Repaso de aritmética para ICPC

VI Campamento Caribeño de Entrenamiento para el ACM-ICPC

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

16 de marzo de 2015

Contenidos

- Números naturales
  - Cálculo de números primos
  - Factorización, $\varphi(n)$ de Euler y cantidad de divisores de n

Contenidos

- Números naturales
  - Cálculo de números primos
  - Factorización, $\varphi(n)$ de Euler y cantidad de divisores de n
- Aritmética modular
  - Operaciones básicas
  - ModExp
  - GCD y su extensión
  - Teorema chino del resto

Contenidos

- Números naturales
  - Cálculo de números primos
  - Factorización, $\varphi(n)$ de Euler y cantidad de divisores de n
- Aritmética modular
  - Operaciones básicas
  - ModExp
  - GCD y su extensión
  - Teorema chino del resto
- Matrices
  - Notación y operaciones básicas
  - Matriz de adyacencia de un grafo
  - Cadenas de Markov y otros problemas lineales
  - Sistemas de ecuaciones
  - Algoritmo de Gauss-Jordan
  - Caso particular: matrices bidiagonales

Contenidos

- Números naturales
  - Cálculo de números primos
  - Factorización, $\varphi(n)$ de Euler y cantidad de divisores de n
- Aritmética modular
  - Operaciones básicas
  - ModExp
  - GCD y su extensión
  - Teorema chino del resto
- Matrices
  - Notación y operaciones básicas
  - Matriz de adyacencia de un grafo
  - Cadenas de Markov y otros problemas lineales
  - Sistemas de ecuaciones
  - Algoritmo de Gauss-Jordan
  - Caso particular: matrices bidiagonales
- Problemas adicionales

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 en forma 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 en forma única como

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

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

- funciones (completamente) multiplicativas;
- divisores de un número;
- números primos y factorizaciones en general :-)

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor N (*e.g.* para factorizar m necesitamos todos los primos hasta $\sqrt{m}$).

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor N (*e.g.* para factorizar m necesitamos todos los primos hasta $\sqrt{m}$).

- Un algoritmo muy ingenuo: para cada $n \in [2, N)$, controlamos si n es divisible por algún primo menor que $\sqrt{n}$ (todos ellos ya han sido encontrados). Con algunas optimizaciones menores:

```
p[0] = 2; P = 1;
for (i=3; i<N; 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 números primos

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor N (*e.g.* para factorizar m necesitamos todos los primos hasta $\sqrt{m}$).

- Un algoritmo muy ingenuo: para cada $n \in [2, N)$, controlamos si n es divisible por algún primo menor que $\sqrt{n}$ (todos ellos ya han sido encontrados). Con algunas optimizaciones menores:

```
p[0] = 2; P = 1;
for (i=3; i<N; 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 números primos

Cada número considerado puede requerir hasta $\pi(\sqrt{n}) = \mathcal{O}\left(\sqrt{n}/\ln n\right)$ operaciones, luego este algoritmo es supra-lineal.

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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 muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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 muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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	$\cancel{48}$	49	$\cancel{50}$	$\cancel{51}$

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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 muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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.)

El código correspondiente es

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

Criba de Eratóstenes

Este algoritmo es $\mathcal{O}(N \log \log N)$, pero puede llevarse a $\mathcal{O}(N)$ con algunas optimizaciones.

Factorización usando la criba

La criba puede guardar más información:

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

Criba de Eratóstenes, optimizada y extendida

Factorización usando la criba

La criba puede guardar más información:

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

Criba de Eratóstenes, optimizada y extendida

Luego

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

Factorización usando la criba

Divisores de un número

Ahora podemos generar los divisores de un número recursivamente:

```
int d[MAXD], D=0;

void div(int cur, int f[], int s, int e) {
    if (s == e) d[D++] = cur;
    else {
        int m;
        for (m=s+1; m<e && f[m]==f[s]; m++);
        for (int i=s; i<=m; i++) {
            div(cur, f, m, e);
            cur *= f[s];
        }
    }
}
```

Algoritmo tipo DFS para generar todos los divisores de un número

Recordar que $f[\cdots]$ debe contener los factores primos de N **en orden**: primero hay que usar *sort* sobre la salida de *fact*, y después llamar a $div(1, f, 0, F)$.

Algunas funciones de teoría de números

Si podemos factorizar un número n, podemos calcular algunas funciones de teoría de números:

Algunas funciones de teoría de números

Si podemos factorizar un número n, podemos calcular algunas funciones de teoría de números:

- Función φ de Euler: $\varphi(n)$ es el número de enteros positivos menores que n que son coprimos con n. Tenemos

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

Algunas funciones de teoría de números

Si podemos factorizar un número n, podemos calcular algunas funciones de teoría de números:

- Función φ de Euler: $\varphi(n)$ es el número de enteros positivos menores que n que son coprimos con n. Tenemos

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

- El número de divisores de n es

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

  Los números más grandes se alcanzan cuando hay muchos factores primos distintos (n primos distintos $\rightarrow \sigma(n) = 2^n$):

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

  "sólo" tiene 1024 divisores.

- Existen fórmulas similares para $\sigma_m(n) = \sum_{d|n} d^m$

Lectura complementaria

Hay otros algoritmos más eficientes para encontrar números primos. En particular, la criba de Eratóstenes se puede optimizar aun más:

- en memoria, para usar solo un bit por número;
- usando una "rueda" para alcanzar un mejor tiempo de ejecución:

```
int w[8] = {4,2,4,2,4,6,2,6};

for (i=7,cur=0; i*i<N; i+=w[cur++&7]) {
    if (!p[i]) for (j=i*i; j<N; j+=2*i) p[j] = i;
}
```

Una rueda muy sencilla para la criba de Eratóstenes

¿Existe un punto medio en la batalla tiempo vs. memoria?

Aritmética modular

Recordamos que dados $a, r \in \mathbb{Z}$ 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 multiplicación se extienden trivialmente y mantienen sus 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 $a, r \in \mathbb{Z}$ 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 multiplicación se extienden trivialmente y mantienen sus 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 multiplicación por el inverso, de modo que

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

¿Siempre existe un inverso módulo m? ¿Cómo lo calculamos?

ModExp

A veces directamente podemos evitar calcular inversos: *e.g.* el problema *Last Digit* (MCA'07) nos pide calcular el último dígito no nulo de

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

ModExp

A veces directamente podemos evitar calcular inversos: *e.g.* el problema *Last Digit* (MCA'07) nos pide calcular el último 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 la criba, y luego evaluarlo mod 10 después de eliminar la mayor cantidad posible de 2 y 5.
Adicionalmente, necesitamos evaluar $a^b \mod m$ eficientemente:

ModExp

A veces directamente podemos evitar calcular inversos: *e.g.* el problema *Last Digit* (MCA'07) nos pide calcular el último 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 la criba, y luego evaluarlo mod 10 después de eliminar la mayor cantidad posible de 2 y 5. Adicionalmente, necesitamos evaluar $a^b \mod m$ eficientemente:

- Una evaluación directa tomaría $\mathcal{O}(b)$, lo cual es (generalmente) demasiado lento.

ModExp

A veces directamente podemos evitar calcular inversos: *e.g.* el problema *Last Digit* (MCA'07) nos pide calcular el último 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 la criba, y luego evaluarlo mod 10 después de eliminar la mayor cantidad posible de 2 y 5.
Adicionalmente, necesitamos evaluar $a^b \mod m$ eficientemente:

- Una evaluación directa tomaría $\mathcal{O}(b)$, lo cual es (generalmente) demasiado lento.
- Escribimos b en binario, o sea $b = c_0.2^0 + \cdots + c_{\log b}.2^{\log b}$, y luego 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)

```
long long modexp(long long a, int b) {
  long long RES = 1LL;
  while (b > 0) {
    if (b&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;

    e[N]++;
    for (i=0; i<M; i++) if (m[i] > 1) e[m[i]]--;
    for (i=N-1; i>=2; i--) e[i] += e[i+1];
    for (i=N; i>=2; i--) if (p[i]) {
        e[i/p[i]] += e[i];
        e[p[i]] += e[i];
        e[i] = 0;
    }
    int tmp = min(e[2], e[5]);
    e[2] -= tmp; e[5] -= tmp;

    RES = 1;
    for (i=2; i<=N; i++) if (e[i] != 0) {
        RES = (RES*modexp(i, e[i]))% MOD;
        e[i] = 0;
    }
    return RES;
}
```

Last Digit *(MCA'07)*

GCD

El máximo común divisor de dos números a y b es el d más grande tal que $d|a$ y $d|b$. Observamos que

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

lo cual nos lleva al siguiente algoritmo

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

Algoritmo de Euclides

GCD

El máximo común divisor de dos números a y b es el d más grande tal que $d|a$ y $d|b$. Observamos que

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

lo cual nos lleva al siguiente algoritmo

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

Algoritmo de Euclides

Se puede mostrar que $\gcd(F_{n+1}, F_n)$ requiere exactamente n operaciones (donde F_n son los números de Fibonacci). Como F_n crece exponencialmente y es la peor entrada posible para este algoritmo, el tiempo de ejecución es $\mathcal{O}(\log n)$.

GCD extendido

Se puede probar que

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

GCD extendido

Se puede probar que

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

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

GCD extendido

Se puede probar que

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

Luego $x \equiv_m a^{-1}$, de modo que a tiene inverso mod 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; inverso mod m

Teorema chino del resto

Dado un conjunto de condiciones

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

existe un único $x \mod N = n_1 \dots n_k$ que satisface todas estas condiciones simultáneamente.

Teorema chino del resto

Dado un conjunto de condiciones

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

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

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

Si $\bar{m}_i = m_i^{-1} \mod n_i$, tenemos

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

Teorema chino del resto (código)

```
int crt(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

Matrices

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

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

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

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

Matrices

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

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

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

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

Para matrices cuadradas (de ahora en adelante, suponemos que trabajamos con matrices de $N \times N$), tiene sentido preguntarse si una matriz A tiene inverso o no. Se puede ver que si $\det A \neq 0$, la inversa existe y satisface

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

Matrices (cont.)

Representamos las matrices usando arreglos bidimensionales. En C++, para usar una matriz como argumento de una función es conveniente definir una lista de punteros para evitar fijar una de las dimensiones en la definición de la función:

```
type function(int **A, int N, int M) {
  ...
}

int main() {
  int a[MAXN][MAXN], *pa[MAXN];

  for (int i=0; i<MAXN; i++) pa[i] = a[i];
  ...
  function(pa, N, M);
  ...
}
```

Lista de punteros para representar una matriz

Matriz de adyacencia

Podemos usar una matriz mara representar las aristas de un grafo.

Matriz de adyacencia

Podemos usar una matriz mara representar las aristas de un grafo. Para un grafo de N nodos, una matriz $A_{N\times N}$ puede tener A_{ij}:

- el peso de la arista del nodo i al nodo j (o ∞ si no existe tal arista);

Matriz de adyacencia

Podemos usar una matriz mara representar las aristas de un grafo. Para un grafo de N nodos, una matriz $A_{N\times N}$ puede tener A_{ij}:

- el peso de la arista del nodo i al nodo j (o ∞ si no existe tal arista);
- el número de aristas del nodo i al nodo j (0 si no hay ninguna).

En este último caso,

$$(A^2)_{ij} = \sum_k A_{ik}A_{kj} \quad \Longrightarrow \quad \text{caminos } i \to j \text{ con exactamente 2 aristas}$$

y en general A^n contiene el número de caminos de exactamente n aristas entre todos los pares de nodos del grafo original.

Matriz de adyacencia

Podemos usar una matriz mara representar las aristas de un grafo. Para un grafo de N nodos, una matriz $A_{N\times N}$ puede tener A_{ij}:

- el peso de la arista del nodo i al nodo j (o ∞ si no existe tal arista);
- el número de aristas del nodo i al nodo j (0 si no hay ninguna).

En este último caso,

$$(A^2)_{ij} = \sum_k A_{ik}A_{kj} \quad \Longrightarrow \quad \text{caminos } i \to j \text{ con exactamente 2 aristas}$$

y en general A^n contiene el número de caminos de exactamente n aristas entre todos los pares de nodos del grafo original.
Podemos calcular A^n usando una versión adaptada de *ModExp* en $\mathcal{O}(N^3 \log n)$.

Recurrencias lineales

Una recurrencia lineal de orden K define una secuencia $\{a_n\}$ por medio de

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

y un vector $\vec{a}_K = (a_K, a_{K-1}, \ldots, a_1)$ de valores iniciales.

Recurrencias lineales

Una recurrencia lineal de orden K define una secuencia $\{a_n\}$ por medio de

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

y un vector $\vec{a}_K = (a_K, a_{K-1}, \ldots, a_1)$ de valores iniciales.

Sea $\vec{a}_n$ el vector que contiene a_n y sus $K-1$ elementos previos. Entonces la recurrencia se puede representar como

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

y podemos calcular el N-ésimo término usando $\vec{a}_N = R^{N-K}\,\vec{a}_K$.

Cadenas de Markov

Sea $\{S_i\}$ un conjunto de estados, y p_{ij} las probabilidades (conocidas) de observar 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 número esperado E_i de transiciones que se requieren para alcanzar un estado terminal desde el estado S_i?

Cadenas de Markov

Sea $\{S_i\}$ un conjunto de estados, y p_{ij} las probabilidades (conocidas) de observar 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 número esperado E_i de transiciones que se requieren para alcanzar un estado terminal desde el estado S_i?
Para estados terminales, claramente se tiene

$$E_k = 0$$

Para los demás,

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

Cadenas de Markov

Sea $\{S_i\}$ un conjunto de estados, y p_{ij} las probabilidades (conocidas) de observar 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 número esperado E_i de transiciones que se requieren para alcanzar un estado terminal desde el estado S_i?
Para estados terminales, claramente se tiene

$$E_k = 0$$

Para los demás,

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

En otras palabras, debemos resolver un sistema de ecuaciones sobre las cantidades esperadas de transiciones $\{E_i\}$.

Sistemas de ecuaciones

Un sistema de ecuaciones sobre N variables es

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

y se puede representar matricialmente como

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

Sistemas de ecuaciones

Un sistema de ecuaciones sobre N variables es

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

y se puede representar matricialmente como

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

Resolver este sistema se reduce a encontrar la inversa A^{-1}, porque entonces $\vec{x} = A^{-1}\vec{b}$. Podemos resolver varios sistemas de ecuaciones con distintos términos independientes $\vec{b_i}$ construyendo una matriz B con los vectores $\vec{b_i}$ como sus columnas. Luego $X = A^{-1}B$, y en particular si $B \mapsto \mathbb{1}$, $X \mapsto A^{-1}$.

Sistemas de ecuaciones (cont.)

Para resolver un sistema "a mano", despejamos una variable cualquiera en una de las ecuaciones, y luego usamos esto para eliminar esa variable de todas las demás ecuaciones, trabajando simultáneamente con los términos independientes. Para lograr esto, podemos

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

Sistemas de ecuaciones (cont.)

Para resolver un sistema “a mano”, despejamos una variable cualquiera en una de las ecuaciones, y luego usamos esto para eliminar esa variable de todas las demás ecuaciones, trabajando simultáneamente con los términos independientes. Para lograr esto, podemos

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

El algoritmo de Gauss-Jordan formaliza este procedimiento, con un solo detalle: para reducir el error numérico, en cada paso la variable correspondiente se despeja de la ecuación en la que aparece con el coeficiente más grande en valor absoluto (esto se llama *pivoteo*).

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; //Largest el. in A at col. i with row >= 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++) {//Swap rows i and jmax
      swap(A[i][j], A[jmax][j]); swap(B[i][j], B[jmax][j]);
        }

    //Check this matrix is invertible
    if (abs(A[i][i]) < EPS) return false;

    tmp = A[i][i];//Normalize row i
    for (j=1; j<=N; j++) { A[i][j] /= tmp; B[i][j] /= tmp; }

    //Eliminate non-zero values in column 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 es claramente $\mathcal{O}(N^3)$. Se puede ver que si podemos multiplicar dos matrices de $N \times N$ en $\mathcal{O}(T(N))$, entonces 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 es claramente $\mathcal{O}(N^3)$. Se puede ver que si podemos multiplicar dos matrices de $N \times N$ en $\mathcal{O}\left(T(N)\right)$, entonces podemos invertir una matriz o calcular su determinante en el mismo tiempo asintótico.

Usualmente, en lugar de optimizar el algoritmo general es mucho más conveniente aprovechar alguna propiedad especial de las matrices que queremos invertir: por ejemplo, podemos invertir una matriz bidiagonal o tridiagonal (con elementos diagonales no nulos) en $\mathcal{O}(N)$.

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

Lectura complementaria

- Hay algoritmos más eficientes para multiplicar dos matrices (el de Strassen es $\mathcal{O}(n^{2,807})$ y el de Coppersmith–Winograd es $\mathcal{O}(n^{2,376})$), pero no necesariamente son convenientes para una competencia...
- Hay muchas descomposiciones matriciales importantes (*i.e.* formas de escribirlas como sumas o productos de otras matrices con propiedades especiales). Estas tienen un rol muy importante en muchos algoritmos, como los usados para calcular determinantes.
- Hay muchas formas de generalizar el tratamiento de las recurrencias lineales: podemos trabajar con conjuntos de varias recurrencias interdependientes, recurrencias inhomogéneas, *etc.*

Problemas adicionales

Bases (SARC'08): Analizar en qué bases una expresión como $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ es verdadera.

Tough Water Level (CERC'07): Calcular el volumen de un sólido de revolución cuyo perfil es una función arbitraria.

Problemas adicionales

Bases (SARC'08): Analizar en qué bases una expresión como $10000 + 3 * 5 * 334 = 3 * 5000 + 10 + 0$ es verdadera.

Tough Water Level (CERC'07): Calcular el volumen de un sólido de revolución cuyo perfil es una función arbitraria.

Temas que no cubrimos

- Polinomios (evaluación, operaciones, propiedades, *etc.*).
- Evaluación de expresiones matemáticas (*parsing*).
- Integración numérica.