Review on arithmetics for ICPC

Escola de Verão
Maratona de Programação

Fidel I. Schaposnik (UNLP) - fidel.s@gmail.com
January 28, 2013

Contents

- Natural numbers
  - Finding prime numbers
  - Factorization, Euler's $\varphi(n)$ and the number of divisors of n

Contents

- Natural numbers
  - Finding prime numbers
  - Factorization, Euler's $\varphi(n)$ and the number of divisors of n
- Modular arithmetics
  - Basic operations
  - ModExp
  - GCD and its extension
  - Chinese Reminder Theorem

Contents

- Natural numbers
  - Finding prime numbers
  - Factorization, Euler's $\varphi(n)$ and the number of divisors of n
- Modular arithmetics
  - Basic operations
  - ModExp
  - GCD and its extension
  - Chinese Reminder Theorem
- Matrices
  - Notation and basic operations
  - Adjacency matrix for a graph
  - Markov chains and other linear problems
  - Systems of equations
  - Gauss-Jordan algorithm
  - A particular case: bi-diagonal matrices

Contents

- Natural numbers
  - Finding prime numbers
  - Factorization, Euler's $\varphi(n)$ and the number of divisors of n
- Modular arithmetics
  - Basic operations
  - ModExp
  - GCD and its extension
  - Chinese Reminder Theorem
- Matrices
  - Notation and basic operations
  - Adjacency matrix for a graph
  - Markov chains and other linear problems
  - Systems of equations
  - Gauss-Jordan algorithm
  - A particular case: bi-diagonal matrices
- One additional problem

Natural numbers

Recall that

$p \in \mathbb{N}$ is prime $\iff$ 1 and p are p's only divisors in $\mathbb{N}$

Given $n \in \mathbb{N}$, we may uniquely factorize it as

$$n = p_1^{e_i} \dots p_k^{e_k}$$

Natural numbers

Recall that

$p \in \mathbb{N}$ is prime $\iff$ 1 and p are p's only divisors in $\mathbb{N}$

Given $n \in \mathbb{N}$, we may uniquely factorize it as

$$n = p_1^{e_i} \dots p_k^{e_k}$$

Finding prime numbers and/or the factorization of a given number will be useful to solve problems involving:

- (completely) multiplicative functions

Natural numbers

Recall that

$p \in \mathbb{N}$ is prime $\iff$ 1 and p are p's only divisors in $\mathbb{N}$

Given $n \in \mathbb{N}$, we may uniquely factorize it as

$$n = p_1^{e_i} \dots p_k^{e_k}$$

Finding prime numbers and/or the factorization of a given number will be useful to solve problems involving:

- (completely) multiplicative functions
- divisors of a given number

Natural numbers

Recall that

$p \in \mathbb{N}$ is prime $\iff$ 1 and p are p's only divisors in $\mathbb{N}$

Given $n \in \mathbb{N}$, we may uniquely factorize it as

$$n = p_1^{e_i} \dots p_k^{e_k}$$

Finding prime numbers and/or the factorization of a given number will be useful to solve problems involving:

- (completely) multiplicative functions
- divisors of a given number
- prime numbers and factorizations in general :-)

Algorithms to find prime numbers

We would like to find all prime numbers up to a given number N (*e.g.* to factorize m we need all prime numbers up to $\sqrt{m}$).

Algorithms to find prime numbers

We would like to find all prime numbers up to a given number N (*e.g.* to factorize m we need all prime numbers up to $\sqrt{m}$).

- A very naive algorithm: for each $n \in [2, N)$, we check if n is divisible by some prime number smaller than $\sqrt{n}$ (we have already found all of them). With some minor optimizations:

```
1 p[0] = 2; P = 1;
2 for (i=3; i<N; i+=2) {
3   bool isp = true;
4   for (j=1; isp && j<P && p[j]*p[j]<=i; j++)
5     if (i%p[j] == 0) isp = false;
6   if (isp) p[P++] = i;
7 }
```

First algorithm to find prime numbers

Algorithms to find prime numbers

We would like to find all prime numbers up to a given number N (*e.g.* to factorize m we need all prime numbers up to $\sqrt{m}$).

- A very naive algorithm: for each $n \in [2, N)$, we check if n is divisible by some prime number smaller than $\sqrt{n}$ (we have already found all of them). With some minor optimizations:

```
1 p[0] = 2; P = 1;
2 for (i=3; i<N; i+=2) {
3   bool isp = true;
4   for (j=1; isp && j<P && p[j]*p[j]<=i; j++)
5     if (i%p[j] == 0) isp = false;
6   if (isp) p[P++] = i;
7 }
```

First algorithm to find prime numbers

Each number we consider may take up to $\pi(\sqrt{n}) = \mathcal{O}\left(\sqrt{n}/\ln n\right)$ operations, then this algorithm is supra-linear.

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41
42	43	44	45	46	47	48	49	50	51

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

(2)	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11
~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	21
~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~	31
~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~	41
~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~	51

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

②	③	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~	31
~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~	~~51~~

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

(2)	(3)	~~4~~	(5)	~~6~~	7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~	31
~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~	~~51~~

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

(2)	(3)	~~4~~	(5)	~~6~~	(7)	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~	31
~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~	41
~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~	~~51~~

Algorithms to find prime numbers (cont.)

- A very old algorithm: build a table with the numbers in the range $[2, N)$, and iterate through it. Each un-crossed number we find is a prime, so we can cross all its multiples in the table:

(2)	(3)	~~4~~	(5)	~~6~~	(7)	~~8~~	~~9~~	~~10~~	(11)
~~12~~	(13)	~~14~~	~~15~~	~~16~~	(17)	~~18~~	(19)	~~20~~	~~21~~
~~22~~	(23)	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	(29)	~~30~~	(31)
~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	(37)	~~38~~	~~39~~	~~40~~	(41)
~~42~~	(43)	~~44~~	~~45~~	~~46~~	(47)	~~48~~	~~49~~	~~50~~	~~51~~

Algorithms to find prime numbers (cont.)

The corresponding code looks like

```
1 memset(isp, true, sizeof(isp));
2 for (i=2; i<N; i++)
3   if (isp[i]) for (j=2*i; j<N; j+=i) isp[j] = false;
```

Sieve of Eratosthenes

Algorithms to find prime numbers (cont.)

The corresponding code looks like

```
1 memset(isp, true, sizeof(isp));
2 for (i=2; i<N; i++)
3   if (isp[i]) for (j=2*i; j<N; j+=i) isp[j] = false;
```

Sieve of Eratosthenes

This algorithm takes $\mathcal{O}(N \log \log N)$, but it can be taken to $\mathcal{O}(N)$ with some optimizations.

Factorization using the sieve

The sieve can actually hold more information

```
1 for (i=4; i<N; i+=2) p[i] = 2;
2 for (i=3; i*i<N; i+=2)
3   if (!p[i]) for (j=i*i; j<N; j+=2*i) p[j] = i;
```

Sieve of Eratosthenes, optimized and extended

Factorization using the sieve

The sieve can actually hold more information

```
1 for (i=4; i<N; i+=2) p[i] = 2;
2 for (i=3; i*i<N; i+=2)
3   if (!p[i]) for (j=i*i; j<N; j+=2*i) p[j] = i;
```

Sieve of Eratosthenes, optimized and extended

Then

```
1 int fact(int n, int f[]) {
2   int F = 0;
3   while (p[n]) {
4     f[F++] = p[n];
5     n /= p[n];
6   }
7   f[F++] = n;
8   return F;
9 }
```

Factorization using the sieve

Divisors of a given number

We can now generate the divisors of a given number recursively

```
1  int d[MAXD], D=0;
2
3  void div(int cur, int f[], int s, int e) {
4      if (s == e) d[D++] = cur;
5      else {
6          int m;
7          for (m=s+1; m<e && f[m]==f[s]; m++);
8          for (int i=s; i<=m; i++) {
9              div(cur, f, m, e);
10             cur *= f[s];
11         }
12     }
13 }
```

DFS-like algorithm to generate the divisors of a number

Remember that $f[\cdots]$ should contain the prime factors of N **in order**: use *sort* after getting them from *fact*, then call $div(1, f, 0, F)$.

Some functions from number theory

If we can factorize a given number n, we can calculate some number theory related functions:

Some functions from number theory

If we can factorize a given number n, we can calculate some number theory related functions:

- Euler's φ function: $\varphi(n)$ is the number of positive integers less than n that are coprime with n. We have

$$\varphi(n) = (p_1^{e_1} - p_1^{e_1-1}) \dots (p_k^{e_k} - p_k^{e_k-1})$$

Some functions from number theory

If we can factorize a given number n, we can calculate some number theory related functions:

- Euler's φ function: $\varphi(n)$ is the number of positive integers less than n that are coprime with n. We have

$$\varphi(n) = (p_1^{e_1} - p_1^{e_1-1}) \dots (p_k^{e_k} - p_k^{e_k-1})$$

- The number of divisors of n is

$$\sigma_0(n) = (e_1 + 1) \dots (e_k + 1)$$

  Larger numbers of divisors are achieved when there are many distinct prime factors (n different prime factors $\rightarrow \sigma(n) = 2^n$):

  $6.469.693.230 = 2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 \times 29$

  "only" has 1024 divisors.
- There are similar formulas for $\sigma_m(n) = \sum_{d|n} d^m$

Further reading

There are other, faster algorithms to find prime numbers. In particular, the sieve of Eratosthenes can be further optimized:

- in memory, to use just one bit per number;
- using a wheel to achieve better running time:

```
1 int w[8] = {4,2,4,2,4,6,2,6};
2
3 for (i=7,cur=0; i*i<N; i+=w[cur++&7]) {
4     if (!p[i]) for (j=i*i; j<N; j+=2*i) p[j] = i;
5 }
```

One very simple wheel for the sieve of Eratosthenes

You may also want to look up *Pritchard's sieve*.

Further reading

There are other, faster algorithms to find prime numbers. In particular, the sieve of Eratosthenes can be further optimized:

- in memory, to use just one bit per number;
- using a wheel to achieve better running time:

```
1 int w[8] = {4,2,4,2,4,6,2,6};
2
3 for (i=7,cur=0; i*i<N; i+=w[cur++&7]) {
4     if (!p[i]) for (j=i*i; j<N; j+=2*i) p[j] = i;
5 }
```

One very simple wheel for the sieve of Eratosthenes

You may also want to look up *Pritchard's sieve*.

Exercises will come during the week :-)

Modular arithmetic

Recall that given $a, r \in \mathbb{Z}$ and $m \in \mathbb{N}$

$$a \equiv_m r \quad \Longleftrightarrow \quad a = q.m + r \qquad \text{with} \qquad r = 0, 1, \ldots, m-1$$

Summation, substraction and multiplication operations are trivially extended and keep their well-known properties

$$a \pm b = c \quad \Longrightarrow \quad a \pm b \equiv_m c$$

$$a.b = c \quad \Longrightarrow \quad a.b \equiv_m c$$

Modular arithmetic

Recall that given $a, r \in \mathbb{Z}$ and $m \in \mathbb{N}$

$$a \equiv_m r \iff a = q.m + r \quad \text{with} \quad r = 0, 1, \ldots, m-1$$

Summation, substraction and multiplication operations are trivially extended and keep their well-known properties

$$a \pm b = c \implies a \pm b \equiv_m c$$

$$a.b = c \implies a.b \equiv_m c$$

Division is defined as the inverse of multiplication, so

$$a/b \implies a.b^{-1} \quad \text{with} \quad b.b^{-1} = 1$$

Does the inverse mod m always exist? How do we calculate it?

ModExp

Sometimes we can avoid calculating inverses altogether: *e.g.* problem *Last Digit* (MCA'07) asks us to calculate the last non-zero digit of

$$\chi = \binom{N}{m_1 \ldots m_M} = \frac{N!}{m_1! \ldots m_M!} \quad \text{with} \quad \sum_{i=1}^{M} m_i = N \quad \text{and} \quad N \leq 10^6$$

ModExp

Sometimes we can avoid calculating inverses altogether: *e.g.* problem *Last Digit* (MCA'07) asks us to calculate the last non-zero digit of

$$\chi = \binom{N}{m_1 \ldots m_M} = \frac{N!}{m_1! \ldots m_M!} \quad \text{with} \quad \sum_{i=1}^{M} m_i = N \quad \text{and} \quad N \leq 10^6$$

We may factorize χ using the sieve, then evaluate it mod 10 after eliminating any factors of 10 (*i.e.* the maximum possible number of 5's and 2's). We also need to efficiently evaluate $a^b \mod m$:

ModExp

Sometimes we can avoid calculating inverses altogether: *e.g.* problem *Last Digit* (MCA'07) asks us to calculate the last non-zero digit of

$$\chi = \binom{N}{m_1 \ldots m_M} = \frac{N!}{m_1! \ldots m_M!} \quad \text{with} \quad \sum_{i=1}^{M} m_i = N \quad \text{and} \quad N \leq 10^6$$

We may factorize χ using the sieve, then evaluate it mod 10 after eliminating any factors of 10 (*i.e.* the maximum possible number of 5's and 2's). We also need to efficiently evaluate $a^b \mod m$:

- Direct evaluation is $\mathcal{O}(b)$, which is (usually) too slow.

ModExp

Sometimes we can avoid calculating inverses altogether: *e.g.* problem *Last Digit* (MCA'07) asks us to calculate the last non-zero digit of

$$\chi = \binom{N}{m_1 \ldots m_M} = \frac{N!}{m_1! \ldots m_M!} \quad \text{with} \quad \sum_{i=1}^{M} m_i = N \quad \text{and} \quad N \leq 10^6$$

We may factorize χ using the sieve, then evaluate it mod 10 after eliminating any factors of 10 (*i.e.* the maximum possible number of 5's and 2's). We also need to efficiently evaluate $a^b \mod m$:

- Direct evaluation is $\mathcal{O}(b)$, which is (usually) too slow.
- Write b in binary, then $b = c_0.2^0 + \cdots + c_{\log b}.2^{\log b}$ and we can evaluate a^b in $\mathcal{O}(\log b)$

$$a^b = \prod_{i=0, c_i \neq 0}^{\log b} a^{2^i}$$

ModExp (code)

```
1 long long modexp(long long a, int b) {
2   long long RES = 1LL;
3   while (b > 0) {
4     if (b&1) RES = (RES*a)% MOD;
5     b >>= 1;
6     a = (a*a)% MOD;
7   }
8   return RES;
9 }
```

ModExp

Last Digit (MCA'07)

```
1  int calc(int N, int m[], int M) {
2      int i, RES;
3
4      e[N]++;
5      for (i=0; i<M; i++) if (m[i] > 1) e[m[i]]--;
6      for (i=N-1; i>=2; i--) e[i] += e[i+1];
7      for (i=N; i>=2; i--) if (p[i]) {
8          e[i/p[i]] += e[i];
9          e[p[i]] += e[i];
10         e[i] = 0;
11     }
12     int tmp = min(e[2], e[5]);
13     e[2] -= tmp; e[5] -= tmp;
14
15     RES = 1;
16     for (i=2; i<=N; i++) if (e[i] != 0) {
17         RES = (RES*modexp(i, e[i]))% MOD;
18         e[i] = 0;
19     }
20     return RES;
21 }
```

Last Digit *(MCA'07)*

GCD

The greatest common divisor between a and b is the largest d such that $d|a$ and $d|b$. Note that

$$a = q.b + r \quad \Longrightarrow \quad \gcd(a, b) = \gcd(b, r)$$

Which leads to

```
1 int gcd(int a, int b) {
2   if (b == 0) return a;
3   return gcd(b, a%b);
4 }
```

Euclid's algorithm

GCD

The greatest common divisor between a and b is the largest d such that $d|a$ and $d|b$. Note that

$$a = q.b + r \quad \Longrightarrow \quad \gcd(a, b) = \gcd(b, r)$$

Which leads to

```
1 int gcd(int a, int b) {
2   if (b == 0) return a;
3   return gcd(b, a%b);
4 }
```

Euclid's algorithm

It can be shown that $\gcd(F_{n+1}, F_n)$ takes exactly n operations (here F_n are the Fibonacci numbers). Because F_n grows exponentially and is the worst-case input for this algorithm, running time is $\mathcal{O}(\log n)$.

Extended GCD

One may prove that

$$\gcd(a, m) = 1 \iff 1 = a.x + m.y$$

Extended GCD

One may prove that

$$\gcd(a, m) = 1 \quad \Longleftrightarrow \quad 1 = a.x + m.y$$

Then $x \equiv_m a^{-1}$, so that a has an inverse mod m if and only if $\gcd(a, m) = 1$. [Corolary: $\mathbb{Z}_p$ is a field.]

Extended GCD

One may prove that

$$\gcd(a, m) = 1 \iff 1 = a.x + m.y$$

Then $x \equiv_m a^{-1}$, so that a has an inverse mod m if and only if $\gcd(a, m) = 1$. [Corolary: $\mathbb{Z}_p$ is a field.] To find x and y, we trace them along Euclid's algorithm:

```
1  pii egcd(int a, int b) {
2    if (b == 0) return make_pair(1, 0);
3    else {
4      pii RES = egcd(b, a%b);
5      return make_pair(RES.second,RES.first-RES.second*(a/b));
6    }
7  }
8
9  int inv(int n, int m) {
10   pii EGCD = egcd(n, m);
11   return ( (EGCD.first% m)+m)% m;
12 }
```

Extension of Euclid's algorithm; inverse mod m

Chinese Remainder Theorem

Given a set of conditions

$$x \equiv a_i \mod n_i \quad \text{for} \quad i = 1, \ldots, k \quad \text{with} \quad \gcd(n_i, n_j) = 1 \quad \forall i \neq j$$

there is a unique $x \mod N = n_1 \ldots n_k$ satisfying all these conditions simultaneously.

Chinese Remainder Theorem

Given a set of conditions

$$x \equiv a_i \mod n_i \quad \text{for} \quad i = 1, \ldots, k \quad \text{with} \quad \gcd(n_i, n_j) = 1 \quad \forall i \neq j$$

there is a unique $x \mod N = n_1 \ldots n_k$ satisfying all these conditions simultaneously. We may construct it considering

$$m_i = \prod_{j \neq i} n_j \qquad \Longrightarrow \qquad \gcd(n_i, m_i) = 1$$

Let $\bar{m}_i = m_i^{-1} \mod n_i$, then

$$x \equiv \sum_{i=1}^{k} \bar{m}_i m_i a_i \mod N$$

Chinese reminder theorem (code)

```
1  int crt(int n[], int a[], int k) {
2    int i, tmp, MOD, RES;
3
4    MOD = 1;
5    for (i=0; i<k; i++) MOD *= n[i];
6
7    RES = 0;
8    for (i=0; i<k; i++) {
9      tmp = MOD/n[i];
10     tmp *= inv(tmp, n[i]);
11     RES += (tmp*a[i])% MOD;
12   }
13   return RES% MOD;
14 }
```

Chinese Reminder Theorem

Matrices

An $N \times M$ matrix is an array of N rows and M columns of elements. A natural way to define the sum and the difference of matrices is ($\mathcal{O}(N.M)$):

$$A \pm B = C \qquad \Longleftrightarrow \qquad C_{ij} = A_{ij} \pm B_{ij}$$

The product of matrices is defined as ($\mathcal{O}(N.M.L)$)

$$A_{N \times M} \cdot B_{M \times L} = C_{N \times L} \qquad \Longleftrightarrow \qquad C_{ij} = \sum_{k=1}^{M} A_{ik}.B_{kj}$$

Matrices

An $N \times M$ matrix is an array of N rows and M columns of elements. A natural way to define the sum and the difference of matrices is ($\mathcal{O}(N.M)$):

$$A \pm B = C \qquad \Longleftrightarrow \qquad C_{ij} = A_{ij} \pm B_{ij}$$

The product of matrices is defined as ($\mathcal{O}(N.M.L)$)

$$A_{N\times M} \cdot B_{M\times L} = C_{N\times L} \qquad \Longleftrightarrow \qquad C_{ij} = \sum_{k=1}^{M} A_{ik}.B_{kj}$$

For square matrices (from now on, assume we are working with $N \times N$), it makes sense to ask if a given matrix A has an inverse. One finds that if $\det A \neq 0$, the inverse exists and satisfies

$$A \cdot A^{-1} = \mathbb{1} = A^{-1} \cdot A$$

Matrices (cont.)

We represent matrices using bi-dimensional arrays. In C++, to use a matrix as an argument of a function, it is convenient to define a list of pointers to avoid fixing one of the dimensions in the definition of the function

```
1  type function(int **A, int N, int M) {
2    ...
3  }
4
5  int main() {
6    int a[MAXN][MAXN], *pa[MAXN];
7
8    for (int i=0; i<MAXN; i++) pa[i] = a[i];
9    ...
10   function(pa, N, M);
11   ...
12 }
```

Pointer list to represent a matrix

Adjacency matrix

We may use a matrix to represent the edges of a graph.

Adjacency matrix

We may use a matrix to represent the edges of a graph. For a graph of N nodes, a matrix $A_{N \times N}$ can have A_{ij}:

- the weight of the edge going from node i to node j (or ∞ if this edge does not exist);

Adjacency matrix

We may use a matrix to represent the edges of a graph. For a graph of N nodes, a matrix $A_{N \times N}$ can have A_{ij}:

- the weight of the edge going from node i to node j (or ∞ if this edge does not exist);
- the number of edges that go from node i to node j (0 if there are none).

In the latter case,

$$(A^2)_{ij} = \sum_k A_{ik} A_{kj} \quad \Longrightarrow \quad \text{paths } i \to j \text{ with exactly 2 edges}$$

and in general A^n contains the number of paths between the pairs of nodes of the original graph with exactly n edges.

Adjacency matrix

We may use a matrix to represent the edges of a graph. For a graph of N nodes, a matrix $A_{N \times N}$ can have A_{ij}:

- the weight of the edge going from node i to node j (or ∞ if this edge does not exist);
- the number of edges that go from node i to node j (0 if there are none).

In the latter case,

$$(A^2)_{ij} = \sum_k A_{ik} A_{kj} \quad \Longrightarrow \quad \text{paths } i \to j \text{ with exactly 2 edges}$$

and in general A^n contains the number of paths between the pairs of nodes of the original graph with exactly n edges.
We may calculate A^n using an adapted version of *ModExp* in $\mathcal{O}(N^3 \log n)$.

Linear recurrences

A linear recurrence of order K defines a sequence $\{a_n\}$ through

$$a_n = \sum_{k=1}^{K} c_k a_{n-k}$$

and a vector $\vec{a}_K = (a_K, a_{K-1}, \ldots, a_1)$ of initial values.

Linear recurrences

A linear recurrence of order K defines a sequence $\{a_n\}$ through

$$a_n = \sum_{k=1}^{K} c_k a_{n-k}$$

and a vector $\vec{a}_K = (a_K, a_{K-1}, \ldots, a_1)$ of initial values.

Let $\vec{a}_n$ be a vector containing a_n and its $K-1$ previous elements. Then the recurrence can be represented as

$$\vec{a}_n = R\,\vec{a}_{n-1} \quad \Longleftrightarrow \quad \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_{n-K+2} \\ a_{n-K+1} \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & \cdots & c_K \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_{n-K+1} \\ a_{n-K} \end{pmatrix}$$

So we can calculate the N-th term using $\vec{a}_N = R^{N-K}\,\vec{a}_K$

Markov chains

Let $\{S_i\}$ be a set of states, and p_{ij} the known probabilities to observe a transition from state i to state j. Terminal states $\{S_k\}$ are such that $\sum_i p_{ki} = 0$. What is the expected number of transitions E_i needed to reach a terminal state from state S_i?

Markov chains

Let $\{S_i\}$ be a set of states, and p_{ij} the known probabilities to observe a transition from state i to state j. Terminal states $\{S_k\}$ are such that $\sum_i p_{ki} = 0$. What is the expected number of transitions E_i needed to reach a terminal state from state S_i? For terminal states, we clearly have

$$E_k = 0$$

For the rest,

$$E_i = 1 + \sum_j p_{ij}.E_j$$

Markov chains

Let $\{S_i\}$ be a set of states, and p_{ij} the known probabilities to observe a transition from state i to state j. Terminal states $\{S_k\}$ are such that $\sum_i p_{ki} = 0$. What is the expected number of transitions E_i needed to reach a terminal state from state S_i? For terminal states, we clearly have

$$E_k = 0$$

For the rest,

$$E_i = 1 + \sum_j p_{ij}.E_j$$

In other words, we need to solve a system of equations over the expected number of transitions E_i.

Other linear problems

Some linear systems of equations can be found in computational geometry problems. As an example, something you may need later today: plane-plane intersection ;-)

Other linear problems

Some linear systems of equations can be found in computational geometry problems. As an example, something you may need later today: plane-plane intersection ;-)

A plane is defined by a point $\vec{p_0}$ on it and a vector $\vec{n}_0$ which is orthogonal to the plane. All other points on the plane satisfy

$$(\vec{p} - \vec{p}_0) \cdot \vec{n}_0 = 0$$

which leads to the familiar equation

$$ax + by + cz + d = 0 \quad \text{for} \quad \vec{n}_0 = (a, b, c) \quad \text{and} \quad d = -\vec{p}_0 \cdot \vec{n}_0$$

In the general case, two planes defined by $\{\vec{p}_1, \vec{n}_1\}$ and $\{\vec{p}_2, \vec{n}_2\}$ intersect to give a line. How do we find this line?

Other linear problems (cont.)

A line is defined by a point $\vec{p}$ on it and a directing vector $\vec{v}$ pointing along its direction (either way).

Other linear problems (cont.)

A line is defined by a point $\vec{p}$ on it and a directing vector $\vec{v}$ pointing along its direction (either way).

Our line should belong to both planes, so it is orthogonal to both normal vectors $\vec{n}_1$ and $\vec{n}_2$. Then

$$\vec{v} = \vec{n}_1 \wedge \vec{n}_2$$

If $\vec{v} = \vec{0}$, the planes are parallel and there is no intersection. But how do we find $\vec{p}$?

Other linear problems (cont.)

A line is defined by a point $\vec{p}$ on it and a directing vector $\vec{v}$ pointing along its direction (either way).

Our line should belong to both planes, so it is orthogonal to both normal vectors $\vec{n}_1$ and $\vec{n}_2$. Then

$$\vec{v} = \vec{n}_1 \wedge \vec{n}_2$$

If $\vec{v} = \vec{0}$, the planes are parallel and there is no intersection. But how do we find $\vec{p}$?

Let $\vec{p}$ be that point on the line which is closest to some fixed, arbitrary point (*e.g.* the origin). Then $\vec{p} = (x, y, z)$ minimizes

$$D^2 = x^2 + y^2 + z^2$$

while being on both planes, *i.e.* satisfying

$$(\vec{p} - \vec{p}_1) \cdot \vec{n}_1 = 0 \quad \text{and} \quad (\vec{p} - \vec{p}_2) \cdot \vec{n}_2 = 0$$

Other linear problems (cont.)

We can find the point $\vec{p}$ minimizing D^2 under these constraints using Lagrange multipliers. Define

$$f(x, y, z, \mu_1, \mu_2) = x^2 + y^2 + z^2 + \mu_1 (\vec{p} - \vec{p}_1) \cdot \vec{n}_1 + \mu_2 (\vec{p} - \vec{p}_2) \cdot \vec{n}_2$$

Then we must ask

$$\frac{\partial f}{\partial x} = 2x + \mu_1 n_{1x} = 0 \qquad \frac{\partial f}{\partial y} = 2y + \mu_1 n_{1y} = 0 \qquad \frac{\partial f}{\partial z} = 2z + \mu_1 n_{1z} = 0$$

$$\frac{\partial f}{\partial \mu_1} = (\vec{p} - \vec{p}_1) \cdot \vec{n}_1 = 0 \qquad \frac{\partial f}{\partial \mu_2} = (\vec{p} - \vec{p}_2) \cdot \vec{n}_2 = 0$$

Other linear problems (cont.)

We can find the point $\vec{p}$ minimizing D^2 under these constraints using Lagrange multipliers. Define

$$f(x, y, z, \mu_1, \mu_2) = x^2 + y^2 + z^2 + \mu_1 (\vec{p} - \vec{p}_1) \cdot \vec{n}_1 + \mu_2 (\vec{p} - \vec{p}_2) \cdot \vec{n}_2$$

Then we must ask

$$\frac{\partial f}{\partial x} = 2x + \mu_1 n_{1x} = 0 \qquad \frac{\partial f}{\partial y} = 2y + \mu_1 n_{1y} = 0 \qquad \frac{\partial f}{\partial z} = 2z + \mu_1 n_{1z} = 0$$

$$\frac{\partial f}{\partial \mu_1} = (\vec{p} - \vec{p}_1) \cdot \vec{n}_1 = 0 \qquad \frac{\partial f}{\partial \mu_2} = (\vec{p} - \vec{p}_2) \cdot \vec{n}_2 = 0$$

Or in matrix notation

$$\begin{pmatrix} 2 & 0 & 0 & n_{1x} & n_{2x} \\ 0 & 2 & 0 & n_{1y} & n_{2y} \\ 0 & 0 & 2 & n_{1z} & n_{2z} \\ n_{1x} & n_{1y} & n_{1z} & 0 & 0 \\ n_{2x} & n_{2y} & n_{2z} & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \mu_1 \\ \mu_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vec{p}_1 \cdot \vec{n}_1 \\ \vec{p}_2 \cdot \vec{n}_2 \end{pmatrix}$$

Systems of equations

A system of equations over N variables is

$$\begin{aligned} a_{11}\,x_1 + \cdots + a_{1N}\,x_N &= b_1 \\ &\vdots \\ a_{N1}\,x_1 + \cdots + a_{N}\,x_N &= b_N \end{aligned}$$

It can be represented by matrices as

$$A\,\vec{x} = \vec{b} \qquad \Longleftrightarrow \qquad \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$$

Systems of equations

A system of equations over N variables is

$$\begin{aligned} a_{11}\,x_1 + \cdots + a_{1N}\,x_N &= b_1 \\ &\vdots \\ a_{N1}\,x_1 + \cdots + a_{N}\,x_N &= b_N \end{aligned}$$

It can be represented by matrices as

$$A\,\vec{x} = \vec{b} \qquad \Longleftrightarrow \qquad \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$$

Solving this system reduces to finding the inverse A^{-1}, because then $\vec{x} = A^{-1}\vec{b}$. We can solve many systems of equations with different independent terms $\vec{b_i}$ by building a matrix B with the vectors $\vec{b_i}$ as its columns. Then $X = A^{-1}B$, and in particular if $B \mapsto \mathbb{1}$, $X \mapsto A^{-1}$.

Systems of equations (cont.)

To solve a system by hand, we solve for an arbitrary variable in one of the equations, and then use this to eliminate that variable from every other equation, while working simultaneously with the independent terms. In order to do this, we may:

- Multiply or divide an equation (row) by any number.
- Sum or subtract an equation (row) to another one.
- Swap two rows (leaves the equations unchanged)

Systems of equations (cont.)

To solve a system by hand, we solve for an arbitrary variable in one of the equations, and then use this to eliminate that variable from every other equation, while working simultaneously with the independent terms. In order to do this, we may:

- Multiply or divide an equation (row) by any number.
- Sum or subtract an equation (row) to another one.
- Swap two rows (leaves the equations unchanged)

Gauss-Jordan's algorithm is a formalization of this procedure, with one caveat: to reduce the numerical error, in each step the corresponding variable is solved from the equation where it appears with the largest coefficient in absolute value (this is called *pivoting*).

Gauss-Jordan elimination

```
1  bool invert(double **A, double **B, int N) {
2    int i, j, k, jmax; double tmp;
3    for (i=1; i<=N; i++) {
4      jmax = i; //Largest el. in A at col. i with row >= i
5      for (j=i+1; j<=N; j++)
6        if (abs(A[j][i]) > abs(A[jmax][i])) jmax = j;
7
8      for (j=1; j<=N; j++) {//Swap rows i and jmax
9        swap(A[i][j], A[jmax][j]); swap(B[i][j], B[jmax][j]);
10         }
11
12     //Check this matrix is invertible
13     if (abs(A[i][i]) < EPS) return false;
14
15     tmp = A[i][i];//Normalize row i
16     for (j=1; j<=N; j++) { A[i][j] /= tmp; B[i][j] /= tmp; }
17
18     //Eliminate non-zero values in column i
19     for (j=1; j<=N; j++) {
20       if (i == j) continue;
21       tmp = A[j][i];
22       for (k=1; k<=N; k++) {
23         A[j][k] -= A[i][k]*tmp; B[j][k] -= B[i][k]*tmp;
24       }
25     }
26   }
27   return true;
28 }
```

Gauss-Jordan elimination

Gauss-Jordan elimination for bi-diagonal matrices

Gauss-Jordan's algorithm is clearly $\mathcal{O}(N^3)$. It can be seen that if we can multiply two $N \times N$ matrices in $\mathcal{O}(T(N))$, then we can invert a matrix or calculate its determinant in the same asymptotic time.

Gauss-Jordan elimination for bi-diagonal matrices

Gauss-Jordan's algorithm is clearly $\mathcal{O}(N^3)$. It can be seen that if we can multiply two $N \times N$ matrices in $\mathcal{O}(T(N))$, then we can invert a matrix or calculate its determinant in the same asymptotic time.

Usually, instead of optimizing the general algorithm it is a lot more convenient to take advantage of some special property of the matrices we would like to invert: for instance, we can invert a bi-diagonal or tri-diagonal matrix (with non-zero diagonal elements) in $\mathcal{O}(N)$.

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \dots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \dots & 0 \\ \vdots & & & & & & \vdots \\ 0 & & \dots & & 0 & a_{NN-1} & a_{NN} \end{pmatrix}$$

Further reading

- There are more efficient algorithms that you can use to multiply two matrices (Strassen's is $\mathcal{O}(n^{2,807})$, Coppersmith–Winograd's is $\mathcal{O}(n^{2,376})$), but they are not necessarily convenient for a competition...
- There are many important decompositions of a matrix (*i.e.* ways to write it as a sum or product of some other matrices with special properties). These play an important role in many algorithms, such as those that are used to calculate determinants.
- There are many ways to generalize the treatment of linear recurrences: one can work with sets of multiple dependent recurrences, inhomogeneous recurrences, *etc.*

An additional problem

Bases (SARC'08): Analyze in which bases an expression such as $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ is true.

An additional problem

Bases (SARC'08): Analyze in which bases an expression such as $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ is true.

Things we have not covered:

- Polynomials (evaluation, operations, properties, *etc.*).
- Evaluation of mathematical expressions (parsing).

An additional problem

Bases (SARC'08): Analyze in which bases an expression such as $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ is true.

Things we have not covered:

- Polynomials (evaluation, operations, properties, *etc.*).
- Evaluation of mathematical expressions (parsing).

Maybe next time ;-)