Índice

1. Programación

1.1. Misc

Input

```
getline(cin, s); // reads line from input into 's'
```

Strings

```
s.substr(i, len); // returns s[i..i+len], or s[i..] if i+len >= sz(s)
```

Containers

```
vector<int> v( all(c) );
set<int> s( all(c) );
```

Permutaciones

Next permutation muta un container a la permutación inmediata superior, y retorna si quedó ordenado. El siguiente snippet utiliza dicha función para **recorrer en orden** todas las permutaciones de un arreglo/vector **ya ordenado**:

```
vector<int> v;
sort(v.begin(), v.end());
do {
    // code
} while (next_permutation(v.begin(), v.end()));          amortized O(n!) + K
```

La guarda está al final para que el cuerpo del ciclo corra al menos una vez con el vector ordenado (porque la condición de corte es justamente si está ordenado).

Repetidos

```
set<int> s( vec.begin(), vec.end() );
vec.assign( s.begin(), s.end() );
```

Punteros

- ¡Los punteros son considerados *random-access iterators*! Esto implica que todos los algoritmos de la STL los podemos utilizar sobre arrays (por ejemplo, *sort(arr, arr+n)*).
- Los arrays 'decaen' a punteros en casi todas sus operaciones (son excepciones *sizeof* y *&*). Un ejemplo notable es cuando son pasados como argumentos de funciones, razón por la cual los programas siguen tipando, aún cuando le pasamos un array a una función que espera un puntero (obs: sin embargo, sí fallaría si pasásemos un array de arrays).

Recordatorios

- ¡Para usar un *unordered_set* hace falta implementar una función de hash para pares!
- *std::find* para sets es O(n) porque es genérico, lo que es O(log n) es *std::set::find*
- El operador [] para maps crea un elemento si no lo encuentra, *at* en cambio tira una excepción
- ¡El operador módulo devuelve negativos!

1.2. Bitmasks

Operaciones básicas

Conjunto con n elementos	`int x = (1 << n) - 1;` [1]
Pertenencia del i-esimo elemento	`if (x & 1 << i)`
Cardinal	`__builtin_popcount(x)`
Agregar i-esimo elemento	`x |= 1 << i`
Borrar i-esimo elemento	`x &= ~(1 << i)`
Recorrer subconjuntos	`forn(x, 1 << n)`

Operaciones de conjunto

Complemento	`((1 << n) - 1) ^ x)`
Unión	`x | y`
Intersección	`x & y`
Diferencia	`x & ~y`
Diferencia Simétrica	`x ^ y`

Truquitos

Potencia de dos inmediata inferior	`1 << (31 - __builtin_clz(n))`
Recorrer subconjuntos de subconjuntos en $O(n^3)$ (salvo vacio)	`for (int x = y; x > 0; x = (y & (x-1)))`

Comentarios

- Los operadores & y | tienen menor precedencia que los operadores de comparación, con lo cual `x & 3 == 1` se interpreta como `x & (3 == 1)`. ¡Ojo con eso!
- Los operadores *builtin* tienen versiones para `long` y `long long`, (`__builtin_clzll(n)`)
- Los operadores de bits no están completamente definidos sobre enteros con signo. Para código portable y bien escrito, es mejor utilizar tipos sin signo. Dicho eso, en los jueces no explota nada si utilizamos ints. El único cuidado especial es realizar shifts sobre números negativos, que es *undefined behaviour*, lo que podemos evitar dejando en 0 el bit más significativo.

[1] No utilizamos el más sencillo -1 porque al borrar todos los elementos no nos quedaría 0.

2. Estructuras de Datos

2.1. RMQ

La Sparse Table no es dinámica porque tiene mucha redundancia. Alternativamente, esa redudancia es la que le permite devolver queries en $O(1)$ con operaciones idempotentes.

```
struct SparseTable {
    vector<vector<int>> t;
    SparseTable(vector<int>& src) { // O(n log n)
        int n = sz(src);
        t.assign(1 + 31 - __builtin_clz(n), vector<int>(n));
        forn(i, n) t[0][i] = src[i];
        forn(i, sz(t)-1) forn(j, n-(1<<i)) {
            t[i+1][j] = binary_op(t[i][j], t[i][j+(1<<i)]);
        }
    }
    int query(int l, int r) { // O(1) - interval [l, r)
        int niv = 31 - __builtin_clz(r-l);
        return binary_op(t[niv][l], t[niv][r-(1<<niv)]);
    }
    int binary_op(int a, int b) {
        return min(a, b);
    }
};
```

Segment Tree con mínimo puede ser utilizado usado para encontrar la primera ocurrencia de un elemento que cumpla con cierto predicado

3. Strings

3.1. Trie

Wonder

3.2. Hashing

Wonder | Kth | CPA

Dado un hash polinomial:

$$h(s) = \sum_{i=0}^{n} s_i \, p^i$$

- El hash al agregar un caracter al final es $h(s + c) = h(s) + c * p^{|s|}$
- El hash al agregar un caracter al principio es $h(c + s) = c + h(s) * p$
- El hash al quitar caracteres se puede calcular despejando $h(s)$

3.3. KMP

Jonno | Kth

Un *borde* es un substring **estricto** que es prefijo y sufijo a la vez.

Para todo string s y borde no trivial b, $b[1..n)$ es borde de $s[1..n)$. Luego, existe algun borde de $s[1..n)$ que puede extenderse a b. Por contrarecíproco, si no existe ninguno, b no es borde de s.

Ejemplo:

b	a	d	d	b	a	d	b	a
0	0	0	0	1	2	3	1	2

Por ende, podemos conseguir todos los bordes de s de manera recursiva iterando sobre los bordes de $s[1..n)$.

Además, vale lo siguiente:

Lema. *Sea M el borde máximo de S. Entonces, para todo P distinto de S, P borde de S si y solo si P borde de M.*

Entonces, para calcular todos los bordes de un string basta con calcular el máximo borde de todos sus prefijos. Con esta info, todos los bordes de un string s serían $mb[mb[...[mb[s]]...]]$.

Ejemplo: malu*ma*luma $\rightarrow$ **ma**luma $\rightarrow$ ma

4. Geometria

```
struct pto {
    double x, y;
    bool operator<(const pto &q) const {return y==q.y ? x<q.x : y<q.y;}
    double operator*(pto q) {return x*a.x + y.a.y;}
    //module of the cross product or vectorial product:
    //if a is less than 180 clockwise from b, a^b>0 (?????)
    double operator^(pto q) {return x*a.x - y.a.y;}
};
```

Representación elegida de un segmento: paramétrica ($c + tv \quad c, v \in \mathbb{R}^2$)

```
// Given lines in parametric form 'c + tv' and 'd + tw' this
// function yields the values t1, t2 such that 'c + t1v = d + t2w'
pair<double, double> scalar_intersection(pto c, pto v, pto d, pto w) {
    double ndet = v.x * w.y - v.y * w.x;
    if (abs(ndet) < EPS) {
        return {DBL_MAX, DBL_MAX};
    } else {
        return {(w.y * (d.x - c.x) - w.x * (d.y - c.y))/ndet,
                (v.y * (d.x - c.x) - v.x * (d.y - c.y))/ndet};
    }
}

pto seg_intersection(pto c, pto v, pto d, pto w) {
    auto p = scalar_intersection(c, v, d, w);
    if (p.first == DBL_MAX) return {DBL_MAX, DBL_MAX};
    if (p.first < -EPS || p.first > 1 + EPS) return {DBL_MAX, DBL_MAX};
    if (p.second < -EPS || p.second > 1 + EPS) return {DBL_MAX, DBL_MAX};
    return c + v * p.first;
}
```

5. Matemática

Álgebra

$$a^2 + bx + c = 0 \iff x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

5.1. Aritmética modular

Wonder

Propiedades

Si $a_1 \equiv b_1 \mod m$ y $a_2 \equiv b_2 \mod m$, entonces:

- $a_1 + a_2 \equiv b_1 + b_2 \mod m$
- $a_1 - a_2 \equiv b_1 - b_2 \mod m$
- $a_1 a_2 \equiv b_1 b_2 \mod m$
- $\frac{a_1}{a_2} \equiv b_1 b_2^{-1} \mod m$ (siendo a_2 y m coprimos)
- $a_1^k \equiv b_1^k \mod m$
- $a^{p-2} a \equiv 1 \mod p$ (p primo) [inverso multiplicativo]

Con estas propiedades podemos justificar correctitud al aplicar módulo mientras operamos.

Lemas

- Teorema de Euler: $a^{\varphi(n)} \equiv 1 \mod n$
- Pequeño teorema de Fermat: $a^{p-1} \equiv 1 \mod p$ (con p primo que no divide a a)

Potencia modular

El siguiente algoritmo calcula a^b en módulo de manera eficiente:

```
ll expmod(ll a, ll b) { // O(log b)
    ll acc = 1; // identity element, can be simulated with an if
    for ( ; b; b /= 2) {
        if (b % 2) acc = (acc * a) % MOD;
        a = (a * a) % MOD;
    }
    return acc;
}
```

Podemos pensar que el algoritmo está recorriendo a en binario y por cada k-esima posición con un 1, acumulamos en el resultado a^{2^k}. En otras palabra, estamos calculando el resultado para las potencias de dos que conforman al número b, y combinando todo en nuestro resultado final.

El algoritmo puede generalizarse para calcular $a \circ a \circ \cdots \circ a$ con $\circ$ asociativa, y tal que el operador módulo sea distributivo con respecto a la misma.

```
ll opmod(ll a, ll b) { // // O(log b) [change OP and ID]
    ll acc = ID;
    for ( ; b; b /= 2) {
        if (b % 2) acc = (acc == ID ? a : OP(acc, a)) % MOD;
        a = OP(a, a) % MOD;
    }
    return acc;
}
```

5.2. Teoría de juegos

Juegos combinatorios

Los juegos combinatorios son juegos, de **dos jugadores** que **alternan movimientos**, con **información perfecta** y que necesariamente **terminan con la victoria de uno de los participantes**. Los juegos combinatorios imparciales son aquellos en donde cada jugador tiene disponible el **mismo conjunto de jugadas** en cada configuración posible del juego. Se dice que utilizan reglas *normales* si el última en jugar gana, y *misere* de lo contrario. De estos juegos son de los que vamos a hablar en esta sección.

Veamos para qué juegos conocidos aplican estas definiciones. El poker no lo es porque tiene azar. La batalla naval tampoco porque se esconde información del contrincante. El tatetí tampoco porque puede terminar en empate. Las damas, el ajedrez y el go sí son juegos combinatorios, pero a pesar de que lo parezcan, no son imparciales. Dada una disposición particular del tablero, cada jugador tiene distintos movimientos, pues cada jugador tiene sus propias piezas. Para que fuesen imparciales, un jugador debería poder mover también las piezas de su contrincante. Al final es dificil encontrar ejemplos. Los que se ven como casos introductorios suelen ser el *juego de la reducción* y *Nim*.

Una definición alternativa para juegos combinatorios imparciales sería que son juegos que pueden representarse con un digrafo progresivamente acotado, o sea, que desde cualquier nodo terminamos llegando sí o sí a un nodo saliente en una cantidad finita de pasos (nodo = configuración, arista = movimiento). Jugando con reglas normales donde el último que juega gana, los nodos sin aristas serían posiciones perdedoras.

Posiciones P y N

Una característica importante para los juegos combinatorios es que para cada nodo puedo decir quien gana si los jugadores juegan de forma perfecta. Dado un nodo, decimos que cada nodo es N si gana el primer jugador y P si gana el segundo.

Una estrategia para resolver un juego es arrancar de los nodos terminales e inducir para atrás quien gana (un nodo no terminal es P si y solo si todos sus sucesores son N). O sea, si en un juego encontramos cierta propiedad que cuando se cumple para una configuración, no se cumple para ningún movimiento del contricante, pero la podemos hacer valer de vuelta en el siguiente movimiento, dichas configuraciones son buenos candidato para ser un nodos N (lo que falta es que los nodos terminales sean N y que validen la propiedad) [ejemplo: Nim de 2 pilas, en donde la estrategia ganadora es jugar lo que jugó el contricante pero en la otra pila, y donde la propiedad sería que ambas pilas tengan la misma cantidad de fichas].

Nim

El juego de Nim es un juego donde tengo n pilas con fichas, y un movimiento legal es elegir una pila y substraer alguna cantidad de fichas. El juego termina cuando ya no quedan fichas en ninguna pila.

Teorema de Bouton: Una posición $(f_1, f_2, \ldots, f_n)$ en Nim es P si y solo si $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 0$.

Sprague-Gundy

La función de Sprague-Gundy le asigna recursivamente un entero no-negativo a cada nodo de un juego combinatorio imparcial:

$$sg(x) = mex\{sg(y) : \forall y \in Sucessors(x)\}$$

Para juegos con reglas *normales*, vale que $sg(x) = 0$ si y solo si x es P.

Sumas de juegos

Definimos una suma de juegos como el juego que consiste en jugar n juegos combinatorios independientes, donde el juego termina cuando en ninguno es posible mover, y gana el último jugador.

Teorema de Sprague-Gundy: Si sg_i es la función es Sprague-Gundy del juego G_i, entonces la función de Sprague-Gundy de la suma de los G_i es $sg(x_1, \ldots, x_n) = sg_1(x_1) \oplus \cdots \oplus sg_n(x_n)$.

6. Miscelaneas

6.1. Referencias

Funciones Grandes

	5	**10**	**15**	**20**	**25**	**30**
$\mathbf{2^n}$	32	10^3	$3*10^4$	10^6	$3*10^7$	10^9
$\mathbf{3^n}$	243	$5*10^4$	10^7	$3*10^9$	...	...
$\binom{\mathbf{n}}{\mathbf{n/2}}$	10	252	$5*10^4$	10^5	10^6	10^8
$\mathbf{n!}$	120	10^{12}	...	...	...	...

Límites

INT_MAX $= 2e9$ LLONG_MAX $= 9e18$ DBL_MAX $= 1e37$

Memoria

	ints
$\mathbf{10^3}$	4 *KB*
$\mathbf{10^6}$	4 *MB*
$\mathbf{10^9}$	4 *GB*

6.2. Estrategias

General

- **Simplificación:** Resolver una versión simplificada del problema. Suele ser más facil, y dar pauta de en donde se encuentra la dificultad en el problema original. También puede servir para encontrar una manera de construir la solución.
- **Fuerza Bruta:** Resolver el problema de forma pavota. Está bueno para tener ya una resolución posible, y para tener más en claro qué tan óptimo tiene que ser la respuesta.

Geometria

- Fijar grados de libertad de ser posible (desplazamiento, rotación)

Programación

- ¡Siempre vale la pena pensar un poco más para tratar de hacer el código más sencillo!

6.3. Comandos

Compilación

```
g++ -std=c++17 -O2 -Wconversion -Wall -Wshadow -D_GLIBCXX_DEBUG rta.cpp -o rta
```

Límite de pila (256MB)

```
ulimit -s 262144
```