found the *first* repetition of length $k\lambda$. Since $k\lambda$ is a multiple of λ, it must be true that $x_\mu = x_{\mu+k\lambda}$. The first time we encounter the first repetition of length $k\lambda$ is the value of the μ. In Table 5.3, we find that $\mu = 1$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
1	H		T				
2		$\underline{\text{H}}$		$\underline{\text{T}}$			

Table 5.3: Part 2: Finding μ

Finding λ

Once we get μ, we let tortoise stays in its current position and set hare next to it. Now, we move hare iteratively to the right one by one. Hare will point to a value that is the same as tortoise for the *first* time after λ steps. In Table 5.4, after hare moves once, $x_3 = x_{3+2} = x_5 = 2$. So, $\lambda = 2$.

step	x_0	x_1	x_2	x_3	x_4	x_5	x_6
	7	2	3	2	3	2	3
1				T	H		
2				$\underline{\text{T}}$		$\underline{\text{H}}$	

Table 5.4: Part 3: Finding λ

Therefore, we report $\mu = 1$ and $\lambda = 2$ for $f(x) = (3 \times x + 1)\%4$ and $x_0 = 7$.
In overall, this algorithm runs in $O(\mu + \lambda)$.

[16] To move right one step from x_i, we use $x_i = f(x_i)$. To move right two steps from x_i, we use $x_i = f(f(x_i))$.

The Implementation

The working C/C++ implementation of this algorithm (with comments) is shown below:

```
ii floydCycleFinding(int x0) {  // function int f(int x) is defined earlier
  // 1st part: finding k*mu, hare's speed is 2x tortoise's
  int tortoise = f(x0), hare = f(f(x0));    // f(x0) is the node next to x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2nd part: finding mu, hare and tortoise move at the same speed
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++;}
  // 3rd part: finding lambda, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}
```

Source code: `ch5_07_UVa350.cpp/java`

Exercise 5.7.1*: Richard P. Brent invented an improved version of Floyd's cycle-finding algorithm shown above. Study and implement Brent's algorithm [3]!

Programming Exercises related to Cycle-Finding:

1. UVa 00202 - Repeating Decimals (do expansion digit by digit until it cycles)
2. UVa 00275 - Expanding Fractions (same as UVa 202 except the output format)
3. **UVa 00350 - Pseudo-Random Numbers *** (discussed in this section)
4. UVa 00944 - Happy Numbers (similar to UVa 10591)
5. UVa 10162 - Last Digit (cycle after 100 steps, use Java BigInteger to read the input, precalculate)
6. UVa 10515 - Power et al (concentrate on the last digit)
7. UVa 10591 - Happy Number (this sequence is 'eventually periodic')
8. UVa 11036 - Eventually periodic ... (cycle-finding, evaluate Reverse Polish `f` with a `stack`—also see Section 9.27)
9. **UVa 11053 - Flavius Josephus ... *** (cycle-finding, the answer is $N - \lambda$)
10. UVa 11549 - Calculator Conundrum (repeat squaring with limited digits until it cycles; that is, the Floyd's cycle-finding algorithm is only used to detect the cycle, we do not use the value of μ or λ; instead, we keep track the largest iterated function value found before any cycle is encountered)
11. **UVa 11634 - Generate random ... *** (use DAT of size 10K, extract digits; the programming trick to square 4 digits 'a' and get the resulting middle 4 digits is `a = (a * a / 100) % 10000`)
12. *UVa 12464 - Professor Lazy, Ph.D.* (although n can be very huge, the pattern is actually cyclic; find the length of the cycle l and modulo n with l)

5.8 Game Theory

Game Theory is a mathematical model of strategic situations (not necessarily *games* as in the common meaning of 'games') in which a player's success in making choices depends on the choices of *others*. Many programming problems involving game theory are classified as **Zero-Sum Game**—a mathematical way of saying that if one player wins, then the other player loses. For example, a game of Tic-Tac-Toe (e.g. UVa 10111), Chess, various number/integer games (e.g. UVa 847, 10368, 10578, 10891, 11489), and others (UVa 10165, 10404, 11311) are games with two players playing alternately (usually perfectly) and there can only be one winner.

The common question asked in programming contest problems related to game theory is whether the starting player of a two player competitive game has a winning move assuming that both players are doing **Perfect Play**. That is, each player always choose the most optimal choice available to him.

5.8.1 Decision Tree

One solution is to write a recursive code to explore the **Decision Tree** of the game (a.k.a. the Game Tree). If there is no overlapping subproblem, pure recursive backtracking is suitable. Otherwise, Dynamic Programming is needed. Each vertex describes the current player and the current state of the game. Each vertex is connected to all other vertices legally reachable from that vertex according to the game rules. The root vertex describes the starting player and the initial game state. If the game state at a leaf vertex is a winning state, it is a win for the current player (and a lose for the other player). At an internal vertex, the current player chooses a vertex that guarantees a win with the largest margin (or if a win is not possible, choose a vertex with the least loss). This is called the **Minimax** strategy.

For example, in UVa 10368 - Euclid's Game, there are two players: Stan (player 0) and Ollie (player 1). The state of the game is a triple of integers (id, a, b). The current player id can subtracts any positive multiple of the lesser of the two numbers, integer b, from the greater of the two numbers, integer a, provided that the resulting number must be nonnegative. We always maintain that $a \geq b$. Stan and Ollie plays alternately, until one player is able to subtract a multiple of the lesser number from the greater to reach 0, and thereby wins. The first player is Stan. The decision tree for a game with initial state $id = 0$, $a = 34$, and $b = 12$ is shown below in Figure 5.2 .

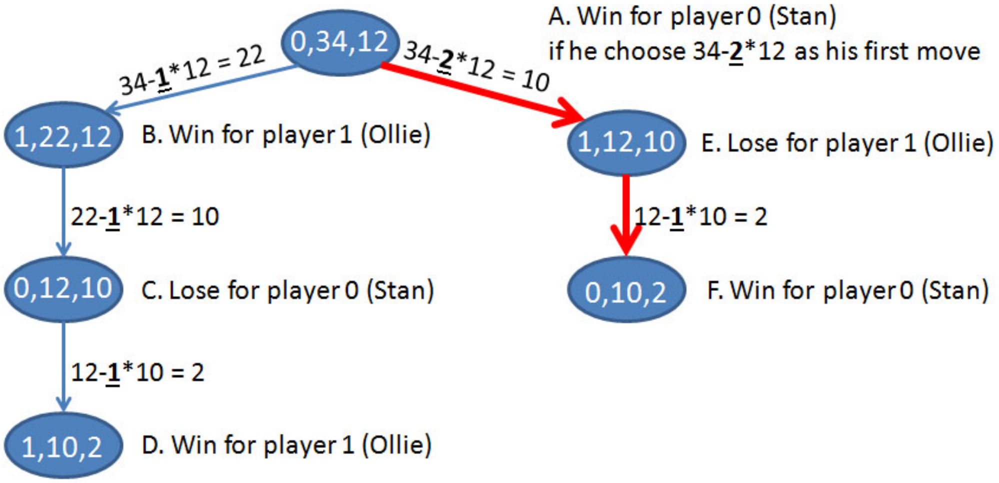
0,34,12
A. Win for player 0 (Stan)
if he choose 34-2*12 as his first move
34-1*12 = 22
34-2*12 = 10
1,22,12
B. Win for player 1 (Ollie)
1,12,10
E. Lose for player 1 (Ollie)
22-1*12 = 10
12-1*10 = 2
0,12,10
C. Lose for player 0 (Stan)
0,10,2
F. Win for player 0 (Stan)
12-1*10 = 2
1,10,2
D. Win for player 1 (Ollie)

Figure 5.2: Decision Tree for an instance of 'Euclid's Game'

Let's trace what happens in Figure 5.2. At the root (initial state), we have triple $(0, 34, 12)$. At this point, player 0 (Stan) has two choices: Either to subtract $a - b = 34 - 12 = 22$ and move to vertex $(1, 22, 12)$ (the left branch) or to subtract $a - 2 \times b = 24 - 2 \times 12 = 10$ and move to vertex $(1, 12, 10)$ (the right branch). We try both choices recursively.

Let's start with the left branch. At vertex $(1, 22, 12)$—(Figure 5.2.B), the current player 1 (Ollie) has no choice but to subtract $a - b = 22 - 12 = 10$. We are now at vertex $(0, 12, 10)$—(Figure 5.2.C). Again, Stan only has one choice which is to subtract $a - b = 12 - 10 = 2$. We are now at leaf vertex $(1, 10, 2)$—(Figure 5.2.D). Ollie has several choices but Ollie can definitely win as $a - 5 \times b = 10 - 5 \times 2 = 0$ and it implies that vertex $(0, 12, 10)$ is a losing state for Stan and vertex $(1, 22, 12)$ is a winning state for Ollie.

Now we explore the right branch. At vertex $(1, 12, 10)$—(Figure 5.2.E), the current player 1 (Ollie) has no choice but to subtract $a - b = 12 - 10 = 2$. We are now at leaf vertex $(0, 10, 2)$—(Figure 5.2.F). Stan has several choices but Stan can definitely win as $a - 5 \times b = 10 - 5 \times 2 = 0$ and it implies that vertex $(1, 12, 10)$ is a losing state for Ollie.

Therefore, for player 0 (Stan) to win this game, Stan should choose $a - 2 \times b = 34 - 2 \times 12$ first, as this is a winning move for Stan—(Figure 5.2.A).

Implementation wise, the first integer *id* in the triple can be dropped as we know that depth 0 (root), 2, 4, ... are always Stan's turns and depth 1, 3, 5, ... are always Ollie's turns. This integer *id* is used in Figure 5.2 to simplify the explanation.

5.8.2 Mathematical Insights to Speed-up the Solution

Not all game theory problems can be solved by exploring the *entire* decision tree of the game, especially if the size of the tree is large. If the problem involves numbers, we may need to come up with some mathematical insights to speed up the computation.

For example, in UVa 847 - A multiplication game, there are two players: Stan (player 0) and Ollie (player 1) again. The state of the game[17] is an integer p. The current player can multiply p with any number between 2 to 9. Stan and Ollie also plays alternately, until one player is able to multiply p with a number between 2 to 9 such that $p \geq n$ (n is the target number), thereby wins. The first player is Stan with $p = 1$.

Figure 5.3 shows an instance of this multiplication game with $n = 17$. Initially, player 0 has up to 8 choices (to multiply $p = 1$ by [2..9]). However, all of these 8 states are winning states of player 1 as player 1 can always multiply the current p by [2..9] to make $p \geq 17$—(Figure 5.3.B). Therefore player 0 will surely lose—(Figure 5.3.A).

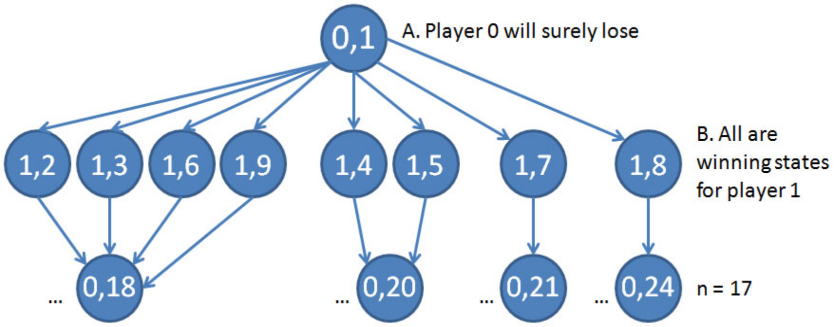
0,1
A. Player 0 will surely lose
1,2
1,3
1,6
1,9
1,4
1,5
1,7
1,8
B. All are winning states for player 1
...
0,18
...
0,20
...
0,21
...
0,24
n = 17

Figure 5.3: Partial Decision Tree for an instance of 'A multiplication game'

As $1 < n < 4294967295$, the resulting decision tree on the largest test case can be extremely huge. This is because each vertex in this decision tree has a *huge* branching factor of 8 (as

[17] This time we omit the player *id*. However, this parameter *id* is still shown in Figure 5.3 for clarity.

there are 8 possible numbers to choose from between 2 to 9). It is not feasible to actually explore the decision tree.

It turns out that the optimal strategy for Stan to win is to *always* multiply p with 9 (the largest possible) while Ollie will *always* multiply p with 2 (the smallest possible). Such optimization insights can be obtained by observing the pattern found in the output of smaller instances of this problem. Note that a maths-savvy contestants may want to prove this observation first before coding the solution.

5.8.3 Nim Game

There is a special game that is worth mentioning as it may appear in programming contests: The **Nim** game[18]. In Nim game, two players take turns to remove objects from distinct heaps. On each turn, a player must remove *at least one object* and may remove *any number of objects* provided they all come from the same heap. The initial state of the game is the number of objects n_i at each of the k heaps: $\{n_1, n_2, \ldots, n_k\}$. There is a nice solution for this game. For the first (starting) player to win, the value of n_1 ^ n_2 ^ ... ^ n_k must be *non zero* where ^ is the bit operator xor (exclusive or)—proof omitted.

Programming Exercises related to Game Theory:

1. UVa 00847 - A multiplication game (simulate the perfect play, discussed above)
2. **UVa 10111 - Find the Winning ... *** (Tic-Tac-Toe, minimax, backtracking)
3. UVa 10165 - Stone Game (Nim game, application of Sprague-Grundy theorem)
4. UVa 10368 - Euclid's Game (minimax, backtracking, discussed in this section)
5. UVa 10404 - Bachet's Game (2 players game, Dynamic Programming)
6. UVa 10578 - The Game of 31 (backtracking; try all; see who wins the game)
7. **UVa 11311 - Exclusively Edible *** (game theory, reducible to Nim game; we can view the game that Handel and Gretel are playing as Nim game, where there are 4 heaps - cakes left/below/right/above the topping; take the Nim sum of these 4 values and if they are equal to 0, Handel loses)
8. **UVa 11489 - Integer Game *** (game theory, reducible to simple math)
9. *UVa 12293 - Box Game* (analyze the game tree of smaller instances to get the mathematical insight to solve this problem)
10. *UVa 12469 - Stones* (game playing, Dynamic Programming, pruning)

[18] The general form of two player games is inside the IOI syllabus [20], but Nim game is not.

5.9 Solution to Non-Starred Exercises

Exercise 5.2.1: The `<cmath>` library in C/C++ has two functions: `log` (base e) and `log10` (base 10); `Java.lang.Math` only has `log` (base e). To get `log`$_b$`(a)` (base b), we use the fact that `log`$_b$`(a) = log(a) / log(b)`.

Exercise 5.2.2: `(int)floor(1 + log10((double)a))` returns the number of digits in decimal number `a`. To count the number of digits in other base `b`, we can use similar formula: `(int)floor(1 + log10((double)a) / log10((double)b))`.

Exercise 5.2.3: $\sqrt[n]{a}$ can be rewritten as $a^{1/n}$. We can then use built in formula like `pow((double)a, 1.0 / (double)n)` or `exp(log((double)a) * 1.0 / (double)n)`.

Exercise 5.3.1.1: Possible, keep the intermediate computations **modulo** 10^6. Keep chipping away the trailing zeroes (either none or a few zeroes are added after a multiplication from $n!$ to $(n+1)!$).

Exercise 5.3.1.2: Possible. $9317 = 7 \times 11^3$. We also list 25! as its prime factors. Then, we check if there are one factor 7 (yes) and three factors 11 (unfortunately no). So 25! is not divisible by 9317. Alternative approach: Use modulo arithmetic (see Section 5.5.8).

Exercise 5.3.2.1: For base number conversion of 32-bit integers, use `parseInt(String s, int radix)` and `toString(int i, int radix)` in the faster Java **Integer** class. You can also use `BufferedReader` and `BufferedWriter` for I/O (see Section 3.2.3).

Exercise 5.4.1.1: Binet's closed-form formula for Fibonacci: $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ should be correct for larger n. But since double precision data type is limited, we have discrepancies for larger n. This closed form formula is correct up to $fib(75)$ if implemented using typical double data type in a computer program. This is unfortunately too small to be useful in typical programming contest problems involving Fibonacci numbers.

Exercise 5.4.2.1: $C(n,2) = \frac{n!}{(n-2)! \times 2!} = \frac{n \times (n-1) \times (n-2)!}{(n-2)! \times 2} = \frac{n \times (n-1)}{2} = 0.5n^2 - 0.5n = O(n^2)$.

Exercise 5.4.4.1: Fundamental counting principle: If there are m ways to do one thing and n ways to do another thing, then there are $m \times n$ ways to do both. The answer for this exercise is therefore: $6 \times 6 \times 2 \times 2 = 6^2 \times 2^2 = 36 \times 4 = 144$ different possible outcomes.

Exercise 5.4.4.2: See above. The answer is: $9 \times 9 \times 8 = 648$. Initially there are 9 choices (1-9), then there are still 9 choices (1-9 minus 1, plus 0), then finally there are only 8 choices.

Exercise 5.4.4.3: A permutation is an arrangement of objects without repetition and the order is important. The formula is ${}_nP_r = \frac{n!}{(n-r)!}$. The answer for this exercise is therefore: $\frac{6!}{(6-3)!} = 6 \times 5 \times 4 = 120$ 3-letters words.

Exercise 5.4.4.4: The formula to count different permutations is: $\frac{n!}{(n_1)! \times (n_2)! \times \ldots \times (n_k)!}$ where n_i is the frequency of each unique letter i and $n_1 + n_2 + \ldots + n_k = n$. The answer for this exercise is therefore: $\frac{5!}{3! \times 1! \times 1!} = \frac{120}{6} = 20$ because there are 3 'B's, 1 'O', and 1 'Y'.

Exercise 5.4.4.5: The answers for few small n = 3, 4, 5, 6, 7, 8, 9, and 10 are 0, 1, 3, 7, 13, 22, 34, and 50, respectively. You can generate these numbers using brute force solution first. Then find the pattern and use it.

Exercise 5.5.2.1: Multiplying $a \times b$ first before dividing the result by $gcd(a,b)$ has a higher chance of overflow in programming contest than $a \times (b/gcd(a,b))$. In the example given, we have $a = 1000000000$ and $b = 8$. The LCM is 1000000000—which should fit in 32-bit signed integers—can only be properly computed with $a \times (b/gcd(a,b))$.

Exercise 5.5.4.1: Since the largest prime in `vi 'primes'` is 9999991, this code can therefore handles $N \leq 9999991^2 = 99999820000081 \approx 9 \times 10^{13}$. If the smallest prime factor of N is greater than 9999991, for example, $N = 1010189899^2 = 1020483632041630201 \approx 1 \times 10^{18}$ (this still within the capacity of 64-bit signed integer), this code will crash or produce wrong result. If we decide to drop the usage of vi ‘primes’ and uses $PF = 3, 5, 7, \ldots$ (with special check for $PF = 2$), then we have a slower code and the newer limit for N is now N with smallest prime factor up to $2^{63} - 1$. However, if such input is given, we need to use the algorithms mentioned in **Exercise 5.5.4.2*** and in Section 9.26.

Exercise 5.5.4.2: See Section 9.26.

Exercise 5.5.5.1: GCD(A, B) can be obtained by taking the lower power of the common prime factors of A and B. LCM(A, B) can be obtained by taking the greater power of all the prime factors of A and B. So, GCD($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = $2^5 = 32$ and LCM($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = $2^6 \times 3^3 \times 5^2 \times 11^2 \times 97^1 = 507038400$.

Exercise 5.5.6.1:

```
ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    if (N % PF == 0) ans++;                           // count this pf only once
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

```
ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}
```

Exercise 5.5.7.1: The modified sieve code to compute the Euler Totient function up to 10^6 is shown below:

```
for (int i = 1; i <= 1000000; i++) EulerPhi[i] = i;
for (int i = 2; i <= 1000000; i++)
  if (EulerPhi[i] == i)                                   // i is a prime number
    for (int j = i; j <= 1000000; j += i)
       EulerPhi[j] = (EulerPhi[j] / i) * (i - 1);
```

Exercise 5.5.8.1: Statement 2 and 4 are not valid. The other 3 are valid.

5.10 Chapter Notes

This chapter has grown significantly since the first edition of this book. However, even until the third edition, we become more aware that there are still many more mathematics problems and algorithms that have not been discussed in this chapter, e.g.

- There are many more but rare **combinatorics** problems and formulas that are not yet discussed: **Burnside's lemma**, **Stirling Numbers**, etc.
- There are other theorems, hypothesis, and conjectures that cannot be discussed one by one, e.g. **Carmichael's function**, **Riemann's hypothesis**, **Fermat's Little Test**, **Chinese Remainder Theorem**, **Sprague-Grundy Theorem**, etc.
- We only briefly mention Brent's cycle-finding algorithm (that is slightly faster than Floyd's version) in **Exercise 5.7.1***.
- (Computational) Geometry is also part of Mathematics, but since we have a special chapter for that, we reserve the discussions about geometry problems in Chapter 7.
- Later in Chapter 9, we briefly discuss a few more mathematics-related algorithm, e.g. Gaussian Elimination for solving system of linear equations (Section 9.9), Matrix Power and its usages (Section 9.21), Pollard's rho algorithm (Section 9.26), Postfix Calculator and (Infix to Postfix) Conversion and (Section 9.27), Roman Numerals (Section 9.28).

There are really *many* topics about mathematics. This is not surprising since various mathematics problems have been investigated by people since hundreds years ago. Some of them are discussed in this chapter, many others are not, and yet only 1 or 2 will actually appear in a problem set. To do well in ICPC, it is a good idea to have at least *one strong mathematician* in your ICPC team in order to have those 1 or 2 mathematics problems solved. Mathematical prowess is also important for IOI contestants. Although the amount of problem-specific topics to be mastered is smaller, many IOI tasks require some form of 'mathematical insights'.

We end this chapter by listing some pointers that may be of interest to some readers: Read number theory books, e.g. [56], investigate mathematical topics in `mathworld.wolfram.com` or Wikipedia, and attempt many more programming exercises related to mathematics problems like the ones in `http://projecteuler.net` [17] and `https://brilliant.org` [4].

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	17	29 (+71%)	41 (+41%)
Written Exercises	-	19	20+10*=30 (+58%)
Programming Exercises	175	296 (+69%)	369 (+25%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
5.2	**Ad Hoc Mathematics ...**	144	39%	9%
5.3	Java BigInteger Class	45	12%	3%
5.4	Combinatorics	54	15%	3%
5.5	**Number Theory**	86	23%	5%
5.6	Probability Theory	18	5%	1%
5.7	Cycle-Finding	13	3%	1%
5.8	Game Theory	10	3%	1%

L-R: Steven, Ranald, Hubert, Wei Liang, Bernard, Zi Chun
I Came
Saw
eleted
our files