Mathematics

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

Modular arithmetic

- $a \equiv b \bmod m$ iff there exists some integer k such that $a = b + mk$
- $b \bmod m$ is the remainder of b when divided by m
- In C/C++, the % symbol is used for the modulo operator
- Be careful with negative numbers! In C/C++, the behaviour is:
  - $(-4)\%3 == (-4)\%(-3) == -1$
  - $(-4)\%5 == (-4)\%(-5) == -4$
  - $4\%(-3) == 1$
  - $4\%(-5) == 4$

Modular arithmetic

- If $a \equiv b \bmod m$, then $a + c \equiv b + c \bmod m$
- If $a \equiv b \bmod m$, then $ac \equiv bc \bmod m$
- If $ac \equiv bc \bmod mc$, then $a \equiv b \bmod m$
- $ac \equiv bc \bmod m$ does not necessarily mean $a \equiv b \bmod m$!

Primes

- A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself
- Primes are the fundamental building blocks of all of number theory

Primality testing

- **Algorithm** For every possible factor F of N, check if it divides. Observing that if an $F_1 > \sqrt{N}$ is a factor, then there must be another $F_2 < \sqrt{N}$ that is also a factor, we need only check factors $F \leq \sqrt{N}$.
- **Complexity** $O(\sqrt{N})$ time

Primality testing

- **Implementation**

```
bool isprime(int x) {
  if (x < 2) return false;

  for (int f = 2; f*f <= x; f++) {
    if (x % f == 0) return false;
  }
  return true;
}
```

Sieve of Erastosthenes

- **Algorithm** Starting with 2, mark all multiples of 2 as composite. Then, starting with the next smallest number not marked composite (which therefore must be prime), 3, mark out all its multiples and repeat until we hit the upper bound.
- **Complexity** For some upper bound N and each prime p, we must strike out about $\frac{N}{p}$ multiples, so the amount of work we do is roughly $\frac{N}{p_1} + \frac{N}{p_2} + \ldots + \frac{N}{p_P}$. This is clearly less than $N + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{N}$, so by harmonic series we can say that this algorithm runs in $O(N \log N)$ time.

Sieve of Erastosthenes

- Note that the $O(N \log N)$ bound is not tight, and the actual time complexity is closer to linear, specifically $O(N \log \log N)$
- However, knowing that we have an upper bound that works is usually sufficient for our purposes
- The algorithm itself can be optimised even more, though this is not usually necessary
    - Throw away the even numbers ($2\times$ speed up)
    - For each prime, start marking at its square because the smaller ones will be marked already

Greatest common divisor

- $\gcd(a, b)$ is the greatest integer that divides both a and b
- One of the most commonly used tools in solving number theory problems
- A few useful facts
  - $\gcd(a, b) = \gcd(a, b - a)$
  - $\gcd(a, 0) = a$
  - $\gcd(a, b)$ is the smallest positive number in $\{ax + by : x, y \in \mathbb{Z}\}$

Greatest common divisor

- Can be computed with the Euclidean algorithm, which is the repeated use of the first property above:

$$\gcd(a, b) = \gcd(a, b - a)$$

- Usually, you'll use a similar rule, $\gcd(a, b) = \gcd(a, b\%a)$, although the speed of the two versions is surprisingly close
- This has a complexity of $O(\log(a + b))$

Greatest common divisor

- **Implementation**

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

- Some versions of `<algorithm>` have a `__gcd` function already defined, but it's not always clear when it's available, so usually it's safest to just write it yourself

Extended Euclidean algorithm

- As with the Euclidean algorithm, we incrementally apply the $\gcd(a, b) = \gcd(a, b - a)$ rule until we've found the GCD, but we also explicitly write the intermediate numbers as integer combinations of a and b, i.e. we find x and y where
$$ax + by = \gcd(a, b),$$
which is called *Bézout's identity*

Extended Euclidean algorithm

- **Implementation**

```
int gcd(int a, int b, int& x, int& y) {
  if (a == 0) {
    x = 0; y = 1;
    return b;
  }
  int x1, y1;
  int d = gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return d;
}
```

Modular inverse

- The inverse a^{-1} of a is an integer such that $a^{-1}a \equiv aa^{-1} \equiv 1 \bmod m$
- Only exists if $\gcd(a, m) = 1$
- **Fermat's little theorem** $a^{m-1} \equiv 1 \bmod m$ for prime m
- Euler's theorem is a generalisation that works for general modulus, based on the *totient function*[1], $\phi(n)$
- Can also be computed with the Extended Euclidean algorithm for general modulus[2] (how?)

[1] counts the numbers less than n which are coprime to n

[2] any modulus which is coprime to a

Fast exponentiation

- We can compute a^n quickly using a kind of divide and conquer in $O(\log n)$ time (assuming constant time multiplication)
- Observe that $a^n = a^{n/2} \times a^{n/2}$ for even n and $a^n = a^{n/2} \times a^{n/2} \times a$ for odd n
- This is equivalent to precomputing each a^{2^m}, and combining powers according to the binary expansion of n

Fast exponentiation

- Example:

$$a^9 = a^4 \times a^4 \times a$$
$$a^4 = a^2 \times a^2$$
$$a^2 = a \times a$$

- This same idea can be used to compute repeated applications of functions quickly

Fast exponentiation

- **Implementation**

```
typedef long long ll;

ll pow(ll x, ll k) {
  if (k == 0) return 1;

  ll a = pow(x, k/2);
  a = a*a;
  if (k%2 == 1) a = a*x;
  return a;
}
```

Fibonacci numbers

- A sequence generated from two initial values and a second-order recurrence relation as follows:

$$\begin{aligned} &F(0) = 0 \\ &F(1) = 1 \\ &F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2 \end{aligned}$$

Fibonacci numbers

- Algorithms for Fibonacci from introductory computing courses:
    - **Algorithm 1** Direct computation of the recurrence. This is $O(2^n)$ time and $O(n)$ memory.
    - **Algorithm 2** Using the recurrence, but caching the value of each computation. This is $O(n)$ time and $O(n)$ memory.
    - **Algorithm 3** Using the recurrence with caching, but only keeping around the two most recently computed values. This is still $O(n)$ time, but $O(1)$ memory.

Fibonacci numbers

- **Algorithm 4** Solving the recurrence, we have the closed form
$$F(n) = \frac{\varphi^n - \psi^n}{\varphi - \psi},$$
where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$.
    - "Constant time" - it's a little more complicated than that
    - Precision issues - the numbers in the sequence grow exponentially quickly

Fibonacci numbers

- **Algorithm 5** Using the matrix form of the recurrence:

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

We can get a closed form:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Fibonacci numbers

- We arrange the recurrence into a matrix, form, such that the nth power of the matrix will give us $F(n)$. Using fast exponentiation, we can then compute $F(n)$ in $O(\log n)$ time.
- It's not really $O(\log n)$ time - since there are $O(n)$ bits in $F(n)$, the cost to multiply the numbers starts dominating the actual time complexity.

Fibonacci numbers

- This technique, rearranging the recurrence into matrix form so fast exponentiation can be used, is very common
- This is similar to finding closed form solutions to first order homogeneous recurrences (assuming powers can be computed quickly)
- The closed form Fibonacci solution using φ and ψ is actually along the same lines, but using the eigenvalues of the matrix instead of the matrix itself

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

Binomial coefficients

- The binomial coefficient $\binom{n}{k}$ is the number of ways to make an unordered selection of k elements out of a set of n distinguishable elements
- One of the most widely used tools in combinatorics

Binomial coefficients

- **Algorithm 1** Compute directly from the formula

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

- **Complexity** $O(n)$ to compute the factorials, although parts of this can be precomputed for repeated uses. The intermediate values can become very large, however this problem can be avoided by rearranging this formula in terms of alternating multiplication and division

Binomial coefficients

- **Algorithm 2** Compute from the recurrence

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- **Complexity** There are $O(nk)$ total values of $\binom{n}{k}$, and it takes $O(1)$ time to compute each value, so this takes $O(nk)$ time

Inclusion/exclusion

- $|A \cup B| = |A| + |B| - |A \cap B|$
- $|A \cup B \cup C| =$
  $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$
- In general,

$$|A_1 \cup A_2 \cup \ldots \cup A_n| = \sum_{I \subseteq \{1,\ldots,n\}, I \neq \emptyset} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|$$

Probability

- For the most part is solved via counting techniques, or sometimes dynamic programming
- Expectations are linear: for random variables X and Y and a constant c,

$$\begin{aligned}\mathbb{E}(X+c) &= \mathbb{E}(X)+c\\ \mathbb{E}(X+Y) &= \mathbb{E}(X)+\mathbb{E}(Y)\\ \mathbb{E}(cX) &= c\mathbb{E}(X)\end{aligned}$$

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

Goldbach's conjecture

- **Problem statement** Goldbach's conjecture states that every even integer greater than 2 can be expressed as the sum of two odd primes. For some even integer N, find a pair of odd primes that sums to N.
- **Input** A single integer N, $3 \leq N \leq 1,000,000$
- **Output** A line containing A and B, two odd primes that sum to N, or "Goldbach's conjecture is wrong" if no such numbers exist

Goldbach's conjecture

- **Algorithm** We know that we have to do something with primes. So let's start by generating all primes up to N, using the sieve.
- After we generate our list of primes, the remaining problem is "given an integer, find a pair from this list that sums to the integer"

Goldbach’s conjecture

- We can solve this problem in $O(n)$ time using a set with a fast membership test
- Since our elements are all small integers, we can just use a boolean array
- **Complexity** To create our list of primes, we use our $O(n \log n)$ time, $O(n)$ space sieve to transform this into a simpler problem which we can solve in $O(n)$ time and $O(n)$ space. So this algorithm runs in $O(n \log n)$ time and $O(n)$ space.

Goldbach's conjecture

- **Implementation**

```
#include <cstdio>

const int N = 1e6+5;
int primes[N], P, notprime[N];

int main() {
  // sieve
  notprime[0] = 1;
  notprime[1] = 1;
  for (int i = 2; i < N; i++) {
    if (notprime[i]) continue;
    // if i is prime, mark all its multiples as not prime
    primes[P++] = i;
    for (int j = i*i; j < N; j += i) notprime[j] = true;
  }

  int n;
  while (scanf("%d", &n), n) {
    // scan primes[] for pair adding to n
    for (int i = 1; i < P; i++) {
      if (!notprime[n-primes[i]]) {
        printf("%d = %d + %d\n", n, primes[i], n - primes[i]);
        break;
      }
    }
  }
  return 0;
}
```

Combinations

- **Problem statement** Compute $\binom{n}{k} \bmod 1,000,000,007$
- **Input** Two integers N and K, $0 \leq K \leq N \leq 1,000,000$
- **Output** A line containing $\binom{n}{k}$

Combinations

- **Algorithm** We can't use the recurrence here; $O(NK)$ is too slow when N and K are each up to 1,000,000
- The only viable method is using the formula

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

- We need to be able to divide in our modulus, i.e. compute inverses
- Luckily, 1,000,000,007 is a prime (what a crazy random happenstance!)

Combinations

- We can solve the problem in $O(1)$ per query after computing factorials and their inverses, using Fermat's little theorem and fast exponentiation
- We precompute every factorial and its corresponding inverse, since there are only $O(N)$ of either of these.
- **Complexity** After $O(N \log N)$ precomputation, we can answer each query in $O(1)$ time.

Combinations

- **Implementation**

```
typedef long long ll;

ll f[N];
// modify earlier fast exponentiation algorithm to work modulo c
ll modpow(ll a, ll b, int c);

ll inv(ll x) {
  // By Fermat's little theorem, a^(p-2) is the inverse of a mod p
  return modpow(x, MOD-2, MOD);
}

int main() {
  // factorials
  f[0] = 1;
  for (int i = 1; i < N; i++) f[i] = (i * f[i-1]) % MOD;

  int T;
  scanf("%d", &T);
  for (int i = 0; i < T; i++) {
    int n, k;
    scanf("%d%d", &n, &k);
    printf("Case %d: ", i + 1);
    ll res = (f[n] * inv(f[n-k])) % MOD;
    res = (res * inv(f[k])) % MOD;
    printf("%lld\n", res);
  }
  return 0;
}
```

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

Geometry

- Most of the useful theorems and formulas are taught in high school
    - Coordinate geometry
    - Trigonometry including sine rule and cosine rule
    - Similar triangles
    - Circle geometry

Geometry

- For simple geometry problems, usually it suffices to do a lot of scribbling
- Hard geometry problems are probably the most frustrating type of algorithm problem
- Precision is a massive problem

Floating-point arithmetic

- Try to keep things in integers as much as possible
- Avoid unnecessary divisions and other arithmetic operations
- Single precision floating point numbers fail very often, but doubles are mostly reliable
    - They are not foolproof though!
    - Going to long doubles is not a panacea

Floating-point comparisons

- When comparing floating point numbers, compare up to a small constant ϵ to allow for rounding errors

```
const double EPS = 1e-8;
if (fabs(x-y) < EPS) {
  // equal
}
```

- Many problems that require floating point answers will specify how many decimal places they want.
  If you are e.g. binary searching for an answer, that tells you how small your ϵ needs to be until you can stop.

Vectors

- Basic vector operations
    - Norm (length)
    - Rotation by some angle
    - Normal
    - Compositions of operations
    - Reflection
    - Projection

Vectors

```
#define x first
#define y second
typedef pair<double, double> pt;

struct line {
  double a, b, c;
  // coefficients in general form, compare up to constant factor
}

pt operator-(pt u, pt v) { return pt(u.x - v.x, u.y - v.y); }
pt operator+(pt u, pt v) { return pt(u.x + v.x, u.y + v.y); }
pt operator*(pt u, double d) { return pt(u.x * d, u.y * d); }

// dot product
double operator*(pt u, pt v) { return u.x * v.x + u.y * v.y; }
// norm
double operator!(pt p) { return sqrt(p * p); }
// "cross product"
double operator^(pt u, pt v) { return u.x * v.y - u.y * v.x; }
```

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

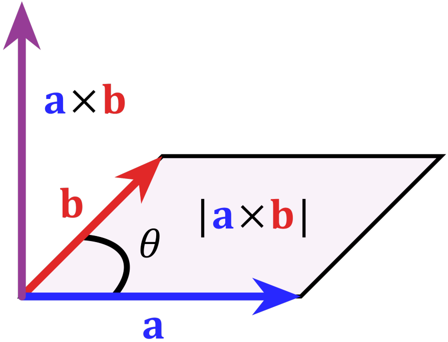
Cross product

- Incredibly useful in programming contests!
- We mostly work with 2D points, so we assume the third component is zero:

$$\begin{pmatrix} u_x \\ u_y \\ 0 \end{pmatrix} \times \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ u_x v_y - u_y v_x \end{pmatrix}$$

Cross product

- Computes the signed area of the parallelogram formed by the two input vectors
  - This is double the area of the triangle
  - The sign depends on the orientation of u and v



Applications of cross product

- Line-line intersection
- Orientation ("ccw") check
- Triangle area

Matrices

- Usually, complicated matrix operations do not come up
- Matrix multiplication or related operations are commonly used
- Solving linear systems using Gaussian elimination is sometimes used

Table of Contents

1. Number Theory
   - Primes
   - GCD
   - Modular Inverse
   - Fast Exponentiation
2. Combinatorics
3. Example problems
4. Geometry
5. Algebra
6. Example problems

Life cycle

- **Problem statement** You are given a set of integers which each describe the life cycle of a set of insects. If an insect has a life cycle of x days, a swarm of them will attack your cicadas every x days, starting from the xth day. For how many days over the first N days will your cicadas be free of attacks?
- **Input** An integer N $(1 \leq N \leq 2,000,000,000)$ and the set M of integers that describe the life cycle of the insects ($1 \leq |M| \leq 15$, and each individual life cycle will be less than 2,000,000,000)
- **Output** A single integer describing the number of attack-free days over the first N days

Life cycle

- The first thing to notice is that N can be large, so we can't even iterate over all the days we need to count
- However, the size of M is relatively small, which suggests we can use an algorithm which could even be exponential in $|M|$
- How can we reformulate the problem so that it's easier to work with?

Life cycle

- What we need to do is to find the size of the union of the sets of days where the insects will attack:
  Let $D_i = \{nx_i : n \in \mathbb{Z}^+, nx_i \leq N\}$ be the set of days on which the ith species will attack.
  Then we want to find $\left|\bigcup_{i=1}^{M} D_i\right|$.
- If we can find a way to quickly compute the intersections of the sets where two or more different insects will attack, then we can just directly use inclusion-exclusion

Life cycle

- So now the problem we want to solve is, given a set of integers, how many numbers in the range 1 to N are divisible by *all* of them?
- These are exactly the numbers divisible by the lowest common multiple of these numbers, which is just the product divided by the GCD
- Given an integer L, how many numbers in the range 1 to N are divisible by L?

Life cycle

- **Complexity** We need to iterate over all subsets of M, of which there are $2^{|M|}$, and compute the GCD of each of these subsets, which takes $O(|M| \log N)$ time, for a total complexity of $O(|M| \, 2^{|M|} \log N)$

Life cycle

- **Implementation**

```
ll res = 0;
// iterate through all bitsets of width m
for (int set = 1; set < (1<<m); set++) {
  int size = 0;
  ll l = 1;
  for (int j = 0; j < m; j++) {
    // if the jth bit is a 1, i.e. element j is included in set
    if (set & (1<<j)) {
      size++;
      l = lcm(l, a[j]);
    }
  }
  // inclusion-exclusion: add if set size is odd, subtract if even
  if (size % 2) {
    res += n / l;
  } else {
    res -= n / l;
  }
}
```

An expected value problem

- There is a unit interval AB. N cuts occur on this interval with uniform probability. What is the expected value of the length of the first segment (the one connected to A) after all cuts are made?

Advanced topics in mathematics

Some fun stuff to look at (beyond course scope):

- Grundy numbers
- Surreal numbers
- Blue/Red/Green Hackenbush
- Chinese remainder theorem
- Burnside's lemma
- Pick's theorem
- Euler's totient function
- Simpson's rule
- Minkowski sums
- Karatsuba algorithm

- Möbius inversion formula
- Shank's algorithm
- Cayley's formula
- Kirchhoff's matrix tree theorem
- Catalan numbers
- Stern-Brocot tree
- Continued fractions
- AKS
- Miller-Rabin