Aritmética para ICPC

ACM ICPC Training Camp
Departamento de Computación, UBA

Fidel I. Schaposnik (UNLP) - `fidel.s@gmail.com`

3 de agosto de 2011

Contenidos

- Números naturales
  - Algoritmos para encontrar números primos
  - Factorización, φ y cantidad de divisores

Contenidos

- Números naturales
  - Algoritmos para encontrar números primos
  - Factorización, φ y cantidad de divisores
- Aritmética modular
  - Operaciones básicas
  - ModExp
  - GCD y su extensión
  - Teorema chino del resto

Contenidos

Contenidos

Números naturales

Recordamos que

$p \in \mathbb{N}$ es primo $\iff$ 1 y p son los únicos divisores de p en $\mathbb{N}$

Dado $n \in \mathbb{N}$, podemos factorizarlo de manera única como

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

Números naturales

Recordamos que

$p \in \mathbb{N}$ es primo $\iff$ 1 y p son los únicos divisores de p en $\mathbb{N}$

Dado $n \in \mathbb{N}$, podemos factorizarlo de manera única como

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

Encontrar números primos y/o la factorización de un número natural será útil para resolver problemas que involucran:

- funciones [completamente] multiplicativas

Números naturales

Recordamos que

$p \in \mathbb{N}$ es primo $\iff$ 1 y p son los únicos divisores de p en $\mathbb{N}$

Dado $n \in \mathbb{N}$, podemos factorizarlo de manera única como

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

Encontrar números primos y/o la factorización de un número natural será útil para resolver problemas que involucran:

- funciones [completamente] multiplicativas
- divisores de un número

Números naturales

Recordamos que

$p \in \mathbb{N}$ es primo $\iff$ 1 y p son los únicos divisores de p en $\mathbb{N}$

Dado $n \in \mathbb{N}$, podemos factorizarlo de manera única como

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

Encontrar números primos y/o la factorización de un número natural será útil para resolver problemas que involucran:

- funciones [completamente] multiplicativas
- divisores de un número
- números primos y factorizaciones :-)

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor (por ejemplo, para factorizar m necesitamos todos los números primos hasta $\sqrt{m}$).

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor (por ejemplo, para factorizar m necesitamos todos los números primos hasta $\sqrt{m}$).

- Un algoritmo ingenuo: para cada $n \in [2, MAXN)$, analizamos si es divisible por algún primo menor que $\sqrt{n}$ de los ya encontrados. Con algunas optimizaciones:

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

Primer algoritmo para encontrar primos

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor (por ejemplo, para factorizar m necesitamos todos los números primos hasta $\sqrt{m}$).

- Un algoritmo ingenuo: para cada $n \in [2, MAXN)$, analizamos si es divisible por algún primo menor que $\sqrt{n}$ de los ya encontrados. Con algunas optimizaciones:

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

Primer algoritmo para encontrar primos

Cada número requiere tiempo $\pi(\sqrt{n}) = \mathcal{O}\left(\sqrt{n}/\ln n\right)$, luego el algoritmo es supralineal.

Algoritmos para encontrar números primos (cont.)

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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

Algoritmos para encontrar números primos (cont.)

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo antiguo: armamos una tabla con los números $[2, MAXN)$, y los recorremos en orden. Cada número que encontramos que todavía no fue tachado es un primo, de modo que podemos tachar todos sus múltiplos:

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

Algoritmos para encontrar números primos (cont.)

El código correspondiente es

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

Criba de Eratóstenes

Algoritmos para encontrar números primos (cont.)

El código correspondiente es

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

Criba de Eratóstenes

El tiempo de ejecución es $\mathcal{O}(N \log\log N)$, y puede ser llevado a $\mathcal{O}(N)$ con algunas optimizaciones.

Factorización usando la criba

La criba puede guardar más información:

```
memset(p, -1, sizeof(p));
for (i=4; i<MAXN; i+=2) p[i] = 2;
for (i=3; i*i<MAXN; i+=2)
  if (p[i] == -1)
    for (j=i*i; j<MAXN; j+=2*i)
      p[j] = i;
```

Criba de Eratóstenes extendida y optimizada

Factorización usando la criba

La criba puede guardar más información:

```
memset(p, -1, sizeof(p));
for (i=4; i<MAXN; i+=2) p[i] = 2;
for (i=3; i*i<MAXN; i+=2)
  if (p[i] == -1)
    for (j=i*i; j<MAXN; j+=2*i)
      p[j] = i;
```

Criba de Eratóstenes extendida y optimizada

Y entonces

```
int fact(int n, int f[]) {
  int F = 0;
  while (p[n] != -1) {
    f[F++] = p[n];
    n /= p[n];
  }
  f[F++] = n;
  return F;
}
```

Factorización usando la criba

Teniendo la factorización de un número n, podemos generar sus divisores, o calcular funciones de teoría de números:

Funciones de teoría de números

Teniendo la factorización de un número n, podemos generar sus divisores, o calcular funciones de teoría de números:

- Función φ de Euler: $\varphi(n)$ es la cantidad de números menores o iguales que n que son coprimos con n. Se tiene

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

Funciones de teoría de números

Teniendo la factorización de un número n, podemos generar sus divisores, o calcular funciones de teoría de números:

- Función φ de Euler: $\varphi(n)$ es la cantidad de números menores o iguales que n que son coprimos con n. Se tiene

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

- La cantidad de divisores de n es

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

(y fórmulas parecidas para $\sigma_m(n) = \sum_{d|n} d^m$)

Ejercicios (1)

Algunos problemas para ir fijando ideas:

- SPOJ, p.2 *Prime Generator*: Encontrar todos los primos en el intervalo $[M, N]$ con $1 \leq M \leq N \leq 10^9$ y $N - M \leq 10^5$.
- SPOJ, p.526 *Divisors*: Encontrar todos los N tales que $\sigma_0(N) = p.q$ con $p \neq q$ primos y $N \leq 10^6$.
- CodeJam 2011, R2 *Expensive Dinner*: Calcular $\sum_{i=1}^{k} e_k - 1$ para la factorización de $MCM(1, \ldots, N)$ con $N \leq 10^{12}$.

Ejercicios (1)

Algunos problemas para ir fijando ideas:

- SPOJ, p.2 *Prime Generator*: Encontrar todos los primos en el intervalo $[M, N]$ con $1 \leq M \leq N \leq 10^9$ y $N - M \leq 10^5$.
- SPOJ, p.526 *Divisors*: Encontrar todos los N tales que $\sigma_0(N) = p.q$ con $p \neq q$ primos y $N \leq 10^6$.
- CodeJam 2011, R2 *Expensive Dinner*: Calcular $\sum_{i=1}^{k} e_k - 1$ para la factorización de $MCM(1, \ldots, N)$ con $N \leq 10^{12}$.

Un poco de teoría de números:

- SPOJ, p.5971 *LCM Sum*: Calcular ($\leq$ 300000 veces) $\sum_{n=1}^{N} lcm(i, n)$ con $N \leq 10^6$.
- SER'08, p.H *GCD Determinant*: Dado $\{x_1, \ldots, x_N\}$, calcular $\det S$ con $S_{ij} = \gcd(x_i, x_j)$ y $N \leq 10^3$.

Aritmética modular

Recordamos que dados $a \in \mathbb{Z}$ y $m \in \mathbb{N}$

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

Las operaciones de suma, resta y producto se extienden trivialmente, y mantienen las propiedades conocidas

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

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

Aritmética modular

Recordamos que dados $a \in \mathbb{Z}$ y $m \in \mathbb{N}$

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

Las operaciones de suma, resta y producto se extienden trivialmente, y mantienen las propiedades conocidas

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

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

La división se define como la inversa del producto, es decir que

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

¿Siempre existe el inverso módulo m? ¿Cómo podemos calcularlo?

Modexp

A veces podemos directamente evitar buscar los inversos: en MCA'07, p.C *Last Digit*, nos piden calcular el ultimo dígito no nulo de

$$\chi = \binom{N}{m_1 \dots m_M} = \frac{N!}{m_1! \dots m_M!} \quad \text{con} \quad \sum_{i=1}^{M} m_i = N \quad \text{y} \quad N \leq 10^6$$

Modexp

A veces podemos directamente evitar buscar los inversos: en MCA'07, p.C *Last Digit*, nos piden calcular el ultimo dígito no nulo de

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

Podemos factorizar χ usando lo que ya aprendimos, y evaluarlo módulo 10 eliminando antes todos los factores 5 y una cantidad igual de factores 2. Necesitamos evaluar eficientemente $a^b \bmod m$:

Modexp

A veces podemos directamente evitar buscar los inversos: en MCA'07, p.C *Last Digit*, nos piden calcular el ultimo dígito no nulo de

$$\chi = \binom{N}{m_1 \dots m_M} = \frac{N!}{m_1! \dots m_M!} \quad \text{con} \quad \sum_{i=1}^{M} m_i = N \quad \text{y} \quad N \leq 10^6$$

Podemos factorizar χ usando lo que ya aprendimos, y evaluarlo módulo 10 eliminando antes todos los factores 5 y una cantidad igual de factores 2. Necesitamos evaluar eficientemente $a^b \bmod m$:

- La evaluación directa es $\mathcal{O}(b)$, que es demasiado lento.

Modexp

A veces podemos directamente evitar buscar los inversos: en MCA'07, p.C *Last Digit*, nos piden calcular el ultimo dígito no nulo de

$$\chi = \binom{N}{m_1 \dots m_M} = \frac{N!}{m_1! \dots m_M!} \quad \text{con} \quad \sum_{i=1}^{M} m_i = N \quad \text{y} \quad N \leq 10^6$$

Podemos factorizar χ usando lo que ya aprendimos, y evaluarlo módulo 10 eliminando antes todos los factores 5 y una cantidad igual de factores 2. Necesitamos evaluar eficientemente a^b mod m:

- La evaluación directa es $\mathcal{O}(b)$, que es demasiado lento.
- Si escribimos a b en binario, $b = c_0.2^0 + \cdots + c_{\log b}.2^{\log b}$, podemos evaluar a^b en $\mathcal{O}(\log b)$

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

Modexp (código)

```
tint modexp(tint a, tint b) {
  tint RES = 1;
  while (b > 0) {
    if ((b&1) == 1) RES = (RES*a)% MOD;
    b >>= 1;
    a = (a*a)% MOD;
  }
  return RES;
}
```

Modexp

```
int calc(int N, int m[], int M) {
  int i, RES;

  memset(e, 0, sizeof(e));
  e[N]++;
  for (i=0; i<M; i++) if (m[i] > 1) e[m[i]]--;
  for (i=MAXN-2; i>=0; i--) e[i] += e[i+1];

  RES = 1;
  for (i=MAXN-1; i>=0; i--)
    if (p[i] != -1) {
      e[i/p[i]] += e[i];
      e[p[i]] += e[i];
      e[i] = 0;
    }
  e[2] -= e[5]; e[5] = 0;

  for (i=2; i<MAXN; i++)
    if (e[i] != 0) RES = (RES*modexp(i, e[i]))% MOD;
  return RES;
}
```

MCA'07, p.C Last Digit

GCD

El máximo común divisor entre a y b es es el mayor d tal que $d|a$ y $d|b$. Observamos que

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

Y tenemos entonces

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

Algoritmo de Euclides

GCD

El máximo común divisor entre a y b es es el mayor d tal que $d|a$ y $d|b$. Observamos que

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

Y tenemos entonces

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

Algoritmo de Euclides

Puede verse que $\gcd(F_{n+1}, F_n)$ requiere exactamente n operaciones (siendo F_n los números de Fibonacci). Como los F_n crecen exponencialmente, y son la peor entrada posible para el algoritmo, el tiempo es $\mathcal{O}(\log n)$.

Extensión del GCD

Puede verse que

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

Extensión del GCD

Puede verse que

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

Entonces $x \equiv_m a^{-1}$, de modo que a tiene inverso módulo m si y sólo si $\gcd(a, m) = 1$. [Corolario: $\mathbb{Z}_p$ es un cuerpo.]

Extensión del GCD

Puede verse que

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

Entonces $x \equiv_m a^{-1}$, de modo que a tiene inverso módulo m si y sólo si $\gcd(a, m) = 1$. [Corolario: $\mathbb{Z}_p$ es un cuerpo.] Para encontrar x e y, los rastreamos a través del algoritmo de Euclides:

```
pii egcd(int a, int b) {
  if (b == 0) return make_pair(1, 0);
  else {
    pii RES = egcd(b, a%b);
    return make_pair(RES.second,RES.first-RES.second*(a/b));
  }
}

int inv(int n, int m) {
  pii EGCD = egcd(n, m);
  return ( (EGCD.first% m)+m)% m;
}
```

Algoritmo de Euclides extendido e inverso módulo m

Teorema chino del resto

Dado un conjunto de condiciones

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

existe un único $x \mod N = n_1 \ldots n_k$ que satisface todas las ecuaciones simultáneamente.

Teorema chino del resto

Dado un conjunto de condiciones

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

existe un único $x \mod N = n_1 \ldots n_k$ que satisface todas las ecuaciones simultáneamente. Podemos construirlo considerando

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

Llamando $\bar{m}_i = m_i^{-1} \mod n_i$, armamos

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

Teorema chino del resto (código)

```
int tcr(int n[], int a[], int k) {
  int i, tmp, MOD, RES;

  MOD = 1;
  for (i=0; i<k; i++) MOD *= n[i];

  RES = 0;
  for (i=0; i<k; i++) {
    tmp = MOD/n[i];
    tmp *= inv(tmp, n[i]);
    RES += (tmp*a[i])% MOD;
  }
  return RES% MOD;
}
```

Teorema chino del resto

Ejercicios (2)

- TCO'10 Round 1, p.2 *TwoRegisters*: Muchas veces el algoritmo de GCD aparece en problemas que no tienen demasiado que ver con teoría de números ;-)

- CEPC'08, p.I *Counting heaps*: Calcular el número (módulo M) de asignaciones de los valores $\{1, \ldots, N\}$ a los $N \leq 5.10^5$ nodos de un árbol que respetan la condición de min-heap.

- WF Warmup I, p.C *Code Feat*: Aplicar el teorema chino del resto con $k \leq 9$ y $a_i \in \left\{a_i^{(1)}, \ldots, a_i^{(A_i)}\right\}$ siendo $A_i \leq 100$.

Matrices

Una matriz de $N \times M$ es un arreglo de N filas y M columnas de elementos. Podemos definir la suma y la resta de matrices en forma natural ($\mathcal{O}(N.M)$):

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

El producto de matrices se define como ($\mathcal{O}(N.M.L)$)

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

Matrices

Una matriz de $N \times M$ es un arreglo de N filas y M columnas de elementos. Podemos definir la suma y la resta de matrices en forma natural ($\mathcal{O}(N.M)$):

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

El producto de matrices se define como ($\mathcal{O}(N.M.L)$)

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

Para matrices cuadradas (a partir de ahora, trabajamos en $N \times N$), tiene sentido preguntarse si existe la inversa multiplicativa de una matriz A. Resulta que si $\det A \neq 0$, la inversa existe y se tiene

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

Matrices (cont.)

Representamos matrices usando arreglos bidimensionales, pero para pasar una matriz como argumento a una función conviene definir una lista de punteros, así evitamos tener que fijar una de las dimensiones en la definición de la función

```
tipo funcion(int **A, int N, int M) {
  ...
}

int main() {
  int a[MAXN][MAXN],  *ra[MAXN];
  for (int i=0; i<MAXN; i++) ra[i] = a[i];
  ...
  funcion(ra, N, M);
  ...
}
```

Lista de punteros que referencia a una matriz

Matriz de adyacencias

Uno de los usos que podemos darle a las matrices es el de representar las aristas de un grafo.

Matriz de adyacencias

Uno de los usos que podemos darle a las matrices es el de representar las aristas de un grafo. Para una grafo de N nodos, una matriz $A_{N \times N}$ puede tener en A_{ij}:

- el costo de la arista que va del nodo i al j (∞ si la arista no existe).

Matriz de adyacencias

Uno de los usos que podemos darle a las matrices es el de representar las aristas de un grafo. Para una grafo de N nodos, una matriz $A_{N \times N}$ puede tener en A_{ij}:

- el costo de la arista que va del nodo i al j (∞ si la arista no existe).
- la cantidad de aristas que van del nodo i al j (0 si no hay).

Matriz de adyacencias

Uno de los usos que podemos darle a las matrices es el de representar las aristas de un grafo. Para una grafo de N nodos, una matriz $A_{N \times N}$ puede tener en A_{ij}:

- el costo de la arista que va del nodo i al j (∞ si la arista no existe).
- la cantidad de aristas que van del nodo i al j (0 si no hay).

En este último caso, $(A^2)_{ij} = \sum_k A_{ik} A_{kj}$ es la cantidad de caminos con exactamente dos aristas que van del nodo i al j. Esto puede generalizarse para A^n, que entonces contiene la cantidad de caminos con exactamente n aristas entre los pares de nodos del grafo original.

Matriz de adyacencias

Uno de los usos que podemos darle a las matrices es el de representar las aristas de un grafo. Para una grafo de N nodos, una matriz $A_{N \times N}$ puede tener en A_{ij}:

- el costo de la arista que va del nodo i al j (∞ si la arista no existe).
- la cantidad de aristas que van del nodo i al j (0 si no hay).

En este último caso, $(A^2)_{ij} = \sum_k A_{ik} A_{kj}$ es la cantidad de caminos con exactamente dos aristas que van del nodo i al j. Esto puede generalizarse para A^n, que entonces contiene la cantidad de caminos con exactamente n aristas entre los pares de nodos del grafo original.

Podemos calcular A^n usando una version adaptada de *modexp* en $\mathcal{O}(N^3 \log n)$. Hay algoritmos más eficientes para multiplicar (el algoritmo de Strassen es $\mathcal{O}(n^{2,807})$, y el de Coppersmith-Winograd es $\mathcal{O}(n^{2,376})$), pero no necesariamente conviene usarlos en una competencia...

Cadenas de Markov

Si tenemos un sistema con un conjunto de estados $\{S_i\}$, con probabilidad p_{ij} conocida de efectuar una transición del estado i al estado j, los estados terminales $\{S_k\}$ son aquellos en los que $\sum_i p_{ki} = 0$. ¿Cuál es el tiempo esperado E_i para alcanzar un estado terminal desde el estado S_i?

Cadenas de Markov

Si tenemos un sistema con un conjunto de estados $\{S_i\}$, con probabilidad p_{ij} conocida de efectuar una transición del estado i al estado j, los estados terminales $\{S_k\}$ son aquellos en los que $\sum_i p_{ki} = 0$. ¿Cuál es el tiempo esperado E_i para alcanzar un estado terminal desde el estado S_i?
Para los estados terminales, claramente

$$E_k = 0$$

Para los demas estados

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

Cadenas de Markov

Si tenemos un sistema con un conjunto de estados $\{S_i\}$, con probabilidad p_{ij} conocida de efectuar una transición del estado i al estado j, los estados terminales $\{S_k\}$ son aquellos en los que $\sum_i p_{ki} = 0$. ¿Cuál es el tiempo esperado E_i para alcanzar un estado terminal desde el estado S_i?
Para los estados terminales, claramente

$$E_k = 0$$

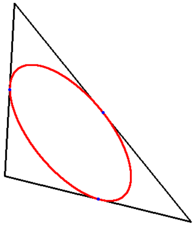
Para los demas estados

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

Es decir que debemos resolver un sistema de ecuaciones sobre los tiempos esperados.

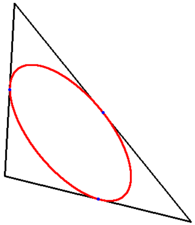
Otros problemas lineales

Tambié aparecen sistemas de ecuaciones en problemas de geometría computacional: *Joe's Triangular Gardens* (NA-GNY'08) pide hallar la elipse tangente a un triángulo en los puntos medios de sus lados:



Otros problemas lineales

Tambié aparecen sistemas de ecuaciones en problemas de geometría computacional: *Joe's Triangular Gardens* (NA-GNY'08) pide hallar la elipse tangente a un triángulo en los puntos medios de sus lados:



Una elipse queda definida por $ax^2 + bxy + cy^2 + dx + ey + f = 0$ con $b^2 - 4ac < 0$, de modo que tenemos 5 parámetros que definen la elipse (a, b, c, d, e y f).

Otros problemas lineales (cont.)

Si (x_i^m, y_i^m) con $i = 1, 2, 3$ son los puntos medios de los lados del triángulo, tenemos 3 ecuaciones de intersección

$$a(x_i^m)^2 + b\,x_i^m y_i^m + c(y_i^m)^2 + d\,x_i^m + e\,y_i^m + f = 0$$

Otros problemas lineales (cont.)

Si (x_i^m, y_i^m) con $i = 1, 2, 3$ son los puntos medios de los lados del triángulo, tenemos 3 ecuaciones de intersección

$$a(x_i^m)^2 + b\,x_i^m y_i^m + c(y_i^m)^2 + d\,x_i^m + e\,y_i^m + f = 0$$

y 2 ecuaciones de tangencia (derivando implícitamente para dos lados no verticales)

$$2a\,x_i^m + b(y_i^m + x_i^m y'(x_i^m, y_i^m)) + 2c\,y_i^m y'(x_i^m, y_i^m) + d + ey'(x_i^m, y_i^m) = 0$$

Los valores de las derivadas son simplemente las pendientes de los correspondientes lados del tríangulo: $y'(x_i^m, y_i^m) = \frac{\Delta y_i}{\Delta x_i}$.

Otros problemas lineales (cont.)

Si (x_i^m, y_i^m) con $i = 1, 2, 3$ son los puntos medios de los lados del triángulo, tenemos 3 ecuaciones de intersección

$$a(x_i^m)^2 + b\,x_i^m y_i^m + c(y_i^m)^2 + d\,x_i^m + e\,y_i^m + f = 0$$

y 2 ecuaciones de tangencia (derivando implícitamente para dos lados no verticales)

$$2a\,x_i^m + b(y_i^m + x_i^m y'(x_i^m, y_i^m)) + 2c\,y_i^m y'(x_i^m, y_i^m) + d + ey'(x_i^m, y_i^m) = 0$$

Los valores de las derivadas son simplemente las pendientes de los correspondientes lados del tríangulo: $y'(x_i^m, y_i^m) = \frac{\Delta y_i}{\Delta x_i}$.

Si resolvemos el sistema de 5 ecuaciones con 5 incógnitas, la solución al problema está prácticamente dada.

Sistemas de ecuaciones

Un sistema de ecuaciones sobre N variables

$$a_{11}\,x_1 + \cdots + a_{1N}\,x_N = b_1$$
$$\vdots$$
$$a_{N1}\,x_1 + \cdots + a_N\,x_N = b_N$$

Puede representarse matricialmente como

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

Sistemas de ecuaciones

Un sistema de ecuaciones sobre N variables

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

Puede representarse matricialmente como

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

Resolver el sistema consiste en encontrar la inversa A^{-1}, porque entonces $\vec{x} = A^{-1}\vec{b}$. Podemos resolver varios sistemas de ecuaciones con diferentes términos independientes $\vec{b_i}$ armando una matriz B con los vectores $\vec{b_i}$ como sus columnas. Entonces $X = A^{-1}B$ y observamos que si $B \mapsto \mathbb{1}$, $X \mapsto A^{-1}$.

Sistemas de ecuaciones (cont.)

Para resolver un sistema a mano, despejamos una variable de una ecuación y la usamos para eliminar las apariciones de esa variable en las demás ecuaciones, trabajando simultáneamente con los términos independientes. Para eso podemos:

- Multiplicar o dividir una ecuación (fila) por un número.
- Sumar o restar una ecuación (fila) a otra.
- Intercambiar dos filas (no modifica las ecuaciones).

Sistemas de ecuaciones (cont.)

Para resolver un sistema a mano, despejamos una variable de una ecuación y la usamos para eliminar las apariciones de esa variable en las demás ecuaciones, trabajando simultáneamente con los términos independientes. Para eso podemos:

- Multiplicar o dividir una ecuación (fila) por un número.
- Sumar o restar una ecuación (fila) a otra.
- Intercambiar dos filas (no modifica las ecuaciones).

El algoritmo de Gauss-Jordan consiste en formalizar este procedimiento con un sólo cuidado: para reducir el error numérico, las variables se despejan de las ecuaciones en las que aparecen con el coeficiente más grande en valor absoluto en cada paso (llamamos a esto el *pivoteo*).

Sistemas de ecuaciones (cont.)

Para resolver un sistema a mano, despejamos una variable de una ecuación y la usamos para eliminar las apariciones de esa variable en las demás ecuaciones, trabajando simultáneamente con los términos independientes. Para eso podemos:

- Multiplicar o dividir una ecuación (fila) por un número.
- Sumar o restar una ecuación (fila) a otra.
- Intercambiar dos filas (no modifica las ecuaciones).

El algoritmo de Gauss-Jordan consiste en formalizar este procedimiento con un sólo cuidado: para reducir el error numérico, las variables se despejan de las ecuaciones en las que aparecen con el coeficiente más grande en valor absoluto en cada paso (llamamos a esto el *pivoteo*).

Nota: ¡Nada nos impide usar esta misma técnica para resolver ecuaciones en $\mathbb{Z}_p$, multiplicando por el inverso en lugar de dividir!

Eliminación de Gauss-Jordan

```
bool invert(double **A, double **B, int N) {
  int i, j, k, jmax; double tmp;
  for (i=1; i<=N; i++) {
    jmax = i; //Maximo el. de A en la col. i con fila >= i
    for (j=i+1; j<=N; j++)
      if (abs(A[j][i]) > abs(A[jmax][i])) jmax = j;

    for (j=1; j<=N; j++) {//Intercambiar las filas i y jmax
      swap(A[i][j], A[jmax][j]); swap(B[i][j], B[jmax][j]);
        }

    //Controlar que la matriz sea invertible
    if (abs(A[i][i]) < EPS) return false;

    tmp = A[i][i];//Normalizar la fila i
    for (j=1; j<=N; j++) { A[i][j] /= tmp; B[i][j] /= tmp; }

    //Eliminar los valores no nulos de la columna i
    for (j=1; j<=N; j++) {
      if (i == j) continue;
      tmp = A[j][i];
      for (k=1; k<=N; k++) {
        A[j][k] -= A[i][k]*tmp; B[j][k] -= B[i][k]*tmp;
      }
    }
  }
  return true;
}
```

Eliminación de Gauss-Jordan

Eliminación de Gauss-Jordan para matrices bidiagonales

El algoritmo de Gauss-Jordan claramente es $\mathcal{O}(N^3)$. Puede verse que si sabemos multiplicar dos matrices de $N \times N$ en $\mathcal{O}(T(N))$, podemos invertir una matriz o calcular su determinante en el mismo tiempo asintótico.

Eliminación de Gauss-Jordan para matrices bidiagonales

El algoritmo de Gauss-Jordan claramente es $\mathcal{O}(N^3)$. Puede verse que si sabemos multiplicar dos matrices de $N \times N$ en $\mathcal{O}\left(T(N)\right)$, podemos invertir una matriz o calcular su determinante en el mismo tiempo asintótico.

En general, en lugar de optimizar el algoritmo general conviene aprovechar alguna propiedad particular de las matrices que queremos invertir: podemos invertir una matriz bidiagonal o tridiagonal (con elementos diagonales no nulos) en $\mathcal{O}(N)$.

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

Ejercicios (3)

Para implementar y poner a prueba lo que hablamos

- SWERC'08, p.B *First Knight*
- SPOJ, p.339 *Recursive Sequence*

Ejercicios (3)

Para implementar y poner a prueba lo que hablamos

- SWERC'08, p.B *First Knight*
- SPOJ, p.339 *Recursive Sequence*

Algunos problemas entretenidos

- TC SRM 443, p.3 *ShuffledPlaylist*: Contar la cantidad de caminos en un grafo, con un poco de imaginación...
- TCO'08 Semifinal Room 2, p.3 *ColorfulBalls*
- CodeForces BR24, p.D *Broken robot*: Calcular el tiempo esperado para llegar a la ultima fila desde una posición arbitraria de una grilla de $N \times N$ con $N \leq 10^3$, cuando podemos en cada paso quedarnos quietos, movernos a los lados o hacia abajo con ciertas probabilidades dadas.

Un problema adicional para ver qué nos falta

SARC'08, p.B *Bases*: Analizar en qué bases una expresión como $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ es válida.

Un problema adicional para ver qué nos falta

SARC'08, p.B *Bases*: Analizar en qué bases una expresión como $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ es válida.

Queda para la próxima discutir:

- Polinomios (evaluación, operaciones, propiedades, etc).
- Evaluación de expresiones matemáticas (parseo).

Suerte en Varsovia :-)