Contents

C++

Plantilla

Plantilla para cualquier programa en C++.

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    return 0;
}
```

La principal ventaja de esta plantilla es que el *include* que usa trae toda la libreria estándar, yo siempre uso *cin* y *cout* así que esta plantilla incluye optimizaciones para el manejo de entrada. También hay que recordar que '$\backslash n$' es más rápido que *endl*.

Compilación

Recomiendo este comando para compilar, a menos que el concurso sugiera uno.

```
g++ -O2 -Wall test.cpp
```

Con esto se optimiza el código además de indicar que se muestren la mayoría de advertencias.

Manejo de archivos

Si el concurso requiere que se lea o escriba en archivos, se puede agregar esto al principio.

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Con esto ya no se ocupan modificar más cosas.

Tipos de dato y sus rangos

Tipo de dato	Tamaño (en bytes)	Rango
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long long int	8	-(2^63) to (2^63) - 1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
float	4	-3.4 × 10^38 to 3.4 × 10^38
double	8	-1.7 × 10^308 to 1.7 × 10^308
long double	12	-1.1 × 10^4932 to 1.1 × 10^4932

Algo a considerar sobre *long long* es que tiene un prefijo.

```
long long x = 123456789123456789LL;
```

También pasa algo parecido a la división entera cuando se multiplican dos enteros hacia un *long long*, pasa que, a pesar de que esa variable es *long long*, el resultado de multiplicar dos enteros es un entero, y no se guarda bien.

Números flotantes

En la mayoría de los casos con usar *double* es suficiente, si eso no basta se puede usar *long double*. Pero algo a tener en cuenta cuando se usan números flotantes es que compararlos con == no siempre funciona, esto por pequeños *errores* de precisión, por lo que una forma de compararlos es ver si la diferencia entre ellos es lo suficientemente pequeña.

```
if (abs(a - b) < 1e-9) {
    // a and b are equal
}
```

Estimando la eficiencia

Aquí hay una tabla para tener una idea de la complejidad necesaria según el tamaño de la entrada.

tamaño de entrada	complejidad requerida
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \cdot log(n))$ o $O(n)$
n es grande	$O(1)$ o $O(log(n))$

Esto no es para tomarlo como la verdad absoluta, pero puede servir para descartar ideas sin desperdiciar tiempo.

Matemáticas

Aritmética modular

Calcular un exponente grande modulo de un número ($x^n mod m$) se puede hacer con exponenciación binaria.

```
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

Sucesión simple

Fórmulas útiles para calcular la suma de números consecutivos, hay una fórmula mucho más general pero no la entendí.

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$$

Sucesión aritmetica

Otra cosa es una sucesión aritmetica, donde la diferencia entre dos números cualesquiera es la misma.

$$3, 7, 11, 15$$

En esta sucesión aumentan de 4 en 4. La formula se ve así.

$$a + \cdots + b = \frac{n(a+b)}{2}$$

Es esta fórmula a es el primer número, b el segundo y n es la cantidad de números, se puede ver que en la fórmula no se considera el tamaño de los saltos.

Sucesión geometrica

Esta sucesión es una secuencia de números donde la proporción entre dos números consecutivos es la misma.

$$a + ak + ak^2 + \cdots + b = \frac{bk - a}{k - 1}$$

Tabla de verdad

A	B	not A	not B	A and B	A or B	A implies B	A equals B
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Factorial

Se puede definir iterativamente.

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \ldots n$$

O recursivamente.

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

Números de Fibonacci.

Se define recursivamente así.

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

También hay una fórmula directa para calcularlos, pero es poco práctica porque requiere un gran nivel de precisión.

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Hay una forma de calcular el n-th número de Fibonacci en $O(n)$.

```
int fib(int n) {
    int a = 0;
    int b = 1;
    for (int i = 0; i < n; i++) {
        int tmp = a + b;
        a = b;
        b = tmp;
    }
    return a;
}
```

Y hay una de hacerlo en $O(log(n))$.

```
struct matrix {
    long long mat[2][2];
    matrix friend operator *(const matrix &a, const matrix &b) {
        matrix c;
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c.mat[i][j] = 0;
                for (int k = 0; k < 2; k++) {
                    c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
                }
            }
        }
        return c;
    }
};

matrix matpow(matrix base, long long n) {
    matrix ans{
        {
            { 1, 0 },
            { 0, 1 }
        }
```

```
    };
    while (n) {
        if(n & 1)
            ans = ans * base;
        base = base * base;
        n >>= 1;
    }
    return ans;
}

long long fib(int n) {
    matrix base{
        {
            { 1, 1 },
            { 1, 0 }
        }
    };
    return matpow(base, n).mat[0][1];
}
```

Logaritmos

Propiedades de los logaritmos.

$$log_k(x) = a \implies k^a = x$$

$$log_k(ab) = log_k(a) + log_k(b)$$

$$log_k(x^n) = n \cdot log_k(x)$$

$$log_k(\frac{a}{b}) = log_k(a) - log_k(b)$$

Otra propiedad interesante es que la cantidad de digitos de un entero x en base b es:

$$log_b(x) + 1$$

Promedio

Se puede sumar o restar un número a un promedio ya calculado sin tener que recalcular la suma original.

$$s = \frac{a_1 + \cdots + a_n}{n}$$

$$s' = \frac{a_1 + \cdots + a_n + a_{n+1}}{n+1} = \frac{ns + a_{n+1}}{n+1} = \frac{(n+1)s + a_{n+1}}{n+1} - \frac{s}{n+1} = s + \frac{a_{n+1} - s}{n+1}$$

$$s'' = \frac{a_1 + \cdots + a_{n-1}}{n-1} = \frac{ns - a_n}{n-1} = \frac{(n-1)s + a_n}{n-1} + \frac{s}{n-1} = s + \frac{s - a_n}{n-1}$$

Exponenciación binaria

Es una forma de calcular a^n usando $O(log_2(n))$ multiplicaciones en lugar de $O(n)$. No solo sirve para la aritmetica, ya que se puede aplicar a cualquier operación que tenga propiedad asociativa, es decir.

$$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$$

La idea de exponenciación binaria es dividir el trabajo usando la representación binaria del exponente.

Por ejemplo:

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Y debido a que n tiene $log_2(n) + 1$ digitos en base 2, solo se necesitan $O(log_2(n))$ multiplicaciones. La función *pow* de C++ ya hace esto. Pero tiene otras aplicaciones. Como:

Exponentes con módulo

Calcular $x^n mod m$ se puede hacer con exponenciación binaria, eso esta en la sección de Aritmética Modular

Aplicar permutaciones

Se puede usar la exponenciación binaria para aplicar una determinada permutaciíon n veces.

Por ejemplo:

Secuencia original: [1, 2, 3] Permutación: [1, 2, 0]

Si la aplicamos 3 veces obtenemos:

1era aplicación: [2, 3, 1] 2da aplicación: [3, 1, 2] 3era aplicación: [1, 2, 3]

Lo cual se puede hacer con solo dos aplicaciones si se usa exponenciación binaria.

```
vector<int> applyPermutation(vector<int> sequence, vector<int> permutation) {
    vector<int> newSequence(sequence.size());
    for(int i = 0; i < sequence.size(); i++) {
        newSequence[i] = sequence[permutation[i]];
    }
    return newSequence;
}

vector<int> permute(vector<int> sequence, vector<int> permutation, long long b) {
    while (b > 0) {
        if (b & 1) {
            sequence = applyPermutation(sequence, permutation);
        }
        permutation = applyPermutation(permutation, permutation);
        b >>= 1;
    }
    return sequence;
}
```

Números primos

Generar

La criba de Eratóstenes es una forma de calcular los números primos en el intervalo de $[1; n]$ con complejidad $O(n \cdot log(log(n)))$.

Su implementación es:

```
int n;
vector<bool> is_prime(n + 1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

Prueba

Hay una forma deterministica de comprobar si un número de hasta 64 bits es primo.

```
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool MillerRabin(u64 n) { // returns true if n is prime, else returns false.
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
```

```
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

Factores primos

La forma más simple de obtener los factores primos de un número es pregenerando los primos desde 1 hasta $\sqrt{n}$ y probando con cada uno de ellos.

```
vector<long long> primes;

vector<long long> trial_division4(long long n) {
    vector<long long> factorization;
    for (long long d : primes) {
        if (d * d > n)
            break;
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

Máximo común divisor

A partir de C++17 ya hay una función *gcd* incluida, pero si por alguna razón no esta, aquí esta la declaración:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Una implementación recursiva:

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}
```

Y una iterativa:

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

Mínimo común múltiplo

Igual que el *gcd* ya debería de haber una funcín *lcm* en C++, pero por si llega a ser necesaria aquí esta su definición:

$$lcm(a,b) = \frac{a \cdot b}{\gcd(a,b)}$$

Y aquí su implementación:

```
int lcm (int a, int b) {
    return a / gcd(a, b) * b;
}
```

Ordenar

Casi nunca es una buena idea usar algoritmos de ordenamiento que hayas hecho a mano, esto porque ya hay buenas implementaciones en los lenguajes de programación. Por ejemplo, la STL de C++ ya tiene una función *sort* que puede ser usada para ordenar (es una combinación de quicksort, heapsort, insertion sort), también existe *stable_sort* en caso de que se necesite un ordenamiento estable. Aquí hay un ejemplo de esta función en acción.

Con *sort* de C++

```
vector<int> v = { 4, 2, 5, 3, 5, 8, 3 };
sort(v.begin(), v.end());
```

Después de este ordenamiento, el contenido del vector será [2, 3, 3, 4, 5, 5, 8]. Por defecto se ordena de menor a mayor, pero una forma de implementar un orden inverso es:

```
sort(v.rbegin(), v.rend());
```

Estructuras definidas

Las estructuras que nosotros hagamos no tienen un operador de comparación automaticamente. Este operador se puede definir dentro de la estructura como una función de nombre *operator<* cuyo parametro debe ser un elemento del mismo tipo, debe devolver *true* si el elemento es más pequeño que el parametro y *falso* si no es así. Un ejemplo:

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x)
            return x < p.x;
        else
            return y < p.y;
    }
};
```

Con funciones personalizadas

También se puede dar como parametro una función externa para que se use dentro del *sort*. Por ejemplo:

```
bool comp(string a, string b) {
    if (a.size() != b.size())
        return a.size() < b.size();
```

```
    return a < b;
}
```

Ahora usando esa función para un vector de strings:

```
sort(v.begin(), v.end(), comp);
```

Búsqueda binaria

Forma manual

Hay dos formas de implementar la búsqueda binaria por nuestra cuenta.

```
int a = 0, b = n - 1;
while (a <= b) {
    int k = (a + b) / 2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x)
        b = k - 1;
    else
        a = k + 1;
}
```

Y.

```
int k = 0;
for (int b = n / 2; b >= 1; b /= 2) {
    while (k + b < n && array[k + b] <= x)
        k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

Funciones de C++

La libreria estandar de C++ contiene las siguientes funciones que estan basadas en búsqueda binaria y por lo tanto funcionan en tiempo logaritmico:

- lower_bound : devuelve un apuntador al primer elemento que vale al menos x.
- upper_bound : devuelve un apuntador al primer elemento que vale más que x.
- equal_range : devuelve los dos apuntadores de arriba.

Estas funciones asumen que el arreglo esta ordenado. Si no encuentran el elemento devuelven un apuntador pasado el último elemento.

Manipulación de bits

Es el manejo de números binarios.

Operadores binarios

- & : Es el AND.
- | : Es el OR.
- $\wedge$: Es el XOR.

- $\sim$: Es el NOT.

Ejemplos:

```
n          = 01011000
n-1        = 01010111
--------------------
n & (n-1)  = 01010000
```

```
n          = 01011000
n-1        = 01010111
--------------------
n | (n-1)  = 01011111
```

```
n          = 01011000
n-1        = 01010111
--------------------
n ^ (n-1)  = 00001111
```

```
n          = 01011000
--------------------
~n         = 10100111
```

Desplazamientos

- $\gg$: Desplaza el número a la derecha removiendo los últimos bits, esto es efectivamente lo mismo que dividir a la mitad. Ejemplo: $5 \gg 2 = 101_2 \gg 2 = 1_2 = 1$ o lo que es igual a $\frac{5}{2^2}$. Aunque sea una división es mucho más rápido que dividir tradicionalmente.
- $\ll$: Desplaza el número a la izquierda añadiendo bits vacios, es lo mismo a duplicar un número. Ejemplo: $5 \ll 2 = 101_2 \ll 2 = 10100_2 = 20$ o lo que es igual a $5 \cdot 2^2$.

Trucos útiles

Se puede asignar, invertir o limpiar un bit usando las siguientes propiedades: $1 \ll x$ es un número que solo tiene el bit x activo mientras que $\sim (1 \ll x)$ es un número con todos los bits excepto x activos. Esto nos lleva a:

- $n|(1 \ll x)$ activa el x-th bit en el número n.
- $n \wedge (1 \ll x)$ invierte el x-th bit en el número n.
- $n\& \sim (1 \ll x)$ limpia el x-th bit del número n.

Se puede revisar si un bit x esta activo con:

```cpp
bool is_set(unsigned int number, int x) {
    return (number >> x) & 1;
}
```

Se puede limpiar el bit más a la derecha de un número haciendo un AND con su predecesor.

```
n           = 00110100
n - 1       = 00110011
----------------------
n & (n - 1) = 00110000
```

Funciones de C++

- has_single_bit : Revisa si el número es una potencia de 2.
- bit_ceil / bit_floor : Redondea a la siguiente potencia de 2.
- popcount : Cuenta la cantidad de bits activos.

Submascaras de una mascara

Dada una mascara m se quieren iterar por todas sus submascaras, es decir, mascaras s en las que solo bits que se incluían en m estan activos.

```
for (int s=m; s; s=(s-1)&m)
    ... you can use s ...
```

Esto no incluye la submascara equivalente a 0.

Manejo de números grandes

Para el manejo de números grandes se va a usar un arreglo donde se guarden sus "digitos".

```
// Base a usar
const int base = 1000 * 1000 * 1000;

// Convertir de una string al vector de "digitos"
for (int i = (int) s.length(); i > 0; i -= 9) {
    if (i < 9) {
        a.push_back(atoi(s.substr(0, i).c_str()));
    } else {
        a.push_back(atoi(s.substr(i - 9, 9).c_str()));
    }
}

// Borrar leading 0, combiene hacerlo después de la mayoría de operaciones
while (a.size() > 1 && a.back() == 0) {
    a.pop_back();
}

// Suma a + b resultado en a
int carry = 0;
for (size_t i = 0; i < max(a.size(), b.size()) || carry; ++i) {
    if (i == a.size()) {
        a.push_back(0);
    }
    a[i] += carry + (i < b.size() ? b[i] : 0);
    carry = a[i] >= base;
    if (carry) {
        a[i] -= base;
    }
}

// Resta a - b y guarda en a
int carry = 0;
for (size_t i = 0; i < b.size() || carry; ++i) {
    a[i] -= carry + (i < b.size() ? b[i] : 0);
    carry = a[i] < 0;
    if (carry) {
        a[i] += base;
    }
}

// Multiplicar a por un entero b pequeño (b < base) y guardar en a
```

```cpp
int carry = 0;
for (size_t i = 0; i < a.size() || carry; ++i) {
    if (i == a.size()) {
        a.push_back(0);
    }
    long long cur = carry + a[i] * 1ll * b;
    a[i] = int (cur % base);
    carry = int (cur / base);
}

// Multiplicar a por un entero largo b y guardar en c
vector<int> c(a.size() + b.size());
for (size_t i = 0; i < a.size(); ++i) {
    for (int j = 0, carry = 0; j < (int) b.size() || carry; ++j) {
        long long cur = c[i + j] + a[i] * 1ll * (j < (int) b.size() ? b[j] : 0) + carry;
        c[i + j] = int (cur % base);
        carry = int (cur / base);
    }
}

// División de a entre un entero b pequeño (b < base)
int carry = 0;
for (int i = (int) a.size() - 1; i >= 0; --i) {
    long long cur = a[i] + carry * 1ll * base;
    a[i] = int (cur / b);
    carry = int (cur % b);
}

// Para imprimir
if (a.empty()) {
    cout << 0;
} else {
    cout << a.back();
}

for (int i = (int) a.size() - 2; i >= 0; --i) {
    cout << setw(9) << setfill('0') << a[i];
}
```

Estructuras de datos

Policy Based Data Structures

Es una estructura muy útil, es básicamente un *set* (con inserción y borrado en $O(log(n))$) pero con índices.

```cpp
#include <ext/pb_ds/assoc_container.hpp>

using namespace __gnu_pbds;

typedef tree<
    pair<unsigned long long, int>,
    null_type,
    less<pair<unsigned long long, int>>,
    rb_tree_tag,
```

```
    tree_order_statistics_node_update> indexed_set;
```

Esta implementación es para un multiset, por eso se almacena un *pair*, pero si se necesita que no haya repeticiones se sustituye el *pair* por el tipo de dato. También esta ordenado de menor a mayor, si se necesita lo opuesto se cambia el *less*.

```
indexed_set is;

is.order_of_key(key); // Regresa el índice que esa key tendría dentro del set, exista o no
is.find_by_order(order); // Regresa un apuntador al índice que se uso como parametro
```

Minimum stack / Minimum queue

Aquí hay modificaciones al stack y a la fila para además de tener sus características poder acceder al menor elemento en $O(1)$.

Minimum stack

```
stack<pair<int, int>> st;

// Agregar elemento
int new_min = st.empty() ? new_elem : min(new_elem, st.top().second);
st.push({ new_elem, new_min });

// Remover elemento
int removed_element = st.top().first;
st.pop();

// Encontrar el mínimo
int minimum = st.top().second;
```

Minimum queue

```
stack<pair<int, int>> s1, s2;

// Encontrar el mínimo
if (s1.empty() || s2.empty())
    minimum = s1.empty() ? s2.top().second : s1.top().second;
else
    minimum = min(s1.top().second, s2.top().second);

// Añadir elemento
int minimum = s1.empty() ? new_element : min(new_element, s1.top().second);
s1.push({new_element, minimum});

// Remover elemento
if (s2.empty()) {
    while (!s1.empty()) {
        int element = s1.top().first;
        s1.pop();
        int minimum = s2.empty() ? element : min(element, s2.top().second);
        s2.push({element, minimum});
    }
}
int remove_element = s2.top().first;
s2.pop();
```

Problemas clásicos

Suma máxima de subarreglo

Aquí se habla de un problema clásico cuya solución más *directa* es O(n^3), pero que pensandola mejor se puede lograr reducir a O(n). Dado un arreglo de n númeors, hay que calcular cual es la suma máxima de una subsección, osea la suma más grande de una secuencia consecutiva de valores en el arreglo, esto se vuelve más interesante si el arreglo puede tener números negativos. Aquí hay un ejemplo.

-1 | 2 | 4 | -3 | 5 | 2 | -5 | 2 |

El siguiente subarreglo produce una suma de 10:

-1	2	4	-3	5	2	-5	2
	^	^	^	^	^		

Asumimos que un subarreglo vacio esta permitido, por lo que la suma máxima siempre será al menos 0. Ahora, para resolverlo en O(n) partimos de la idea de que para cada posición queremos calcular la suma máxima posible hasta ahí, entonces la del arreglo completo será la mayor de todas esas sumas. Consideramos dos posibilidades para la máxima suma del arreglo que termina en la posición k:

1. El subarreglo solo contiene el elemento de la posición k.
2. El subarreglo consiste de un subarreglo que termina en la posición k - 1, seguido del elemento en la posición k.

En el segundo caso, dado que queremos encontrar un subarreglo con la máxima suma, el subarreglo que termina en la posición k - 1 debería tener también la máxima suma para ese punto. Entonces podemos resolver el problema al calcular la máxima suma de subarreglos para cada posición de izquierda a derecha. El siguiente código demuestra una implementación de este algoritmo:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k],sum+array[k]);
    best = max(best,sum);
}
cout << best << '\n';
```

Este algoritmo solo tiene un ciclo, por lo que la complejidad es O(n). Esta es la mejor complejidad posible, ya que cualquier algoritmo para este problema tiene que analizar todos los datos al menos una vez.