

Notas de programación competitiva

Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
<https://racc.mx>
<https://omegaup.com/profile/rcc>

Última revisión: 21 de septiembre de 2022

1 Notas

```
#include <stdint.h>

// int32_t equivale a int
// int64_t equivale a long long
// uint32_t equivale a unsigned
// uint64_t equivale a unsigned long long
// Los tipos sin signo usan aritmética mod. #bits
// size_t (std::size_t) es unsigned long long
```

```
(Linux)
g++ codigo.cpp -O3 -lm -std=c++20 -o codigo
./codigo
```

```
Incluir toda la biblioteca
#include <bits/stdc++.h>
```

```
Optimizar la entrada y la salida
std::ios_base::sync_with_stdio(false);
std::cin.tie(nullptr), std::cout.tie(nullptr);
```

2 Tokenización de líneas

```
int main( ) {
    std::string linea;
    while (std::getline(std::cin, linea)) {
        std::istringstream extractor(linea);
        std::string palabra;
        while (extractor >> palabra) {
            std::cout << palabra << " ";
        }
        std::cout << "\n";
    }
}
```

3 Construcción de cadenas concatenando valores arbitrarios

```
int main( ) {
    int a = 57;
    char c = '@';
    double f = 3.14;
```

```
std::ostringstream bufer;
bufer << a << " " << c << " " << f;
std::string cadena = bufer.str( );
std::cout << cadena;
}
```

4 Potencia rápida de naturales

```
int potencia(int a, int b) {
    if (b == 0) {
        return 1;
    } else {
        int t = potencia(a, b / 2);
        if (b % 2 == 0) {
            return t * t;
        } else {
            return t * t * a;
        }
    }
}
```

5 Potencia rápida de naturales con aritmética modular

```
int64_t potencia(int64_t b, int64_t e, int64_t mod) {
    int64_t res = 1; b %= mod;
    while (e != 0) {
        if (e % 2 == 1) {
            res = (__int128_t(res) * b) % mod;
        }
        b = (__int128_t(b) * b) % mod;
        e /= 2;
    }
    return res;
}
```

6 Distancia entre dos puntos

```
double distancia(const auto& a, const auto& b) {
    auto dx = a.x - b.x;
    auto dy = a.y - b.y;
```

```

    return std::sqrt(dx * dx + dy * dy);
}

```

7 Algoritmos de Dijkstra y Prim

```

struct entrada {
    int vertice, costo;
};

bool operator<(entrada a, entrada b) {
    return a.costo > b.costo;
}

int main( ) {
    int n, m;
    std::cin >> n >> m;

    std::vector<entrada> adyacencia[n];
    for (int i = 0; i < m; ++i) {
        int x, y, c;
        std::cin >> x >> y >> c;
        adyacencia[x].push_back({ y, c });
        adyacencia[y].push_back({ x, c });
    }

    std::priority_queue<entrada> cp;
    cp.push({0, 0});
    int distancia[n];
    std::fill(&distancia[0], &distancia[n], -1);
    do {
        entrada actual = cp.top( );
        cp.pop( );
        if (distancia[actual.vertice] == -1) {
            distancia[actual.vertice] = actual.costo;
            for (entrada vecino : adyacencia[actual.
                vertice]) {
                cp.push({ vecino.vertice, actual.costo
                    + vecino.costo });
            } // quiar la suma para Prim (dejar
            actual.costo)
        }
    } while (!cp.empty( ));

    for (int i = 0; i < n; ++i) {
        std::cout << i << ": " << distancia[i] << "\n";
    }
}

```

8 Algoritmo de Floyd (distancias más cortas entre cualquier pareja de vértices)

```

// adyacencia[i][j] es la matriz original
// la siguiente implementación la modifica
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {

```

```

            adyacencia[i][j] = std::min(adyacencia[i]
                ][j], adyacencia[i][k] + adyacencia[k][j]);
        }
    }
}

```

9 Coeficiente binomial iterativo

```

unsigned long long binomial(int n, int k) {
    unsigned long long res = 1;
    for (int i = 1; i <= k; ++i) {
        res *= n + 1 - i;
        res /= i;
    }
    return res;
}

```

10 Inverso modular

```

long long primo = X; // X dado por el problema

long long inverso(int n) {
    if (primo == 1) {
        return 0;
    }
    long long m = primo, y = 0, x = 1;
    while (n > 1) {
        int q = n / m, t = m;
        m = n % m;
        n = t;
        t = y;
        y = x - q * y;
        x = t;
    }
    return (x + primo) % primo;
}

```

11 Coeficiente binomial modular

```

long long factorial[N + 1]; // N dado por el
    problema

long long binomial(int n, int k){
    return (((factorial[n] * inverso(factorial[k])
        ) % primo) * inverso(factorial[n - k])) %
        primo;
}

int main( ) {
    for (int i = 0; i <= N; ++i) {
        factorial[i] = (i == 0 ? 1 : (i * factorial[
            i - 1]) % primo);
    }
    //...
}

```

12 Criba de Eratóstenes

```
struct criba {
    int64_t tope;
    std::vector<int64_t> primos, factor;

    criba(int64_t t)
    : tope(t), factor(t + 1) {
        for (int64_t i = 2; i <= tope; ++i) {
            if (factor[i] == 0) {
                primos.push_back(i);
                factor[i] = i;
                for (int64_t j = i * i; j <= tope; j
+= i) {
                    factor[j] = i;
                }
            }
        }
    }
};

int main( ) {
    criba c(N);    // N dado por el problema
    std::cout << c.factor[i];    // un factor de i
    // si i es primo, factor[i] == i
}
```

13 Primalidad Miller Rabin

```
bool primalidad_miller_rabin(int64_t n) {
    if (n < 2) {
        return false;
    }

    int64_t d = n - 1, s = 0;
    while (d % 2 == 0) {
        d /= 2, s += 1;
    }
    auto compuesto_con = [&](int64_t a) {
        int64_t x = potencia(a, d, n);    // potencia
        modular
        if (x == 1 || x == n - 1) {
            return false;
        }
        for (int64_t r = 1; r < s; ++r) {
            x = (__int128_t(x) * x) % n;
            if (x == n - 1) {
                return false;
            }
        }
        return true;
    };

    for (int64_t a : { 2, 3, 5, 7, 11, 13, 17, 19,
        23, 29, 31, 37 }) {
        if (n == a) {
            return true;
        } else if (compuesto_con(a)) {
            return false;
        }
    }
}
```

```
    }
}
return true;
}
```

14 Factorización de números grandes

```
int64_t factor_pollard_rho(int64_t n, int64_t c) {
    int64_t x = 2, y = 2, k = 2;
    for (int64_t i = 2; ; ++i) {
        x = ((__int128_t(x) * x) % n) + c;
        if (x >= n) {
            x -= n;
        }
        int64_t d = std::gcd(x - y, n);
        if (d != 1) {
            return d;
        }
        if (i == k) {
            y = x, k *= 2;
        }
    }
}

// usar una criba con un tope razonable
std::vector<int64_t> factoriza(int64_t n, const
    criba& c) {    // suposición: n >= 2
    std::vector<int64_t> factores;
    if (n <= c.tope) {
        do {
            factores.push_back(c.factor[n]);
            n /= c.factor[n];
        } while (n != 1);
    } else if (n <= c.tope * c.tope) {
        int64_t raiz = std::ceil(std::sqrt(n));
        for (int64_t i = 0; i < c.primos.size( ) &&
            c.primos[i] <= raiz; ++i) {
            while (n % c.primos[i] == 0) {
                factores.push_back(c.primos[i]);
                n /= c.primos[i];
            }
        }
        if (n != 1) {
            factores.push_back(n);
        }
    } else if (primalidad_miller_rabin(n)) {
        factores.push_back(n);
    } else {
        for (int64_t i = 2; ; i++) {
            auto checar = factor_pollard_rho(n, i);
            if (checar != n) {
                std::vector<int64_t> factores1 =
                    factoriza(checar, c);
                std::vector<int64_t> factores2 =
                    factoriza(n / checar, c);
                factores.insert(factores.end( ),
                    factores1.begin( ), factores1.end( ));
                factores.insert(factores.end( ),
                    factores2.begin( ), factores2.end( ));
                break;
            }
        }
    }
}
```

```

    }
}
return factores;
}

```

15 Búsqueda binaria generalizada

```

template<typename T, typename F>
T busqueda_binaria(T ini, T fin, F pred) {
    auto res = fin;
    while (ini != fin) {
        auto mitad = ini + (fin - ini) / 2;
        if (pred(mitad)) {
            res = mitad, fin = mitad;
        } else {
            ini = mitad + 1;
        }
    }

    return res;
}

int main( ) {
    // encontrar el menor entero de 0 a 10^9 que
    // sea mayor o igual a 50
    int v = busqueda_binaria(0, 1000000000, [](int
    checar) {
        return checar >= 50;
    });
}

```

16 Búsqueda de subcadenas en tiempo logarítmico con arreglo de sufijos

```

template<typename RI>
std::vector<int> suffix_ranking(RI si, RI sf) {
    std::vector<int> rank(si, sf), indices(sf - si);
    ;
    std::iota(indices.begin( ), indices.end( ), 0);
    for (int t = 1; t <= sf - si; t *= 2) {
        auto pred = [&, rank](int i1, int i2) {
            return std::make_pair(rank[i1], (i1 + t <
            sf - si ? rank[i1 + t] : -1)) < std:::
            make_pair(rank[i2], (i2 + t < sf - si ? rank[
            i2 + t] : -1));
        };
        std::sort(indices.begin( ), indices.end( ),
        std::cref(pred));
        for (int i = 0, r = 0; i < indices.size( );
        ++i) {
            rank[indices[i]] = r;
            r += (i + 1 != indices.size( ) && pred(
            indices[i], indices[i + 1]));
        }
    }
    return rank;
}

```

```

template<typename RI>
std::vector<RI> suffix_array(RI si, RI sf, const
std::vector<int>& rank) {
    std::vector<RI> res(sf - si);
    for (int i = 0; i < rank.size( ); ++i) {
        res[rank[i]] = si + i;
    }
    return res;
}

// función extra: ¿cuál es el prefijo más grande
// entre dos sufijos consecutivos?
template<typename RI>
std::vector<int> longest_prefix(RI si, RI sf,
const std::vector<int>& rank, const std:::
vector<RI>& suffix) {
    std::vector<int> res(sf - si);
    for (int i = 0, t = 0; i < rank.size( ); ++i) {
        if (rank[i] + 1 != sf - si) {
            t += std::mismatch(si + i + t, sf, suffix
            [rank[i] + 1] + t, sf).first - (si + i + t);
            res[rank[i]] = t;
            t -= (t > 0);
        } else {
            t = 0;
        }
    }
    return res;
}

template<typename RI1, typename RI2>
auto substring_search(RI1 si, RI1 sf, const std:::
vector<RI1>& suffix, const std:::vector<int>&
lcp, RI2 bi, RI2 bf) {
    auto xi = suffix.begin( ), xf = suffix.end( );
    auto li = lcp.begin( ), lf = lcp.end( );
    for (int i = 0; i < bf - bi; ++i) {
        while (xi != xf && (*xi)[i] != bi[i] && *li
        >= i) {
            ++xi, ++li;
        }
        if (xi == xf || (*xi)[i] != bi[i]) {
            return std::make_pair(xi, xi);
        }
    }

    xf = xi + 1;
    while (xf != suffix.end( ) && *li >= bf - bi) {
        ++xf, ++li;
    }
    return std::make_pair(xi, xf);
}

int main( ) {
    std::string s;
    std::cin >> s;

    auto rank = suffix_ranking(s.begin(), s.end());
    auto suffix = suffix_array(s.begin(), s.end(),
    rank);
}

```

```

std::string b;
std::cin >> b;

auto res = substring_search(s.begin(), s.end(),
    suffix, lcp, b.begin(), b.end());
std::cout << res.second - res.first;
// iteradores sobre suffix que denotan todas
// las cadenas donde b es prefijo, si res.second
// == res.first, la búsqueda fracasó
}

```

17 Búsqueda de subcadenas en tiempo lineal con Knuth-Morris-Pratt

```

std::vector<size_t> preprocesa(const std::string&
    s) {
    std::vector<size_t> b = { size_t(-1) };
    for (size_t i = 0, j = -1; i < s.size(); ++i)
    {
        while (j != -1 && s[i] != s[j]) {
            j = b[j];
        }
        b.push_back(++j);
    }
    return b;
}

```

```

std::vector<size_t> busca(const std::string& s,
    const std::string& t, const std::vector<size_t>
    &b) {
    std::vector<size_t> res;
    for (size_t i = 0, j = 0; i < t.size(); ++i) {
        while (j != -1 && t[i] != s[j]) {
            j = b[j];
        }
        if (++j == s.size()) {
            res.push_back(i + 1 - s.size());
            // si basta encontrar una coincidencia,
            // hacer break
            j = b[j];
        }
    }
    return res;
}

```

```

int main() {
    std::string t = "
abbabaabababaaabaaababbbaaaababbabaaaba";
    std::string s = "bbaaaababba";

    auto pre = preprocesa(s);
    auto res = busca(s, t, pre);
    std::cout << res.size() << "\n";
    // posiciones donde aparece la subcadena
    for (int pos : res) {
        std::cout << pos << " ";
    }
}

```

18 Subsecuencia creciente más larga

```

template<typename FI> // sólo tamaño
std::size_t longest_increasing_subsequence_size(FI
    ini, FI fin) {
    std::vector<typename std::iterator_traits<FI>::
        value_type> valores;
    for (auto i = ini; i != fin; ++i) {
        auto cambiar = std::upper_bound(valores.
            begin(), valores.end(), *i); //
        // upper_bound para creciente no estricta,
        // lower_bound para creciente estricta
        if (cambiar == valores.end()) {
            valores.push_back(*i);
        } else {
            *cambiar = *i;
        }
    }
    return valores.size();
}

```

```

template<typename BI> // elementos de la
    subsecuencia
std::vector<BI> longest_increasing_subsequence(BI
    ini, BI fin) {
    std::vector<BI> posiciones(1), atras;
    auto pred = [&](BI i, BI j) {
        return *i < *j;
    };

    for (auto i = ini; i != fin; ++i) {
        auto cambiar = std::upper_bound(posiciones.
            begin() + 1, posiciones.end(), i, pred); //
        // lower_bound para creciente estricta
        if (cambiar == posiciones.end()) {
            atras.push_back(posiciones.back());
            posiciones.push_back(i);
        } else if (pred(i, *cambiar)) {
            //
            // tautología con upper_bound (creciente no
            // estricta) pero no con lower_bound (creciente
            // estricta)
            atras.push_back(*(cambiar - 1));
            *cambiar = i;
        } else {
            atras.emplace_back();
        }
    }

    for (auto i = posiciones.end(); i !=
        posiciones.begin() + 1; --i) {
        *(i - 2) = atras[*i - 1 - ini];
    }

    return std::vector<BI>(posiciones.begin() + 1,
        posiciones.end());
}

```

```

int main() {
    std::string s;
    std::cin >> s;
}

```

```

auto tam = longest_increasing_subsequence_size(
    s.begin(), s.end());
auto res = longest_increasing_subsequence(s.
    begin(), s.end());

```

19 Código Lehmer ($i \Leftrightarrow i$ -ésima permutación)

```

constexpr std::size_t factorial(std::size_t n) {
    return (n == 0 ? 1 : n * factorial(n - 1));
}

template<typename T> // inicio y fin del arreglo
    para guardar la permutación del índice dado (
    permutación de { 0, 1, etc, N-1 } donde N es
    el tamaño es el tamaño del arreglo
void permutacion(T ini, T fin, std::size_t indice)
{
    std::iota(ini, fin, std::size_t(0));
    std::size_t n = fin - ini, r = factorial(n - 1)
    ;
    for (T i = ini; i != fin; ++i) {
        std::size_t d = indice / r; indice %= r;
        std::rotate(i, i + d, i + d + 1);
        if (fin - i - 1 != 0) {
            r /= fin - i - 1;
        }
    }
}

template<typename T> // inicio y fin del arreglo
    con la permutación, regresa el índice
std::size_t indice(T ini, T fin) {
    std::size_t n = fin - ini, r = factorial(n - 1)
    , res = 0;
    for (T i = ini; i != fin; ++i) {
        res += r * std::count_if(i + 1, fin, [&](std
        ::size_t v) { return v < *i; });
        if (fin - i - 1 != 0) {
            r /= fin - i - 1;
        }
    }
    return res;
}

```

20 Generación de cadenas binarias usando recursión

```

int n;
bool arr[MAX];

// llamada inicial: cadenas_binarias(0)
void cadenas_binarias(int i) {
    if (i == n) {
        for (int i = 0; i < n; ++i) {
            std::cout << arr[i];
        }
    }
}

```

```

        std::cout << "\n";
    } else {
        arr[i] = false;
        cadenas_binarias(i + 1);
        arr[i] = true;
        cadenas_binarias(i + 1);
    }
}

```

21 Generación de cadenas numéricas usando recursión

```

int n;
int arr[MAX];

// llamada inicial: cadenas_numericas(0)
void cadenas_numericas(int i) {
    if (i == n) {
        for (int i = 0; i < n; ++i) {
            std::cout << arr[i] << " ";
        }
        std::cout << "\n";
    } else {
        for (int d = 0; d <= 9; ++d) {
            arr[i] = d;
            cadenas_numericas(i + 1);
        }
    }
}

```

22 Generación de permutaciones usando recursión

```

int n;
int arr[MAX]; // inicializar con { 0, 1, ..., n - 1 }

// llamada inicial: permutaciones(0)
void permutaciones(int i) {
    if (i == n) {
        for (int i = 0; i < n; ++i) {
            std::cout << arr[i] << " ";
        }
        std::cout << "\n";
    } else {
        for (int j = i; j < n; ++j) {
            std::swap(arr[i], arr[j]);
            permutaciones(i + 1);
            std::swap(arr[i], arr[j]);
        }
    }
}

```

23 Generación de permutaciones usando la biblioteca

```

int n;
int arr[MAX]; // inicializar con { 0, 1, ..., n - 1 }
do {
    // procesar
} while (std::next_permutation(&arr[0], &arr[n]));

```

24 Fibonacci con programación dinámica

```

int fibonacci_pd(int n) {
    int mem[std::max(2, n + 1)];
    for (int i = 0; i <= n; ++i) {
        if (i <= 1) {
            mem[i] = i;
        } else {
            mem[i] = mem[i - 1] + mem[i - 2];
        }
    }
    return mem[n];
}

```

25 Subsecuencia común más larga con programación dinámica

```

auto lcs(const std::string& a, const std::string& b) {
    int mem[a.size() + 1][b.size() + 1];
    for (int i = a.size(); i >= 0; --i) {
        for (int j = b.size(); j >= 0; --j) {
            if (i == a.size() || j == b.size()) {
                mem[i][j] = 0;
            } else if (a[i] == b[j]) {
                mem[i][j] = 1 + mem[i + 1][j + 1];
            } else {
                mem[i][j] = std::max(mem[i][j + 1], mem[i + 1][j]);
            }
        }
    }

    std::vector<std::pair<int, int>> res;
    int i = 0, j = 0;
    while (i < a.size() && j < b.size()) {
        if (a[i] == b[j]) {
            res.emplace_back(i++, j++);
        } else if (mem[i][j] == mem[i][j + 1]) {
            ++j;
        } else {
            ++i;
        }
    }
    return res;
}

```

26 Subsecuencia común más larga con memoria lineal

```

int lcs(const std::string& a, const std::string& b) {
    int mem[2][b.size() + 1];
    int *actual = mem[0], *previo = mem[1];
    for (int i = a.size(); i >= 0; --i, std::swap(actual, previo)) {
        for (int j = b.size(); j >= 0; --j) {
            if (i == a.size() || j == b.size()) {
                actual[j] = 0;
            } else if (a[i] == b[j]) {
                actual[j] = 1 + mem[1][j + 1];
            } else {
                actual[j] = std::max(actual[j + 1], mem[1][j]);
            }
        }
    }
    return previo[0];
}

```

27 Trie

```

class trie {
public:
    bool inserta(const std::string& s) {
        auto actual = this;
        for (int i = 0; i < s.size(); ++i) {
            auto& siguiente = actual->nivel_[s[i]];
            if (siguiente == nullptr) {
                siguiente = new trie;
            }
            actual = siguiente;
        }
        return actual->nivel_.emplace('\0', nullptr).second;
    }

    trie* posicion(const std::string& s) {
        auto actual = this;
        for (int i = 0; i < s.size(); ++i) {
            auto iter = actual->nivel_.find(s[i]);
            if (iter == actual->nivel_.end()) {
                return nullptr;
            }
            actual = iter->second;
        }
        return actual;
    }

    bool busca(const std::string& s) {
        auto pos = posicion(s);
        return pos != nullptr && pos->nivel_.find('\0') != pos->nivel_.end();
    }

    bool prefijo(const std::string& s) {
        auto pos = posicion(s);
        return pos != nullptr;
    }
}

```

```
private:
    std::map<char, trie*> nivel_;
};
```

28 Árbol de Fenwick

```
template<typename T>
class fenwick_tree {
public:
    fenwick_tree(int n)
    : mem_(n + 1) {
    }

    int size( ) const {
        return mem_.size( ) - 1;
    }

    T operator[](int i) const {
        return query(i, i + 1);
    }

    T query(int i, int f) const {
        return query_until(f) - query_until(i);
    }

    T query_until(int f) const {
        T res = 0;
        for (; f != 0; f -= (f & -f)) {
            res += mem_[f];
        }
        return res;
    }

    int min_prefix(T v) const { // calcula la
        // cantidad mínima de elementos (comenzando por
        // la izquierda) que se necesitan
        int i = 0; // para lograr
        // un acumulado >= v; si es imposible lograr
        // dicha suma, regresa .size( ) + 1
        for (int j = std::bit_floor(mem_.size( )); j
            > 0; j /= 2) {
            if (i + j < mem_.size( ) && mem_[i + j]
                <= v) {
                v -= mem_[i + j];
                i += j;
            }
        }
        return i + (v > 0);
    }

    void replace(int i, const T& v) {
        modify_add(i, v - operator[](i));
    }

    void modify_add(int i, const T& d) {
        for (i += 1; i < mem_.size( ); i += (i & -i)) {
            mem_[i] += d;
        }
    }
};
```

```
private:
    std::vector<T> mem_;
};
```

29 Árbol de Fenwick multidimensional

```
template<typename T, int D>
class fenwick_tree {
public:
    template<typename... P>
    fenwick_tree(int n, const P&... s)
    : mem_(n + 1, fenwick_tree<T, D - 1>(s...)) {
    }

    template<typename... P>
    void modify_add(int i, const P&... s) {
        for (i += 1; i < mem_.size( ); i += (i & -i)) {
            mem_[i].modify_add(s...);
        }
    }

    template<typename... P>
    T prefix_until(int f, const P&... s) const {
        T res = 0;
        for (; f != 0; f -= (f & -f)) {
            res += mem_[f].prefix_until(s...);
        }
        return res;
    }

private:
    std::vector<fenwick_tree<T, D - 1>> mem_;
};

template<typename T>
class fenwick_tree<T, 0> {
public:
    void modify_add(const T& d) {
        v_ += d;
    }

    T prefix_until( ) const {
        return v_;
    }

private:
    T v_ = 0;
};
```

30 Árbol de segmentos

```
template<typename T, typename F = const T&(*) (
    const T&, const T&)>
class segment_tree {
public:
    segment_tree(T neutro, F f)
```



```

: pisos_(1), neutro_(std::move(neutro)),
funcion_(std::move(f)) {
}

int size( ) const {
    return pisos_[0].size( );
}

const T& operator[](int i) const {
    return pisos_[0][i];
}

void push_back(T v) {
    for (int p = 0;; ++p, pisos_.resize(std::max
(p + 1, int(pisos_.size( )))) {
        pisos_[p].push_back(std::move(v));
        if (pisos_[p].size( ) % 2 == 1) {
            break;
        }
        v = funcion_*(pisos_[p].end( ) - 2), *(
pisos_[p].end( ) - 1));
    }
}

void pop_back( ) {
    for (int p = 0;; ++p) {
        pisos_[p].pop_back( );
        if (pisos_[p].size( ) % 2 == 0) {
            break;
        }
    }
}

void replace(int i, T v) {
    for (int p = 0;; ++p, i /= 2) {
        pisos_[p][i] = std::move(v);
        if (i + (i % 2 == 0) == pisos_[p].size( )
) {
            break;
        }
        v = funcion_(pisos_[p][i - i % 2], pisos_
[p][i - i % 2 + 1]);
    }
}

T query(int ini, int fin) const {
    T res = neutro_;
    visit(ini, fin, [&](const T& actual) {
        res = funcion_(res, actual);
    });
    return res;
}

template<typename V>
void visit(int ini, int fin, V&& vis) const {
    for (int p = 0; ini != fin; ++p, ini /= 2,
fin /= 2) {
        if (ini % 2 == 1) {
            vis(pisos_[p][ini++]);
        }
        if (fin % 2 == 1) {
                vis(pisos_[p][--fin]);
            }
        }
    }

private:
    std::vector<std::vector<T>> pisos_;
    F funcion_;
    T neutro_;
};

// < C++17
/*template<typename T, typename F>
segment_tree<T, F> make_segment_tree(const T& v, F
f) {
    return segment_tree<T, F>(v, f);
}*/

int main( ) {
    // C++17
    // auto s = make_segment_tree(0, std::plus<int
>( ));
    auto s = segment_tree(0, std::plus( ));
    for (int i = 0; i < 50; ++i) {
        s.push_back(i);
    }
    std::cout << s.query(5, 10) << "\n";

    s.visit(5, 10, [&](int actual) {
        std::cout << actual << " ";
    });
}

31 Cerco convexo

template<typename T>
T producto_cruz(const std::pair<T, T>& a, const
std::pair<T, T>& b, const std::pair<T, T>& c)
{
    return (b.first - a.first) * (c.second - a.
second) - (b.second - a.second) * (c.first - a
.first);
}

template<typename RI1, typename RI2>
RI2 cerco_parcial(RI1 ai, RI1 af, RI2 bi) {
    auto bw = bi;
    for (; ai != af; *bw++ = *ai++) {
        while (bw - bi >= 2 && producto_cruz(*(bw -
2), *(bw - 1), *ai) <= 0) {
            --bw;
        }
    }

    return bw;
}

template<typename RI>
auto cerco_convexo(RI ai, RI af) {

```

```

std::vector<typename std::iterator_traits<RI>::
value_type> res(af - ai + 2);
auto iter1 = cerco_parcial(ai, af, res.begin( )
) - 1;
auto iter2 = cerco_parcial(std::
make_reverse_iterator(af), std::
make_reverse_iterator(ai), iter1) - 1;
res.resize(iter2 - res.begin( ));

return res;
}

template<typename RI>
auto area_convexo(RI ai, RI af) {
    typename std::iterator_traits<RI>::value_type
    res(0);
    for (std::size_t i = 0; i < af - ai; ++i) {
        res += ai[i].first * ai[(i + 1) % (af - ai)]
        .second - ai[i].second * ai[(i + 1) % (af -
        ai)].first;
    }
    return res / 2;
}

```

32 Acoplamiento bipartito de cardinalidad máxima

```

namespace bipartito {
    bool aumenta(int a, std::vector<int>
    adyacencia_a[], std::vector<int>& nivel_a, std
    ::vector<int>& pareja_a, std::vector<int>&
    pareja_b, bool visto_a[]) {
        visto_a[a] = true;
        for (int b : adyacencia_a[a]) {
            if (pareja_b[b] == -1 || !visto_a[
            pareja_b[b]] && nivel_a[a] < nivel_a[pareja_b[
            b]] && aumenta(pareja_b[b], adyacencia_a,
            nivel_a, pareja_a, pareja_b, visto_a)) {
                pareja_a[a] = b;
                pareja_b[b] = a;
                return true;
            }
        }
        return false;
    }

    auto calcula(std::vector<int> adyacencia_a[],
    int n, int m) {
        std::vector<int> nivel_a(n), pareja_a(n, -1)
        , pareja_b(m, -1);
        for (;;) {
            std::queue<int> cola;
            for (int a = 0; a < n; ++a) {
                if (pareja_a[a] == -1) {
                    nivel_a[a] = 0, cola.push(a);
                } else {
                    nivel_a[a] = -1;
                }
            }
        }
    }
}

```

```

        for (; !cola.empty( ); cola.pop( )) {
            int a = cola.front( );
            for (int b : adyacencia_a[a]) {
                if (pareja_b[b] != -1 && nivel_a[
                pareja_b[b]] < 0) {
                    nivel_a[pareja_b[b]] = nivel_a[a
                ] + 1;
                    cola.push(pareja_b[b]);
                }
            }
        }

        bool mejora = false, visto_a[n] = { };
        for (int a = 0; a < n; ++a) {
            mejora |= (pareja_a[a] == -1 &&
            aumenta(a, adyacencia_a, nivel_a, pareja_a,
            pareja_b, visto_a));
        }
        if (!mejora) {
            int cardinalidad = n - std::count(
            pareja_a.begin( ), pareja_a.end( ), -1);
            return std::tuple(std::move(pareja_a),
            std::move(pareja_b), cardinalidad);
        }
    }
};

```

```

int main( ) {
    // gráfica bipartita donde |A| = tam_a, |B| =
    tam_b; pensar que las aristas van de A a B
    // Los identificadores de vértices van de 0 ->
    tam_a - 1 en A y 0 -> tam_b - 1 en B.
    // auto [pa, pb, tam] = bipartito(
    lista_adyacencia_a, tam_a, tam_b);
    // pa es un arreglo de enteros que indica quién
    es la pareja de cada vértice de a
    // pb es un arreglo de enteros que indica quién
    es la pareja de cada vértice de b
}

```

33 Acoplamiento bipartito de cardinalidad máxima con costo mínimo

```

// cost[i][j] = cost for pairing left node i
with right node j
// Lmate[i] = index of right node that left node
i pairs with
// Rmate[j] = index of left node that right node
j pairs with
//
// The values in cost[i][j] may be positive or
negative. To perform
// maximization, simply negate the cost[][] matrix

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>
#include <iostream>

```

```

using namespace std;

typedef vector<double> VD; // el algoritmo usa
double para los costos durante el cálculo
typedef vector<VD> VVD; // std::vector<std::
vector<double>> como matriz de adyacencia para
los costos
typedef vector<int> VI; // std::vector<int>
para guardar el índice de la pareja (pasarlo
vacío, el algoritmo lo llena)

double MinCostMatching(const VVD &cost, VI &Lmate,
VI &Rmate) {
int n = int(cost.size());

// construct dual feasible solution
VD u(n);
VD v(n);
for (int i = 0; i < n; i++) {
u[i] = cost[i][0];
for (int j = 1; j < n; j++) u[i] = min(u[i],
cost[i][j]);
}
for (int j = 0; j < n; j++) {
v[j] = cost[0][j] - u[0];
for (int i = 1; i < n; i++) v[j] = min(v[j],
cost[i][j] - u[i]);
}

// construct primal solution satisfying
complementary slackness
Lmate = VI(n, -1);
Rmate = VI(n, -1);
int mated = 0;
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (Rmate[j] != -1) continue;
if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10)
{
Lmate[i] = j;
Rmate[j] = i;
mated++;
break;
}
}
}

VD dist(n);
VI dad(n);
VI seen(n);

// repeat until primal solution is feasible
while (mated < n) {

// find an unmatched left node
int s = 0;
while (Lmate[s] != -1) s++;

// initialize Dijkstra
fill(dad.begin(), dad.end(), -1);

fill(seen.begin(), seen.end(), 0);
for (int k = 0; k < n; k++)
dist[k] = cost[s][k] - u[s] - v[k];

int j = 0;
while (true) {

// find closest
j = -1;
for (int k = 0; k < n; k++) {
if (seen[k]) continue;
if (j == -1 || dist[k] < dist[j]) j = k;
}
seen[j] = 1;

// termination condition
if (Rmate[j] == -1) break;

// relax neighbors
const int i = Rmate[j];
for (int k = 0; k < n; k++) {
if (seen[k]) continue;
const double new_dist = dist[j] + cost[i][k] -
u[i] - v[k];
if (dist[k] > new_dist) {
dist[k] = new_dist;
dad[k] = j;
}
}
}

// update dual variables
for (int k = 0; k < n; k++) {
if (k == j || !seen[k]) continue;
const int i = Rmate[k];
v[k] += dist[k] - dist[j];
u[i] -= dist[k] - dist[j];
}
u[s] += dist[j];

// augment along path
while (dad[j] >= 0) {
const int d = dad[j];
Rmate[j] = Rmate[d];
Lmate[Rmate[j]] = j;
j = d;
}
Rmate[j] = s;
Lmate[s] = j;

mated++;
}

double value = 0;
for (int i = 0; i < n; i++)
value += cost[i][Lmate[i]];

return value;
}

```

34 Flujo máximo

```
// Running time:
//       $O(|V|^2 |E|)$ 
//
// INPUT:
//      - graph, constructed using AddEdge()
//      - source
//      - sink
//
// OUTPUT:
//      - maximum flow value
//      - To obtain the actual flow values, look
//        at all edges with
//        capacity > 0 (zero capacity edges are
//        residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

struct Edge {
    int u, v, cap, flow;

    Edge() {}
    Edge(int u, int v, int cap): u(u), v(v), cap(
        cap), flow(0) {}
};

struct Dinic {
    int N;
    vector<Edge> E;
    vector<vector<int>> g;
    vector<int> d, pt;
    Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

    void AddEdge(int u, int v, int cap) {
        if (u != v) {
            E.emplace_back(Edge(u, v, cap));
            g[u].emplace_back(E.size() - 1);
            E.emplace_back(Edge(v, u, 0));
            g[v].emplace_back(E.size() - 1);
        }
    }

    bool BFS(int S, int T) {
        queue<int> q({S});
        fill(d.begin(), d.end(), N + 1);
        d[S] = 0;
        while(!q.empty()) {
            int u = q.front(); q.pop();
            if (u == T) break;
            for (int k: g[u]) {
                Edge &e = E[k];
                if (e.flow < e.cap && d[e.v] > d[e.u]
+ 1) {
                    d[e.v] = d[e.u] + 1;
                    q.emplace(e.v);
                }
            }
        }
    }
};
```

```
    }
    return d[T] != N + 1;
}

int DFS(int u, int T, int flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
        Edge &e = E[g[u][i]];
        Edge &oe = E[g[u][i]^1];
        if (d[e.v] == d[e.u] + 1) {
            int amt = e.cap - e.flow;
            if (flow != -1 && amt > flow) amt =
flow;
            if (int pushed = DFS(e.v, T, amt)) {
                e.flow += pushed;
                oe.flow -= pushed;
                return pushed;
            }
        }
    }
    return 0;
}

int MaxFlow(int S, int T) {
    int total = 0;
    while (BFS(S, T)) {
        fill(pt.begin(), pt.end(), 0);
        while (int flow = DFS(S, T))
            total += flow;
    }
    return total;
}

int main( ) {
    int n, e;
    std::cin >> n >> e;
    Dinic dinic(n);

    for(int i = 0; i < e; i++) {
        int u, v, cap;
        std::cin >> u >> v >> cap;
        dinic.AddEdge(u, v, cap);
        dinic.AddEdge(v, u, cap);
    }
    std::cout << dinic.MaxFlow(0, n - 1);
}
```

35 Fibonacci matricial en tiempo logarítmico

```
template<typename T>
T potencia(T a, int b) {
    if (b == 0) {
        return T(1);
    } else {
        auto res = potencia(a, b / 2);
        res *= res;
        if (b % 2 == 1) {
            res *= a;
        }
    }
}
```

```

        return res;
    }
}

struct matriz {
    std::array<std::array<int, 2>, 2> mat;

    explicit matriz( ) = default;
    explicit matriz(int v) {
        if (v == 0) {
            mat = {{ { 0, 0 }, { 0, 0 } }};
        } else if (v == 1) {
            mat = {{ { 1, 0 }, { 0, 1 } }};
        }
    }
    explicit matriz(std::array<std::array<int, 2>,
        2> m)
        : mat(m) {
    }

    void operator*=(const matriz& m) {
        std::array<std::array<int, 2>, 2> temp;
        for (int i = 0; i < 2; ++i) {
            for (int j = 0; j < 2; ++j) {
                long long res = 0;
                for (int k = 0; k < 2; ++k) {
                    res += ((long long)mat[i][k] * m.
mat[k][j]) % 1000000007;
                }
                temp[i][j] = res % 1000000007;
            }
        }
        mat = temp;
    }
};

int fibonacci(int n) {
    const auto& elevada = potencia(matriz({{ { 1, 1
    }, { 1, 0 } }}), n);
    return elevada.mat[1][0];
}

```

36 Algoritmo de Kruskal (unión-pertenencia)

```

struct arista {
    int x, y, costo;
};

int jefe(int tabla[], int x) {
    if (tabla[x] != x) {
        tabla[x] = jefe(tabla, tabla[x]);
    }
    return tabla[x];
}

```

```

bool mismo(int tabla[], int x, int y) {
    return jefe(tabla, x) == jefe(tabla, y);
}

void une(int tabla[], int x, int y) {
    tabla[jefe(tabla, x)] = jefe(tabla, y);
}

int main( ) {
    int v, a;
    std::cin >> v >> a;

    std::vector<arista> aristas;
    for (int i = 0; i < a; ++i) {
        int x, y, costo;
        std::cin >> x >> y;
        aristas.push_back({ x, y, costo });
    }
    std::sort(aristas.begin( ), aristas.end( ), [](
        arista a, arista b) {
        return a.costo < b.costo;
    });

    int tabla[v];
    std::iota(&tabla[0], &tabla[0] + v, 0);
    for (int i = 0; i < aristas.size( ); ++i) {
        if (!mismo(tabla, aristas[i].x, aristas[i].y
        )) {
            une(tabla, aristas[i].x, aristas[i].y);
        }
    }
}

```