#define DB(a) cerr << \_\_LINE\_\_ << ": " << \

#a << " = " << (a) << endl;

**ios\_base**::sync\_with\_stdio(0); **cin**.tie(0);

**cout** << boolalpha << **setprecision**(6) << fixed;

**// Criba, Factores Primos y Divisores de N**

#define MAXD 10000000

**int** N, p[MAXD], d[MAXD], D;

**void** criba (**int** T = MAXD) {

**for** (**int** i = 4; i < T; i+=2) p[i] = 2;

**for** (**int** i = 3; i\*i < T; i+=2)

**if** (!p[i]) **for** (**int** j = i\*i; j < T; j+=2\*i) p[j] = i;

}

**int** fact (**int** n, **int** f[]) {

**int** F = 0;

**while** (p[n]) {

f[F++] = p[n];

n /= p[n];

}

f[F++] = n;

**return** F;

}

**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];

}

}

}

Recordar que *f*[···] 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*).

**// Sieve Eratosthenes O(loglogN)**

//The number of primes below 10^8 is 5761455

#define GET(b) ((sieve[(b) >> 5] >> ((b) & 31)) & 1)

**const int** MAXN = 10000000, // maximum value of N

P1 = (MAXN + 63) / 64, // ceil(MAXN / 64)

P2 = (MAXN + 1) / 2, // ceil(MAXN / 2)

P3 = 5000; // ceil(ceil(sqrt(MAXN))/2)

**int** sieve[P1];

**void** make () {

**for** (**int** k = 1; k <= P3; ++k) {

**if** (GET(k) == 0) {

**for** (**int** i = 2\*k\*(k + 1), j = 2\*k + 1; i<P2; i += j)

sieve[i >> 5] |= 1 << (i & 31);

}

}

}

**inline** **int** is\_prime (**int** p) {

**return** p == 2 || (p > 2 && (p&1) == 1 && (GET(p>>1) == 0));

}

**int** main() {

make();

**int** ans = 2;

**for** (**int** i = 6; i <= MAXN; i += 6) {

ans += is\_prime(i - 1) + is\_prime(i + 1);

}

}

**// NUMBER THEORY**

// NOTES

Let f[x] be the smallest prime divisor of x, and inv[x] the inverse of x, then

inv[x] = (inv[x/f[x]] \* inv[f[x]]) % mod. (x is non-prime)

Inverse element:

p % i = p - (p / i) \* i

p % i = -(p / i) \* i (mod p) // divide by i \* (p % i)

inv[i] = -(p / i) \* inv[p % i]

**// Inverso Modular (mod p), soluciones en a[1,2,...,p)**

ll inv\_mod2 (ll a, ll p) {

**static** **int** first = **true**, inv[MAXN]; // MAXN = 1e7

**if** (first) {

first = false;

inv[1] = 1;

**for** (**int** i= 2; i < p; ++i)

inv[i] = (p - (p / i) \* inv[p % i] % p) % p;

}

**return** inv[a];

}

**// Euler Phi Funtion, soluciones en f[1,2,...,MAXN)**

ll phi2 (ll n) {

**static** **int** first = **true**, p[MAXN], f[MAXN]; // MAXN = 1e7

**if** (first) {

first = false;

**for** (**int** i = 0; i < MAXN; ++i)

p[i] = 1, f[i] = i;

**for** (**int** i = 2; i < MAXN; ++i) {

**if** (p[i]) {

f[i] -= f[i] / i;

**for** (**int** j = i + i; j < MAXN; j += i)

p[j] = **false**, f[j] -= f[j] / i;

}

}

}

**return** f[n];

}

**// RMQ Modificado – Operaciones más Generales**

**void** Init (**int** \*m, **int** N, **int** \*\*st) { // O(N log N)  
 **for** (**int** i = 0; i < N; i++)

st[0][i] = m[i];  
 **for** (**int** k = 1; (1 << k) <= N; k++)  
 **for** (**int** i = 0; i + (1 << k) <= N; i++)  
 st[k][i] = oper (st[k−1][i], st[k−1][i+(1<<(k−1))]);

}

// Para operaciones básicas como mínimo o máximo  
**int** Query (**int** \*\*st, **int** s, **int** e) { // O(1)  
 **int** k = 31 − \_\_builtin\_clz(e−s);  
 **return** **min**(st[k][s], st[k][e−(1<<k)]);  
}

// Operaciones más generales O(log N)

**int** Query (**int** \*\*st, **int** s, **int** e) {

**int** RES = 0, k = e−s;  
 **for** (**int** i = 0; (1 << i) <= k; i++) **if** (k & (1 << i)) {

RES = oper (RES, sm[i][s]);

s += (1 << i);  
 }  
 **return** RES;  
}

**// STRINGS – Manacher**

rad[i] = If i is odd, it's the largest even palindrome centered at position i / 2. Otherwise, it's the size of the largest odd palindrome centered at position i / 2.

**const** **int** LEN = 1e5 + 5;

**char** s[LEN];

**int** rad[2 \* LEN], n;

**void** build\_rad () { // O(N)

**for** (**int** i=0, j=0, k; i < 2\*n; i += k, j = **max**(j-k, 0)) {

**for** (; i >= j && i + j + 1 < 2\*n &&

s[(i - j) / 2]==s[(i + j + 1) / 2]; ++j);

rad[i] = j;

**for** (k=1; i>=k && rad[i] >= k && rad[i-k]!=rad[i]-k; ++k)

rad[i + k] = **min**(rad[i - k], rad[i] - k);

}

}

**bool** is\_palindrome (**int** b, **int** e) { // O(1)

**return** b >= 0 && e < n && rad[b + e] >= e - b + 1;

}

**// Binary Function (Segment Tree)**

#define MAXN 1000

**typedef** **pair**<**int**, **int**> ii;

**int** tree[4 \* MAXN], a[MAXN] = {1, 5, 3, 7, 3, 8, 5, 3};

**int** (\*funct)(**int** c, **int** d), neuter;

/\* Initialize the segment tree O(n) \*/

**void** init (**int** v, **int** l, **int** r) {

**if** (l == r) tree[v] = funct (a[l], neuter);

**else** {

**int** m = l + (r - l) / 2;

init (2 \* v, l, m);

init (2 \* v + 1, m + 1, r);

tree[v] = funct (tree[2 \* v], tree[2 \* v + 1]);

}

}

/\* Get the value of funct (nl,nl+1,...,nr-1,nr) O(logn) \*/

**int** query (**int** v, **int** l, **int** r, **int** nl, **int** nr) {

**if** (l >= nl && r <= nr)

**return** tree[v];

**if** (l > nr || r < l)

**return** neuter;

**int** m = l + (r - l) / 2;

**int** lval = query (2 \* v, l, m, nl, nr);

**int** rval = query (2 \* v + 1, m + 1, r, nl, nr);

**return** funct (lval, rval);

}

/\* Update the value in a given position O(logn) \*/

**void** update (**int** v, **int** l, **int** r, **int** pos, **int** val) {

**if** (l == r)

tree[v] = funct (val, neuter);

**else** {

**int** m = l + (r - l) / 2;

**if** (pos <= m) update (2 \* v, l, m, pos, val);

**else** update (2 \* v + 1, m + 1, r, pos, val);

tree[v] = funct (tree[2 \* v], tree[2 \* v + 1]);

}

}

**int** gcd (**int** c, **int** d) {

**while** (c && d) {

**if** (c > d) c %= d;

**else** d %= c;

} **return** c + d;

}

// neuter = 0x0;

// funct = gcd;

// init (1, 0, n - 1);

**// Binary Indexed Tree**

Permite calcular las frecuencias acumuladas en un intervalo.

**typedef** **vector**<**int**> vi;

// Leer frecuencia acumulada hasta idx. O(log MaxVal)

**int** Query (vi &tree, **int** idx) {

**int** sum = 0;

**for** (; idx > 0; idx &= idx - 1)

sum += tree[idx];

**return** sum;

}

// Cambiar frecuencia en una posición y actualizar tree. O(log MaxVal)

**void** Update (vi &tree, **int** idx, **int** val) {

**for**(; idx < tree.**size**(); idx += (idx & -idx))

tree[idx] += val;

}

// Leer frecuencia en una posición determinada.

// c \* O(log MaxVal), where c is less than 1.

**int** ReadSingle (vi &tree, **int** idx) {

**int** sum = tree[idx];

**if**(idx > 0) {

**int** z = idx - (idx & -idx);

--idx;

**while**(idx != z) {

sum -= tree[idx];

idx -= (idx & -idx);

}

} **return** sum;

}

// Dividiendo todas las frecuencias por un valor constante.

**void** Scale(vi &tree, **int** c) {

**for**(**int** i = 1; i < tree.**size**(); ++i)

tree[i] /= c;

}

// Encontrar un índice con una frecuencia determinada.

// El valor debe ser <= que la mayor frecuencia acumulativa,

// de lo contrario hay desbordamiento en tIdx.

**int** Find(vi &tree, **int** comFrec) {

**int** idx = 0;

// Bit más significativo del mayor índice posible.

**int** bitMask = m(tree.**size**() - 1);

**while** ((bitMask != 0) && (idx < (tree.**size**() - 1))) {

**int** tIdx = idx + bitMask;

**if** (comFrec == tree[tIdx]) **return** tIdx;

**if** (comFrec > tree[tIdx]) {

idx = tIdx;

comFrec -= tree[tIdx];

}

bitMask >>= 1;

}

**if** (comFrec != 0) **return** -1;

**return** idx;

}

//Encuentra el mayor índice con una frecuencia determinada.

**int** FindG(vi &tree, **int** comFrec) {

**int** idx = 0;

// Bit más significativo del mayor indice posible.

**int** bitMask = m(tree.**size**() - 1);

**while** ((bitMask != 0) && (idx < (tree.**size**() - 1))) {

**int** tIdx = idx + bitMask;

**if** (comFrec >= tree[tIdx]) {

idx = tIdx;

comFrec -= tree[tIdx];

}

bitMask >>= 1;

}

**if** (comFrec != 0) **return** -1;

**return** idx;

}

**// Binary Indexed Tree 2D**

Sirve para conocer en un conjunto de puntos, cuantos están en el rectángulo (0, 0) - (x, y)

//Insertar (eliminar) el punto (a,b), llamar con (a,b,1(-1))

**void** Update (vvi &tree, **int** x, **int** y, **int** val) {

**int** yl;

**while** (x < tree.**size**()) {

yl = y;

**while** (yl < tree[x].**size**()) {

tree[x][yl] += val;

yl += (yl & -yl);

}

x += (x & -x);

}

}

**int** Query (vvi &tree, **int** x, **int** y) {

**int** sum = 0;

**while** (x > 0) {

yl = y;

**while** (yl > 0) {

sum += tree[x][yl];

yl ^= (yl & -yl);

}

x ^= (x & -x);

}

**return** sum;

}

**/\* HAMILTONIAN WALKS & CYCLES \*/**

#define BIT(n) (1 << n)

#define INF 0x1fffffff

**const** **int** MAXN = 20;

**int** n, m, u, v;

**// Amount of Hamiltonian Walks O(2^n \* n^2)**

Finding the number of Hamiltonian walks in the unweighted and directed graph G=(V,E). NOTES: Let dp[msk][v] be the amount of Hamiltonian walks on the subgraph generated by vertices in msk that end in the vertex v.

**int** g[MAXN], dp[BIT(MAXN)][MAXN], ans;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[u] |= BIT(v);

}

**for** (**int** i = 0; i < n; ++i)

dp[BIT(i)][i] = 1;

**for** (**int** msk = 1; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i)

**if** (msk & BIT(i)) {

**int** tmsk = msk ^ BIT(i);

**for** (**int** j = 0; tmsk && j < n; ++j) {

**if** (g[j] & BIT(i))

dp[msk][i] += dp[tmsk][j];

}

}

}

**for** (**int** i = 0; i < n; ++i)

ans += dp[BIT(n) - 1][i];

**cout** << ans << **endl**;

}

**// Existence of Hamiltonian Cycle O(2^n \* n)**

Check for existence of Hamiltonian cycle in a directed graph G=(V,E). NOTES: Let dp[msk] be the mask of the subset consisting of those vertices j such that exist a Hamiltonian walk over the subset msk beginning in vertex 0 and ending in j.

**int** g[MAXN], dp[BIT(MAXN)];

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[v] |= BIT(u);

}

dp[1] = 1;

**for** (**int** msk = 2; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i) {

**if** ((msk & BIT(i)) && (dp[msk ^ BIT(i)] & g[i]))

dp[msk] |= BIT(i);

}

}

**cout** << ((dp[BIT(n) - 1] & g[0]) != 0) << **endl**;

}

**// Existence of Hamiltonian Walk O(2^n \* n)**

Check for existence of Hamiltonian walk in the directed graph G=(V,E). NOTES: Let dp[msk] be the mask of the subset consisting of those vertices v for which exist a Hamiltonian walk over the subset msk ending in v.

**int** g[MAXN], dp[BIT(MAXN)];

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[v] |= BIT(u);

}

**for** (**int** i = 0; i < n; ++i)

dp[BIT(i)] = BIT(i);

**for** (**int** msk = 1; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i) {

**if** ((msk & BIT(i)) && (dp[msk ^ BIT(i)] & g[i]))

dp[msk] |= BIT(i);

}

}

**cout** << (dp[BIT(n) - 1] != 0) << **endl**;

}

**// Finding the number of simple paths**

Finding the number of simple paths in the directed graph G=(V,E). NOTES: Let dp[msk][v] be the number of Hamiltonian walks in the subgraph generated by vertices in msk that end in v.

**int** g[MAXN], dp[BIT(MAXN)][MAXN], ans;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[u] |= BIT(v);

}

**for** (**int** i = 0; i < n; ++i)

dp[BIT(i)][i] = 1;

**for** (**int** msk = 1; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i)

**if** (BIT(i) & msk) {

**int** tmsk = msk ^ BIT(i);

**for** (**int** j = 0; tmsk && j < n; ++j)

**if** (g[j] & BIT(i))

dp[msk][i] += dp[tmsk][j];

ans += dp[msk][i];

}

}

**cout** << ans - n << **endl**;

}

**// Finding the shortest Hamiltonian cycle O(2^n \* n^2)**

Search for the shortest Hamiltonian cycle. Let the directed graph G = (V, E) have n vertices, and each edge have weight d(i, j). We want to find a Hamiltonian cycle for which the sum of weights of its edges is minimal. NOTES: Let dp[msk][v] be the length of the shortest Hamiltonian walk on the subgraph generated by vertices in msk beginning in verex 0 and ending in vertex v.

**int** g[MAXN][MAXN], dp[BIT(MAXN)][MAXN], ans = INF;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < n; ++i) {

**for** (**int** j = 0; j < n; ++j)

g[i][j] = INF;

}

**for** (**int** i = 0; i < BIT(n); ++i) {

**for** (**int** j = 0; j < n; ++j)

dp[i][j] = INF;

}

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

**cin** >> g[u][v];

}

dp[1][0] = 0;

**for** (**int** msk = 2; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i) **if** (msk & BIT(i)) {

**int** tmsk = msk ^ BIT(i);

**for** (int j = 0; tmsk && j < n; ++j)

dp[msk][i] = **min**(dp[msk][i], dp[tmsk][j] + g[j][i]);

}

}

**for** (**int** i = 1; i < n; ++i)

ans = **min**(ans, dp[BIT(n) - 1][i] + g[i][0]);

**cout** << ans << **endl**;

}

**// Number of Hamiltonian cycles O(2^n \* n^2)**

Finding the number of Hamiltonian cycles in the unweighted and directed graph G = (V, E). NOTES: Let dp[msk][v] be the amount of Hamiltonian walks on the subgraph generated by vertices in msk that begin in vertex 0 and end in vertex v.

**int** g[MAXN], dp[BIT(MAXN)][MAXN], ans;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[u] |= (1 << v);

}

dp[1][0] = 1;

**for** (**int** msk = 2; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i) **if** (msk & BIT(i)) {

**int** tmsk = msk ^ BIT(i);

**for** (int j = 0; tmsk && j < n; ++j)

**if** (g[j] & BIT(i)) dp[msk][i] += dp[tmsk][j];

}

}

**for** (**int** i = 1; i < n; ++i) **if** (g[i] & 1)

ans += dp[BIT(n) - 1][i];

**cout** << ans << **endl**;

}

**// Number of simple cycles O(2^n \* n^2)**

Finding the number of simple cycles in a directed graph G=(V,E). NOTES: Let dp[msk][v] be the number of Hamiltonian walks in the subgraph generated by vertices in msk that begin in the lowest vertex in msk and end in vertex v.

#define ONES(n) \_\_builtin\_popcount(n)

**int** g[MAXN];

**long** **long** dp[BIT(MAXN)][MAXN], ans;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

g[u] |= BIT(v);

}

**for** (**int** i = 0; i < n; ++i)

dp[BIT(i)][i] = 1;

**for** (**int** msk = 1; msk < BIT(n); ++msk) {

**for** (**int** i = 0; i < n; ++i) {

**if** ((msk & BIT(i)) && !(msk & -msk & BIT(i))) {

**int** tmsk = msk ^ BIT(i);

**for** (**int** j = 0; tmsk && j < n; ++j)

**if** (g[j] & BIT(i))

dp[msk][i] += dp[tmsk][j];

**if** (ONES(msk) > 2 && (g[i] & msk & -msk))

ans += dp[msk][i];

}

}

}

**cout** << ans << **endl**;

}

**// Shortest Hamiltonian Walk O(2^n \* n^2)**

Search for the shortest Hamiltonian walk. Let the directed graph G = (V, E) have n vertices, and each edge have weight d(i, j). We want to find a Hamiltonian walk for which the sum of weights of its edges is minimal. NOTES: Let dp[msk][v] be the length of the shortest Hamiltonian walk on the subgraph generated by vertices in msk that end in vertex v.

**int** d[MAXN][MAXN], dp[1 << MAXN][MAXN], ans = INF;

**int** main() {

**cin** >> n >> m;

**for** (**int** i = 0; i < n; ++i) {

**for** (**int** j = 0; j < n; ++j)

d[i][j] = INF;

}

**for** (**int** i = 0; i < BIT(n); ++i) {

**for** (**int** j = 0; j < n; ++j)

dp[i][j] = INF;

}

**for** (**int** i = 0; i < m; ++i) {

**cin** >> u >> v;

**cin** >> d[u][v];

}

**for** (**int** i = 0; i < n; ++i)

dp[1 << i][i] = 0;

**for** (**int** msk = 1; msk < (1 << n); ++msk) {

**for** (**int** i = 0; i < n; ++i) **if** (msk & BIT(i)) {

**int** tmsk = msk ^ BIT(i);

**for** (**int** j = 0; tmsk && j < n; ++j)

dp[msk][i] = **min**(dp[tmsk][j] + d[j][i], dp[msk][i]);

}

}

**for** (**int** i = 0; i < n; ++i)

ans = **min**(ans, dp[BIT(n) - 1][i]);

**cout** << ans << **endl**;

}

**// DYNAMIC PROGRAMMING – Dominoes Tiling**

Given a N x M table (N < 10), determine the number of different ways to pave the table with non-overlapping dominoes (rectangles 2 x 1 and 1 x 2).

**const** **int** MAXN = 10, MAXM = 10000;

**vector**<**int**> d[1 << MAXN];

**long long** dp[MAXM + 2][1 << MAXN];

**int** n, m;

**void** go (**int** p, **int** p2, **int** l) { // O(2 ^ (2n))

**if** (l == n)

d[p2].**push\_back**(p);

**else** **if** ((1 << l) & p)

go(p, p2, l + 1);

**else** {

go(p, p2 | (1 << l), l + 1);

**if** (l < n - 1 && ((1 << (l + 1)) & p) == 0)

go(p, p2, l + 2);

}

}

**long** **long** solve () { // O(m \* (2 ^ (2n)))

**for** (**int** i = 0; i < (1 << n); ++i)

go (i, 0, 0);

dp[1][0] = 1;

**for** (**int** i = 2; i <= m + 1; ++i) {

**for** (**int** msk = 0; msk < (1 << n); ++msk) {

**for** (**int** j = 0; j < d[msk].**size**(); ++j)

dp[i][msk] = dp[i][msk] + dp[i - 1][d[msk][j]];

}

}

**return** dp[m + 1][0];

}

**NÚMEROS DE STIRLING**

Consideremos un conjunto con n elementos, cuántos conjuntos de k subconjuntos podemos formar que excluyan el elemento vacío y que la unión de ellos, nos da el conjunto original:

S(0, 0) = 1 and S(n, k) = 0 if n ≤ 0 or k ≤ 0

S(n, 1) = S(n, n) = 1, S(n, 2) = 2^n−1 − 1, S(n, n − 1) = C(n, 2)

S(n, k) = S(n − 1, k − 1) + kS(n − 1, k)

**NÚMEROS EULERIANOS**

Sea p = {a1,a2,...,an}, deseamos conocer todas las permutaciones que cumplen la relación ai < ai+1 k veces: Sean {1234} y una permutación {2341}, esta cumple la propiedad 2 veces: 2<3 y 3<4. Los números eulerianos cuentan la cantidad de dichas permutaciones:

E(n, k) = k E(n−1, k) + (n−k+1) E(n−1, k−1)

**PARTICIONES ENTERAS**

Se quiere contar de cuántas formas se puede escribir un número entero positivo como la suma de k enteros positivos: 3 se escribe como 1+1+1, 1+2, 3, 1 como 3, 1 como 2 y 1 como 1. p(n,k) cuenta las formas de escribir n como k sumandos: p(n, k) = p(n − 1, k − 1) + p(n − k, k)

**NUMBERS THEORY**

Un número R en base N es divisible por (N−1) si y solo si la suma de sus dígitos (en decimal) es divisible por (N−1).

If p is a prime and a !≡ 0 mod p, ap−1 ≡ 1 mod p; if a (p−1)/2 ≡ 1 mod p then there exist b such that b^2 ≡ a mod p.

Let n be a positive integer greater than 1 and let its unique prime factorization be p1^e1\*p2^e2\*. . .\*pk^ek where ei > 0 and pi is prime for all i. Then the Euler Φ function

Φ(n) = n(1 – 1/p1)(1 – 1/p2) . . . (1 – 1/pk) = ∏(pi^ei − pi^(ei−1)) describes the number of positive integers co-prime to n in [1..n]. As a special case, Φ(p) = p − 1 for prime p. The number of divisors of n is ∏(ei + 1).

Euler’s Theorem, which extends Fermat’s Little Theorem:

If mcd(a, n) = 1, aΦ(n) ≡ 1 mod p.

**PROPIEDADES DE FIBONACCI**

**AMOUNT OF SPANNING TREES IN COMPLETE GRAPH**

T(Kn) = n^(n−2)

**AMOUNT OF SPANNING TREES IN COMPLETE BIPARTITE GRAPH**

T(Kp,q) = p^(q−1) \* q^(p−1)

**AMOUNT OF SPANNING TREES IN GRAPH (KIRCHHOFF'S THEOREM)**

if vertex i is adjacent to vertex j in G, then Qi,j equals −m, where m is the number of edges between i and j;

Qi,i = degree(i) , when counting the degree of a vertex, all loops are excluded. Then the amount of spanning trees in the graph is equal to the determinant of Q matrix erasing the last row and column.

**/\* Fenwick Tree (2D) \*/**

**struct** Fenwick\_Tree\_2D {

**vector**<**vector**<**int**> > data;

Fenwick\_Tree(**int** N, **int** M):data(N, **vector**<**int**>(M, 0)) {}

**inline** **int** lobit(**int** x) {

**return** x & -x;

}

**int** query(**int** i, **int** j) {

**int** sum = 0;

**for** (; i >= 0; i -= lobit(i + 1))

**for** (**int** y = j; y >= 0; y -= lobit(y + 1))

sum += data[i][y];

**return** sum;

}

**void** update(**int** i, **int** j, **int** val) {

**for** (; i < data.**size**(); i += lobit(i + 1))

**for** (**int** y = j; y < data[i].**size**(); y += lobit(y + 1))

data[i][y] += val;

}

};

**/\* Convert an Roman number into integer \*/**

**int** Roman\_to\_int (**string** &s) {

**string** symb = "IVXLCDM";

**int** val[] = {1, 5, 10, 50, 100, 500, 1000};

**int** ans = 0, back = 0;

**for** (**int** i = s.**size**() - 1; i >= 0; --i) {

**int** curr = val[symb.**find**(s[i])];

**if** (curr < back)

ans -= curr;

**else**

ans += curr;

back = curr;

}

**return** ans;

}

**/\* Gray Code \*/**

// Returns the n-th gray code

**int** gray (**int** n) {

**return** n ^ (n >> 1);

}

// Gets the number n such that g is the n-th gray code

**int** reverse\_gray (**int** g) {

**int** n = 0;

**for** (; g; g>>=1)

n ^= g;

**return** n;

}

**/\* TRABAJO CON BITS \*/**

1. **Media aritmética** // (a+b)/2 - parte entera por arriba

(a & b) + ((a ^ b) >> 1)

1. **Linked List doble con un solo puntero.**

Guardamos un puntero al nodo actual y otro a uno de sus vecinos, el valor que guardamos en cada nodo es el XOR de sus dos vecinos. Descodificarlo es con el XOR al vecino guardado y el valor que hay en el nodo.

1. **Código de Gray**

El k-th número de gray es k^(k >> 1)

Para convertir de gray a decimal es:

**unsigned** **int** g2b(**unsigned** **int** gray) {

gray ^= (gray >> 16);

gray ^= (gray >> 8);

gray ^= (gray >> 4);

gray ^= (gray >> 2);

gray ^= (gray >> 1);

**return** (gray);

}

1. **Borrar el bit menos significativo**

x & (x-1)

1. **Obtener el bit más significativo**

**unsigned** **int** m(**unsigned** **int** x) {

x |= (x >> 1); x |= (x >> 2);

x |= (x >> 4); x |= (x >> 8);

x |= (x >> 16);

**return** (x ^ (x >> 1));

**return** (x & ~(x >> 1)); // lo mismo que arriba

**return** x + 1; // La siguiente potencia de dos

// mayor que x;

}

1. **Cantidad de unos en un entero de 32 bits**

**unsigned int** ones32(**unsigned** **int** x) {

x -= ((x >> 1) & 0x55555555);

x = (((x >> 2) & 0x33333333) + (x & 0x33333333));

x = (((x >> 4) + x) & 0x0f0f0f0f);

x += (x >> 8);

x += (x >> 16);

**return** (x & 0x0000003f);

}

1. **Intercambiar valores sin una variable auxiliar**

x ^= y; // x' = (x^y)

y ^= x; // y' = (y^(x^y)) = x

x ^= y; // x' = (x^y)^x = y

1. **Mantener solamente el último bit (1) de un número**

x & (-x)

1. Si en un árbol x[i] representa los vecinos del vértice **i**,para añadirle el vértice j como vecino hacemos

x[i] ^= j;

x[j] ^= i;

degree[i]++;

degree[j]++;

Está claro que los únicos que tendrán una referencia verdadera a su padre son las hojas (degree[i] == 1), por lo que el padre de la hoja i sería

j = x[i];

x[j] ^= i; // De esta manera quitamos el nodo i

// como vecino del nodo j

--degree[j]; // Si ahora degree[j] == 1 entonces

// x[j] tendrá una referencia verdadera

// al padre de j.

1. **Comprobar si un número es de la forma 2^n – 1**

x & (x + 1) == 0

1. **Formar una máscara que identifica la cantidad de ceros al final de un número.** 01011000 -> 00000111

~x & (x - 1) ó

~(x | -x) ó

(x & -x) – 1

1. **Formar una máscara que identifica la cantidad de ceros al final de un número y el ultimo 1.** 01011000 -> 00001111

x ^ (x - 1)

1. **Alternar la variable 'x' entre dos valores 'a' y 'b'.**

x = a + b - x; ó

x ^= a ^ b

1. En una lista en la cual todos los números se repiten una cantidad par de veces excepto uno, la forma de saber cuál es ese número es hacerle XOR a todos los elementos de la lista y el valor resultante es el que se encuentra en la lista una cantidad impar de veces.
2. Sea g(n) el número de 1-s en la representación binaria de n y P(i) la i-ésima fila del triángulo de Pascal módulo 2 (comenzando en 0), la cantidad de 1-s en P(n) es 2^g(n).

P(0) = 1

P(1) = 1 1

P(2) = 1 0 1

P(3) = 1 1 1 1

P(4) = 1 0 0 0 1

P(5) = 1 1 0 0 1 1

1. Los códigos ASCI de una letra minúscula y su correspondiente mayúscula difieren en 32, por lo que para convertir una letra mayúscula a su correspondiente minúscula y viceversa lo podemos hacer mediante

A = a ^ (1<<5), a = A ^ (1<<5)

1. **Máscaras de bits**

// Recorrer todos los subconjuntos del conjunto *s*

**for** (**int** mask = s; mask != 0; mask = (mask − 1)&s) {

/∗ CODIGO ∗/

}

// Recorrer todos los subconjuntos de *fixed*

// que no tienen ningún elemento de *pro*

**int** perm = (((1<<N) − 1) & ~pro);

**int** mask = fixed;

**while** ((mask | pro) != ((1<<N) − 1)) {

/∗ CODIGO ∗/

mask = ((mask + pro + 1) & perm) | fixed;

}

// O bien

**int** rest = (1<<N) − 1) ^ pro ^ fixed;

**for** (**int** mask2=rest; mask2!=0; mask2=(mask2−1)&rest) {

**int** mask = mask2 | fixed;

/∗ CODIGO ∗/

}