

Contents

1	C++	2
1.1	Plantilla básica	2
1.2	Compilación	2
1.3	Manejo de archivos	2
1.4	Tipos de dato y sus rangos	2
1.5	Números flotantes	3
1.6	Estimación de eficiencia	3
2	Funciones y estructuras útiles	3
2.1	Manejo de fechas	3
2.2	Imprimir tiempo	5
2.3	Exponenciación binaria mod n	5
3	Matemáticas	5
3.1	Sucesión simple	5
3.2	Sucesión aritmetica	5
3.3	Sucesión geometrica	6
3.4	Factorial	6
3.5	Números de Fibonacci	6
3.6	Logaritmos	7
3.7	Promedio	8
3.8	Generar números primos	8
3.9	Prueba de primalidad	8
3.10	Factores primos	9
3.11	Máximo común divisor	10
3.12	Mínimo común múltiplo	10
3.13	Ceil/Floor	10
3.14	Cantidad de múltiplos de un número	11
4	Ordenamiento	11
4.1	Estructuras definidas	11
4.2	Con funciones personalizadas	11
4.3	Permutaciones	12
4.4	Ordenamiento parcial	12

1 C++

1.1 Plantilla básica

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

using namespace std;

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

    return 0;
}
```

1.2 Compilación

```
g++ -O2 -Wall -std=c++11 test.cpp
```

1.3 Manejo de archivos

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

1.4 Tipos de dato y sus rangos

Tipo de dato	Tamaño (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 int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	$-(2^{63})$ to $(2^{63})-1$
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
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 es que *long long* tiene un prefijo.

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

Pasa algo parecido a la división entera, el resultado de multiplicar dos enteros es un entero.

1.5 Números flotantes

Existen los errores de precisión, para tratar de contrarestarlos se puede usar esta lógica.

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

1.6 Estimación de eficiencia

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

2 Funciones y estructuras útiles

2.1 Manejo de fechas

```
struct Date {  
    int day, month, year;  
    vector<vector<int>> calendar = {  
        { 0, 0 },  
        { 31, 31 },  
        { 28, 29 },  
        { 31, 31 },  
        { 30, 30 },  
        { 31, 31 },  
        { 30, 30 },  
        { 31, 31 },  
        { 31, 31 },  
        { 30, 30 },  
        { 31, 31 },  
        { 30, 30 },  
        { 31, 31 },  
        { 31, 31 },  
    };  
    map<string, vector<int>> zodiacs = {  
        { "aquarius" , { 1, 21, 2, 19 } },  
        { "pisces" , { 2, 20, 3, 20 } },  
        { "aries" , { 3, 21, 4, 20 } },  
        { "taurus" , { 4, 21, 5, 21 } },  
        { "gemini" , { 5, 22, 6, 21 } },  
        { "cancer" , { 6, 22, 7, 22 } },  
        { "leo" , { 7, 23, 8, 21 } },  
        { "virgo" , { 8, 22, 9, 23 } },  
        { "libra" , { 9, 24, 10, 23 } },  
        { "scorpio" , { 10, 24, 11, 22 } },  
        { "sagittarius" , { 11, 23, 12, 22 } },  
    };
```

```

    { "capricorn" , { 12, 23, 1, 20 } },
};

Date(int dd, int mm, int yy) {
    day = dd;
    month = mm;
    year = yy;

    add_days(0);
}

void add_days(int days_to_add) {
    day += days_to_add;
    int leap = is_leap();

    while (day > calendar[month][leap]) {
        day -= calendar[month][leap];
        month++;
        if (month > 12) {
            month = 1;
            year++;
            leap = is_leap();
        }
    }
}

int is_leap() {
    if (year % 400 == 0) {
        return 1;
    }
    if (year % 4 == 0 && year % 100 != 0) {
        return 1;
    }
    return 0;
}

string zodiac() {
    int di, df, mi, mf;
    map<string, vector<int>>::iterator it;
    for (it = zodiacs.begin(); it != zodiacs.end(); it++) {
        di = it->second[1];
        mi = it->second[0];
        df = it->second[3];
        mf = it->second[2];

        if ((month == mi && day >= di) || (month == mf && day <= df))
            return it->first;
    }
    return "";
}

```

```
};
```

2.2 Imprimir tiempo

Aquí hay una función para imprimir en formato HH:MM:SS a partir de la cantidad de segundos.

```
void printTime(int seconds) {
    int hours = seconds / 3600;
    int minutes = (seconds % 3600) / 60;
    int secs = seconds % 60;

    cout << setfill('0') << setw(2) << hours << ":"
        << setfill('0') << setw(2) << minutes << ":"
        << setfill('0') << setw(2) << secs << "\n";
}
```

2.3 Exponenciación binaria mod n

$$a^b \bmod m$$

```
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;
}
```

3 Matemáticas

3.1 Sucesión simple

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

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

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

3.2 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.

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

3.3 Sucesión geométrica

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

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

3.4 Factorial

Se puede definir iterativamente.

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

O recursivamente.

$$0! = 1$$

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

3.5 Números de Fibonacci

Se define recursivamente así.

$$f(0) = 0$$

$$f(1) = 1$$

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

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];
}

```

3.6 Logaritmos

Propiedades de los logaritmos.

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

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

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

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

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

$$\log_b(x) + 1$$

3.7 Promedio

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

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

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

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

3.8 Generar números primos

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;
    }
}
```

3.9 Prueba de primalidad

Forma determinística 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) { // Main func
    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;
}

```

3.10 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;
}

```

3.11 Máximo común divisor

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

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;
}

```

3.12 Mínimo común múltiplo

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

Y aquí su implementación:

```

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

```

3.13 Ceil/Floor

Estas funciones a veces pueden tener problemas de precisión, por lo que puede ser mejor aplicarlas de forma que siempre se mantienen enteros.

Floor suele ser innecesario porque ese es el comportamiento por defecto de C++ al dividir enteros.

Ceil se puede sustituir por:

$$\lceil \frac{a}{b} \rceil = \lfloor \frac{a + b - 1}{b} \rfloor$$

Esta igualdad se cumple para enteros positivos, para enteros negativos no estoy muy seguro.

3.14 Cantidad de múltiplos de un número

Para problemas donde se quiera saber por ejemplo, cuantos múltiplos de 2 hay del 1 al 100 se puede responder dividiendo este número entre la base.

Para el caso de múltiplos de dos números, por ejemplo, cuantos números hay del 1 al 100 que sean múltiplos de 2 y de 5 se usa el mínimo común múltiplo como base.

4 Ordenamiento

Existen las funciones *sort* y *stable_sort*.

```
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());
```

4.1 Estructuras definidas

Las estructuras que nosotros hagamos no tienen un operador de comparación automáticamente. 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 *false* si no es así. Un ejemplo:

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

4.2 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);
```

4.3 Permutaciones

Una forma simple de pasar por todas las permutaciones de algo es con *next_permutation*, pero primero se ordena lo que se vaya a permutar. Ejemplo:

```
letras = "cba";
sort(letras.begin(), letras.end());
do {
    cout << letras << endl;
} while (next_permutation(letras.begin(), letras.end()));
```

4.4 Ordenamiento parcial

En caso de que no se necesite ordenar todos los datos se pueden hacer dos tipos de ordenamientos parciales. Para ordenar de modo que los primeros N elementos esten ordenados se puede usar:

```
partial_sort( RandomIt first, RandomIt middle, RandomIt last );
```

Esta función se asegura que los elementos hasta *middle* esten ordenados, OJO *middle* no es un valor, es un iterador, así que no ordena los menores a *middle* sino los *middle* - *first*.

Otra forma es si se quiere que el enesimo elemento del arreglo este ordenado, para eso se usa:

```
nth_element( RandomIt first, RandomIt nth, RandomIt last );
```

Que igualmente recibe tres iteradores.

Búsqueda binaria

\subsection{Forma manual}

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

```
\begin{lstlisting}
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;
}
\end{lstlisting}
```

Y.

```
\begin{lstlisting}
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
}
\end{lstlisting}

```

\subsection{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.

\subsection{Operadores binarios}

- $\&$: Es el AND.
- $|$: Es el OR.
- \wedge : Es el XOR.
- \sim : Es el NOT.

Ejemplos:

```

\end{lstlisting}
n          = 01011000
n-1        = 01010111
-----
n & (n-1) = 01010000
\end{lstlisting}

```

```

\end{lstlisting}
n          = 01011000
n-1        = 01010111
-----
n | (n-1) = 01011111
\end{lstlisting}

```

```

\end{lstlisting}
n          = 01011000
n-1        = 01010111
-----
n ^ (n-1) = 00001111

```

```
\end{lstlisting}
```

```
\end{lstlisting}
```

```
n          = 01011000
```

```
-----
```

```
~n         = 10100111
```

```
\end{lstlisting}
```

```
\subsection{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 vacíos, es lo mismo a duplicar un número. Ejemplo: $5 \ll 2 = 101_2 \ll 2 = 10100_2 = 20$ o lo que es igual a $5 \cdot 2^2$.

```
\subsection{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 .
- Si $n \& (n - 1)$ es 0 para $n > 0$ entonces n es una potencia de 2.

Se puede revisar si un bit x está activo con:

```
\begin{lstlisting}
```

```
bool is_set(unsigned int number, int x) {
```

```
    return (number >> x) & 1;
```

```
}
```

```
\end{lstlisting}
```

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

```
\end{lstlisting}
```

```
n          = 00110100
```

```
n - 1      = 00110011
```

```
-----
```

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

```
\end{lstlisting}
```

```
# Submascaras de una mascara
```

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

```
\begin{lstlisting}
for (int s=m; s; s=(s-1)&m)
    ... you can use s ...
\end{lstlisting}
```

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".

```
\begin{lstlisting}
// Base a usar
const int base = 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, conviene 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
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] * 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] * b[j] + 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 * b;
    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];
}
\end{lstlisting}

```

Estructuras de datos

`\subsection{Policy Based Data Structures}`

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

```
\begin{lstlisting}
#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;
\end{lstlisting}
```

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

```
\begin{lstlisting}
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
\end{lstlisting}
```

`\subsection{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 $O(1)$.

`#\subsection{Minimum stack}`

```
\begin{lstlisting}
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;
\end{lstlisting}

#\subsection{Minimum queue}

\begin{lstlisting}
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();
\end{lstlisting}

\subsection{Grafos}

```

Hay tres formas principales de representar un grafo.

Matriz de adyacencias

: Es buena elección si constantemente se necesita revisar si dos vertices estan conectados en un grafo denso. Pero no se recomienda para grafos grandes y dispersos porque requiere $O(V^2)$ espacio y habría mucho desperdiciado con celdas en blanco. Usualmente el límite de vertices para una matriz de adyacencias en una competencia sería de 1000, más de esos y ya se vuelve una mala idea. También se necesita $O(V)$ para enumerar todos los vecinos de un vertice.

Lista de adyacencias

: Es una forma más eficiente de representar un grafo, se recomienda sea la primera opción a considerar al encontrarse con problemas de grafos. Su espacio es $O(V + E)$ y se suele representar como un vector de vectores o como un vector de vectores de pares.

Lista de arcos

: Es otra forma en donde se usa un vector de trios (para grafos con peso) o uno de pares si no hay peso y cada entrada representa un arco, tiene espacio $O(E)$ y aunque dificulta el ver los vecinos de cierto nodo puede simplificar ciertos algoritmos.

`\subsection{Union-Find Disjoint Set}`

Es una estructura de datos que permite de forma eficiente determinar que nodos pertenecen al mismo set, así como poder combinar sets. Su implementacion:

```
\begin{lstlisting}
struct UnionFind {
    vector<int> parent, rank;

    unionFind(int N) {
        rank.assign(N, 0);
        parent.assign(N, 0);
        for (int i = 0; i < N; ++i) parent[i] = i;
    }

    int findSet(int i) {
        if (parent[i] == i) return i;

        parent[i] = findSet(parent[i]);

        return parent[i];
    }

    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }

    void unionSet(int i, int j) {
        if (!isSameSet(i, j)) {
            int x = findSet(i);
            int y = findSet(j);

            if (rank[x] > rank[y])
                parent[y] = x;
            else {
                parent[x] = y;
                if (rank[x] == rank[y]) ++rank[y];
            }
        }
    }
};
\end{lstlisting}
```

`\subsection{Segment Tree}`

El árbol de segmentos es una estructura de datos que sirve para responder de

forma eficiente consultas de rangos en arreglos de números que pueden cambiar.

Primero una implementación para obtener el mínimo en cierto rango:

```
\begin{lstlisting}
struct SegmentTree {
    vector<int> st, A;
    int n;

    int left(int p) { return p << 1; }

    int right(int p) { return (p << 1) + 1; }

    void build(int p, int L, int R) {
        if (L == R)
            st[p] = L;
        else {
            build(left(p), L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            int p1 = st[left(p)];
            int p2 = st[right(p)];
            st[p] = (A[p1] <= A[p2]) ? p1 : p2;
        }
    }

    int rmq(int p, int L, int R, int i, int j) {
        if (i > R || j < L) return -1;

        if (L >= i && R <= j) return st[p];

        int p1 = rmq(left(p), L, (L + R) / 2, i, j);
        int p2 = rmq(right(p), (L + R) / 2 + 1, R, i, j);

        if (p1 == -1) return p2;

        if (p2 == -1) return p1;

        return (A[p1] <= A[p2]) ? p1 : p2;
    }

    void pointUpdate(int p, int L, int R, int i, int v) {
        if (L == R) return;

        if (i >= L && i <= R) {
            if (v <= A[st[p]]) st[p] = i;

            pointUpdate(left(p), L, (L + R) / 2, i, v);
            pointUpdate(right(p), (L + R) / 2 + 1, R, i, v);
        }
    }
}
```

```

SegmentTree(vector<int> &_A) {
    A = _A;
    n = (int)A.size();
    st.assign(4 * n, 0);
    build(1, 0, n - 1);
}

int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }

void pointUpdate(int i, int v) {
    A[i] = v;
    pointUpdate(1, 0, n - 1, i, v);
}

void rangeUpdate(int i, int j, int v) {
    for (int k = j; k >= i; --k) pointUpdate(k, v);
}
};
\end{lstlisting}

```

Ahora una implementación para la suma de los elementos en cierto rango:

```

\begin{lstlisting}
struct SegmentTree {
    vector<int> st, A;
    vector<bool> marked;
    int n;

    int left(int p) { return p << 1; }

    int right(int p) { return (p << 1) + 1; }

    void build(int p, int L, int R) {
        if (L == R)
            st[p] = A[L];
        else {
            build(left(p), L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            int p1 = st[left(p)];
            int p2 = st[right(p)];
            st[p] = p1 + p2;
        }
    }

    int rsq(int p, int L, int R, int i, int j) {
        if (i > R || j < L) return 0;

        if (L >= i && R <= j) return st[p];
    }
}
\end{lstlisting}

```

```

    push(p, L, R);

    int p1 = rsq(left(p), L, (L + R) / 2, i, j);
    int p2 = rsq(right(p), (L + R) / 2 + 1, R, i, j);

    return p1 + p2;
}

void push(int p, int L, int R) {
    if (L == R) return;
    if (marked[p]) {
        int tm = (L + R) / 2;
        int v = st[p] / (R - L + 1);
        st[left(p)] = v * (tm - L + 1);
        st[right(p)] = v * (R - tm);
        marked[left(p)] = marked[right(p)] = true;
        marked[p] = false;
    }
}

void rangeUpdate(int p, int L, int R, int i, int j, int v) {
    if (i > R || j < L) return;
    if (L >= i && R <= j) {
        st[p] = v * (R - L + 1);
        marked[p] = true;
        return;
    }
    push(p, L, R);
    rangeUpdate(left(p), L, (L + R) / 2, i, j, v);
    rangeUpdate(right(p), (L + R) / 2 + 1, R, i, j, v);
    st[p] = st[left(p)] + st[right(p)];
}

SegmentTree(vector<int>& _A) {
    A = _A;
    n = (int)A.size();
    st.assign(4 * n, 0);
    marked.assign(4 * n, false);
    build(1, 0, n - 1);
}

int rsq(int i, int j) { return rsq(1, 0, n - 1, i, j); }

void rangeUpdate(int i, int j, int v) { rangeUpdate(1, 0, n - 1, i, j, v); }
};
\end{lstlisting}

\subsection{Fenwick Tree}

```

Muy similar al Segment Tree, por ahora creo que le daré prioridad al Segment

Tree en lugar de este pero quisiera explorarlo mejor después.

```
\begin{lstlisting}
struct FenwickTree {
    vector<int> ft;

    int LSOne(int S) { return (S & (-S)); }

    FenwickTree(int n) { ft.assign(n + 1, 0); }

    int rsq(int b) {
        int sum = 0;
        for (; b -= LSOne(b)) sum += ft[b];

        return sum;
    }

    int rsq(int a, int b) { return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }

    void adjust(int k, int v) {
        for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v;
    }
};
\end{lstlisting}
```

Problemas de ejemplo

\subsection{Suma máxima de subarreglo}

Aquí se habla de un problema clásico cuya solución más `textit{directa}` es $O(n^3)$, pero que pensandola mejor se puede lograr reducir a $O(n)$. Dado un arreglo de n números, 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 vacío está 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 `textit{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:

```
\begin{lstlisting}
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';
\end{lstlisting}
```

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.

\subsection{Segment Tree para arreglo binario}

Hay un arreglo binario donde se necesita saber cuantos 1's hay en determinado rango, también es necesario poder actualizar un rango así como poder invertir un rango (cambiar 0 \rightarrow 1 y 1 \rightarrow 0).

```
\begin{lstlisting}
struct SegmentTree {
    vector<int> st, A, marked;
    int n;

    int left(int p) { return p << 1; }

    int right(int p) { return (p << 1) + 1; }

    void build(int p, int L, int R) {
        if (L == R)
            st[p] = A[L];
        else {
            build(left(p), L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            int p1 = st[left(p)];
            int p2 = st[right(p)];
            st[p] = p1 + p2;
        }
    }
}
```



```

int rsq(int p, int L, int R, int i, int j) {
    if (i > R || j < L) return 0;

    if (L >= i && R <= j) return st[p];

    push(p, L, R);

    int p1 = rsq(left(p), L, (L + R) / 2, i, j);
    int p2 = rsq(right(p), (L + R) / 2 + 1, R, i, j);

    return p1 + p2;
}

void push(int p, int L, int R) {
    if (L == R) return;
    int tm = (L + R) / 2;
    if (marked[p] == 1) {
        int v = (st[p] ? 1 : 0);
        st[left(p)] = v * (tm - L + 1);
        st[right(p)] = v * (R - tm);
        marked[left(p)] = marked[right(p)] = 1;
        marked[p] = 0;
    } else if (marked[p] == 2) {
        push(left(p), L, tm);
        push(right(p), tm + 1, R);
        st[left(p)] = (tm - L + 1) - st[left(p)];
        st[right(p)] = (R - tm) - st[right(p)];
        marked[left(p)] = marked[right(p)] = 2;
        marked[p] = 0;
    }
}

void invertRange(int p, int L, int R, int i, int j) {
    if (i > R || j < L) return;
    push(p, L, R);
    if (L >= i && R <= j) {
        st[p] = (R - L + 1) - st[p];
        marked[p] = 2;
        return;
    }
    invertRange(left(p), L, (L + R) / 2, i, j);
    invertRange(right(p), (L + R) / 2 + 1, R, i, j);
    st[p] = st[left(p)] + st[right(p)];
}

void rangeUpdate(int p, int L, int R, int i, int j, int v) {
    if (i > R || j < L) return;
    if (L >= i && R <= j) {
        st[p] = v * (R - L + 1);
        marked[p] = 1;
    }
}

```

```

        return;
    }
    push(p, L, R);
    rangeUpdate(left(p), L, (L + R) / 2, i, j, v);
    rangeUpdate(right(p), (L + R) / 2 + 1, R, i, j, v);
    st[p] = st[left(p)] + st[right(p)];
}

SegmentTree(vector<int>& _A) {
    A = _A;
    n = (int)A.size();
    st.assign(4 * n, 0);
    marked.assign(4 * n, 0);
    build(1, 0, n - 1);
}

int rsq(int i, int j) { return rsq(1, 0, n - 1, i, j); }

void rangeUpdate(int i, int j, int v) { rangeUpdate(1, 0, n - 1, i, j, v); }

void invertRange(int i, int j) { invertRange(1, 0, n - 1, i, j); }
};
\end{lstlisting}

```