# Algoritmos básicos.

Aquí podrás encontrar implementaciones eficientes y correctas (espero!) de varios algoritmos básicos. Al final de cada implementación viene una función "main", que sólo está para mostrar un poco cómo usar las clases/funciones.

Al copiar, no tienes que copiar los comentarios, y a veces hay funciones extra que claramente no necesitarás. Por ejemplo, en Graph hay varias versiones de ´add_edge´, pero lo más probable es que utilices sólo una de ellas.

Varios de ellos dependen de otros. Ahí mismo dice de quién dependen. Por ejemplo, min spanning tree depende de disjoint sets y de graph.

Todo el código lo hice yo, EXCEPTO el simplex y Max Flow, que obtuve de aquí: https://github.com/jaehyunp/stanfordacm/blob/master/code

## Índice

# Teoría de Números

Tenemos las siguientes funciones:

- `reduce_mod(a,mod)` reduce a a su residuo **positivo** de dividir a entre mod.
- `modulo(a,mod)` regresa reduce_mod(a,mod)
- `pow(a,n)` y `pow_mod(a,n,mod)` regresan a^n y a^n%mod respectivamente.
- `gcd_extended(a,b)` regresa el máximo común divisor d = gcd(a,b) y también la combinación lineal ax+by=d.
- `mod_inverse(a,n)` regresa el inverso modular de a módulo n. Ejemplo: $4 \cdot 3 \equiv 1 \pmod{11}$, así que 4 y 3 son inversos módulo 11.

Además, hay funciones para convertir enteros de una base a otra.

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

using ll = long long;

template <class T = ll, class U = ll>
void reduce_mod(T& a, const U mod)
{
    a %= mod;
    if (a < 0)
        a += mod;
}

template <class T = ll, class U = ll>
T modulo(T a, const U mod)
{
    reduce_mod(a, mod);
    return a;
}

// calculates a^n efficiently. Mostly like std::pow.
// Can do it for any class that has operator*defined!
template <class T = ll, T identity = 1>
T pow(T a, unsigned long n)
{
    T r = identity;

    while (n > 0)
```

```
    {
        if (n%2 == 1)
            r *= a;

        n /= 2;
        a *= a;
    }

    return r;
}

// a^n (mod mod)
ll pow_mod(ll a, unsigned long n, const ll mod)
{
    ll r = 1;
    reduce_mod(a, mod);
    while (n > 0)
    {
        if (n%2 == 1)
        {
            r *= a;
            reduce_mod(r, mod);
        }

        n /= 2;
        a *= a;
        reduce_mod(a, mod);
    }

    return r;
}

ll gcd(ll a, ll b)
{
    while (b != 0)
    {
        ll r = a%b;
        a = b;
        b = r;
    }
    return a;
}

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

```
struct linearcomb
{
    ll d; // gcd
    ll x; // first coefficient
    ll y; // second coefficient
};

// pseudocode taken from wikipedia
linearcomb gcd_extended(ll a, ll b)
{
    if (b == 0)
        return {a, 1LL, 0LL};

    ll sa = 1, sb = 0, sc, ta = 0, tb = 1, tc;

    do
    {
        auto K = std::div(a, b);

        a = b;
        b = K.rem;

        sc = sa - K.quot*sb;
        sa = sb;
        sb = sc;

        tc = ta - K.quot*tb;
        ta = tb;
        tb = tc;
    } while (b != 0);

    return {a, sa, ta};
}

ll mod_inverse(ll a, const ll n)
{
    ll x = gcd_extended(a, n).x;
    reduce_mod(x, n);
    return x;
}

// digits[i] = coefficient of b^i
template <class IntType>
ll ReadNumberInBaseB(ll b, const std::vector<IntType>& digits)
{
```

```
    ll suma = 0;
    ll power = 1;

    for (ll d : digits)
    {
        suma += power*d;
        power *= b;
    }

    return suma;
}

// Does NOT reverse the digits. add std::reverse at end if desired.
std::vector<int> WriteNumberInBaseB(ll n, int b)
{
    std::vector<int> digits;

    while (n)
    {
        digits.push_back(n%b);
        n /= b;
    }

    return digits;
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    cout << "modulo(-37,10) = " << modulo(-37, 10) << endl;
    cout << "7^1000 (mod 5) = " << pow_mod(7, 1000, 5) << endl;

    auto dxy = gcd_extended(30, 55);
    cout << "\ngcd(30,55) = " << dxy.d << " = 30*" << dxy.x << " + 55*" << dxy.y
         << endl;
    cout << "lcm(30,55) = " << lcm(30, 55) << endl;
```

```
    cout << "\n1/7 (mod 9) = " << mod_inverse(7, 9) << endl;

    std::vector<int> V = {1, 2, 0, 4};
    cout << "\n4021_{5} = " << ReadNumberInBaseB(5, V) << "_{10}" << endl;
    cout << "10 in base 2: " << WriteNumberInBaseB(10, 2) << endl;
    cout << "100 in base 7: " << WriteNumberInBaseB(100, 7) << endl;

    return 0;
}
```

**Output**:

```
modulo(-37,10) = 3
7^1000 (mod 5) = 1

gcd(30,55) = 5 = 30*2 + 55*-1
lcm(30,55) = 330

1/7 (mod 9) = 4

4021_{5} = 511_{10}
10 in base 2: 0 1 0 1
100 in base 7: 2 0 2
```

# Números primos y factorizaciones en primos

Funciones para encontrar la lista de los primeros $k$ primos y para factorizar números. Incluye la función $\phi$ de Euler, definida como sigue: $\phi(n) :=$ cantidad de primos relativos con $n$ menores o iguales a $n$.

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

using ll = long long;

// Represents p^a
struct prime_to_power
{
    prime_to_power(ll prime, ll power) : p(prime), a(power) {}
    ll p;
    ll a;

    explicit operator ll() const { return std::pow(p, a); }
};

class Factorization
{
public:
    using value_type = prime_to_power;
    explicit operator ll() const
    {
        ll t = 1;

        for (auto& pa : m_prime_factors)
            t *= ll(pa);

        return t;
    }

    // returns the power of prime p
    ll operator[](ll p) const
    {
        auto& PF = m_prime_factors;
        auto it = std::partition_point(
          PF.begin(), PF.end(), [p](const prime_to_power& p_a) {
```

```
            return p_a.p < p;
        });

        if (it == PF.end() || it->p != p)
            return 0;

        return it->a;
    }

    ll& operator[](ll p)
    {
        auto& PF = m_prime_factors;
        auto it = std::partition_point(
          PF.begin(), PF.end(), [p](const prime_to_power& p_a) {
              return p_a.p < p;
          });

        if (it == PF.end())
        {
            PF.emplace_back(p, 0);
            return PF.back().a;
        }

        // if it exists, everything is fine
        if (it->p == p)
            return it->a;

        // Has to insert into correct position
        ++it;
        it = PF.insert(it, prime_to_power(p, 0));
        return it->a;
    }

    void emplace_back(ll p, ll a)
    {
        assert(m_prime_factors.empty() || p > m_prime_factors.back().p);
        m_prime_factors.emplace_back(p, a);
    }

    auto begin() const { return m_prime_factors.begin(); }
    auto end() const { return m_prime_factors.end(); }

    ll size() const { return m_prime_factors.size(); }

private:
```

```
    std::vector<prime_to_power> m_prime_factors;
};

std::ostream& operator<<(std::ostream& os, const Factorization& F)
{
    ll i = 0;

    for (auto f : F)
    {
        os << f.p;

        if (f.a != 1)
            os << "^" << f.a;

        if (i + 1 != F.size())
            os << "*";

        ++i;
    }

    return os;
}

// returns the biggest integer t such that t*t <= n
ll integral_sqrt(ll n)
{
    ll t = std::round(std::sqrt(n));
    if (t*t > n)
        return t - 1;

    return t;
}

bool is_square(ll N)
{
    ll t = std::round(std::sqrt(N));
    return t*t == N;
}

ll FermatFactor(ll N)
{
    assert(N%2 == 1);
    ll a = std::ceil(std::sqrt(N));
    ll b2 = a*a - N;
```

```
    while (!is_square(b2))
    {
        ++a;
        b2 = a*a - N;
    }

    return a - integral_sqrt(b2);
}

class PrimeFactorizer
{
public:
    explicit PrimeFactorizer(ll primes_up_to = 1000000) : m_upto(primes_up_to)
    {
        eratosthenes_sieve(m_upto);
    }

    // Number of primes
    ll size() const { return primes.size(); }

    // Calculated all primes up to
    ll up_to() const { return m_upto; }

    bool is_prime(ll p) const
    {
        if (p <= m_upto)
            return std::binary_search(primes.begin(), primes.end(), p);

        ll largest = primes.back();
        if (p <= largest*largest)
            return bf_is_prime(p);

        ll a = FermatFactor(p);
        return a == 1;
    }

    auto begin() const { return primes.begin(); }
    auto end() const { return primes.end(); }
    ll operator[](ll index) const { return primes[index]; }

    const auto& Primes() const { return primes; }

    /// Make sure sqrt(n) <
    /// primes.back()^2, otherwise
    /// this could spit out a wrong factorization.
```

```cpp
    Factorization prime_factorization(ll n) const
    {
        Factorization F;

        if (n <= 1)
            return F;

        for (auto p : primes)
        {
            ll a = 0;
            auto qr = std::div(n, p);

            while (qr.rem == 0)
            {
                n = qr.quot;
                qr = std::div(n, p);
                ++a;
            }

            if (a != 0)
                F.emplace_back(p, a);

            if (p*p > n)
                break;
        }

        if (n > 1)
            ++F[n];

        return F;
    }

private:
    void eratosthenes_sieve(ll n)
    {
        // primecharfunc[a] == true means 2*a+1 is prime
        std::vector<bool> primecharfunc = {false};
        primecharfunc.resize(n/2 + 1, true);

        primes.reserve((1.1*n)/std::log(n) + 10); // can remove this line

        ll i = 1;
        ll p = 3; // p = 2*i + 1
        for (; p*p <= n; ++i, p += 2)
        {
```

```
            if (primecharfunc[i])
            {
                primes.emplace_back(p);
                for (ll j = i + p; j < primecharfunc.size(); j += p)
                    primecharfunc[j] = false;
            }
        }

        for (; p < n; p += 2, ++i)
        {
            if (primecharfunc[i])
                primes.emplace_back(p);
        }
    }

    // private because n has to be odd, and maybe
    // is already a factor in something.
    void fermat_factorization(ll n, Factorization& F)
    {
        assert(n%2 == 1);
        assert(n > 5);
        ll a = FermatFactor(n);

        ll b = n/a;

        assert(a*b == n);

        if (a == 1)
        {
            ++F[b];
        }
        else
        {
            fermat_factorization(a, F);
            fermat_factorization(b, F);
        }
    }

private:
    ll m_upto;
    std::vector<ll> primes = {2};

    bool bf_is_prime(ll n) const
    {
        for (auto p : primes)
```

```
        {
            if (p*p > n)
                break;

            if (n%p == 0)
                return false;
        }

        return true;
    }
}; // end class PrimeFactorizer

class EulerPhi
{
public:
    EulerPhi(const PrimeFactorizer& P) : m_phi(P.up_to())
    {
        m_phi[0] = 0;
        m_phi[1] = 1;
        dfs_helper(P, 1, 0);
    }

    // TODO(mraggi): only works if already calculated.
    ll operator()(ll k) const { return m_phi[k]; }

    ll operator[](ll k) const { return m_phi[k]; }

    ll size() const { return m_phi.size(); }

private:
    void dfs_helper(const PrimeFactorizer& P, ll a, ll i)
    {
        ll n = m_phi.size();
        for (; i < P.size() && P[i]*a < n; ++i)
        {
            ll p = P[i];
            ll multiplier = p - 1;
            if (a%p == 0)
                multiplier = p;
            m_phi[p*a] = multiplier*m_phi[a];
            dfs_helper(P, p*a, i);
        }
    }

    std::vector<ll> m_phi;
```

```
};

int main()
{
    PrimeFactorizer P;

    std::cout << "Primes: ";
    for (auto it = P.Primes().begin(); it != P.Primes().begin() + 100; ++it)
        std::cout << *it << ' ';
    std::cout << std::endl;

    for (ll n = 2; n <= 30; ++n)
    {
        std::cout << n << " = " << P.prime_factorization(n) << std::endl;
    }

    return 0;
}
```

**Output**:

```
Primes: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 1
2 = 2
3 = 3
4 = 2^2
5 = 5
6 = 2 * 3
7 = 7
8 = 2^3
9 = 3^2
10 = 2 * 5
11 = 11
12 = 2^2 * 3
13 = 13
14 = 2 * 7
15 = 3 * 5
16 = 2^4
17 = 17
18 = 2 * 3^2
19 = 19
20 = 2^2 * 5
21 = 3 * 7
22 = 2 * 11
23 = 23
24 = 2^3 * 3
25 = 5^2
```

```
26 = 2 * 13
27 = 3^3
28 = 2^2 * 7
29 = 29
30 = 2 * 3 * 5
```

# Longest Increasing Subsequence

Dada una lista, encuentra la subsecuencia creciente más larga. Puede configurarse qué significa “creciente”. Ver ejemplos.

- Tiempo de procesamiento: $O(n \log(n))$

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

// Pseudocode taken from wikipedia and tweaked for speed :)
template <class T, class Compare = std::less<T>>
auto longest_increasing_subsequence(const std::vector<T>& X,
                                    Compare comp = std::less<T>())
{
    long n = X.size();

    using PII = std::pair<int, T>;

    // M[k] = index i of smallest X[i] for which
    // there is a subsequence of length k ending
    // at X[i]. Note that M will be increasing.
    std::vector<PII> M(2);
    M.reserve((n + 2)/2);

    // P[i] = parent of i.
    std::vector<int> P(n);

    int L = 1;
    M[1].first = 0;
    M[1].second = X[0];
    for (long i = 1; i < n; ++i)
    {
        auto first = M.begin() + 1;
        auto last = M.begin() + L + 1;

        const auto& xi = X[i];

        auto newL = std::partition_point(first,
                                         last,
                                         [xi, &comp](const PII& p) {
```

```
                                                    return comp(p.second, xi);
                                                }) -
          first + 1;

        P[i] = M[newL - 1].first;

        if (newL < M.size())
        {
            M[newL].first = i;
            M[newL].second = xi;
        }
        else
        {
            M.push_back({i, xi});
        }

        if (newL > L)
            L = newL;
    }

    std::vector<T> S(L);
    long k = M[L].first;

    for (auto it = S.rbegin(); it != S.rend(); ++it, k = P[k])
    {
        *it = X[k];
    }

    return S;
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    std::vector<int> A = {0, 4, 2, 3, 5, 2, 1, 7, 3, 5, 4, 3,
                          4, 5, 6, 4, 5, 3, 1, 5, 2, 6, 9};
```

```
    cout << "A = " << A << endl;

    cout << "Longest increasing subsequence: "
         << longest_increasing_subsequence(A) << endl;

    cout << "Longest non-decreasing subsequence: "
         << longest_increasing_subsequence(A, std::less_equal<>()) << endl;

    cout << "Longest decreasing subsequence: "
         << longest_increasing_subsequence(A, std::greater<>()) << endl;

    return 0;
}
```

**Output**:

```
A = 0 4 2 3 5 2 1 7 3 5 4 3 4 5 6 4 5 3 1 5 2 6 9
Longest increasing subsequence: 0 1 3 4 5 6 9
Longest non-decreasing subsequence: 0 2 2 3 3 4 4 5 5 6 9
Longest decreasing subsequence: 7 6 5 3 2
```

# Disjoint Sets

Disjoint sets es una estructura de datos que permite, muy rápidamente, pegar elementos. Tiene heurística de compresión.

- Tiempo para merge y FindRoot: Amortizado $O(\log^*(n))$

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

class disjoint_sets
{
public:
    using size_type = std::int64_t;
    using index_type = std::int64_t;

    explicit disjoint_sets(index_type n) : parent(n), m_num_components(n)
    {
        std::iota(parent.begin(), parent.end(), 0L);
    }

    index_type find_root(index_type t)
    {
        std::vector<index_type> branch;
        branch.emplace_back(t);
        while (t != parent[t])
        {
            t = parent[t];
            branch.emplace_back(t);
        }
        for (auto u : branch)
            parent[u] = t;
        return t;
    }

    void reset()
    {
        std::iota(parent.begin(), parent.end(), 0);
        m_num_components = size();
    }

    void merge(index_type a, index_type b)
    {
```

```cpp
        index_type ra = find_root(a);
        index_type rb = set_parent(b, ra);

        if (ra != rb)
            --m_num_components;
    }

    bool are_in_same_connected_component(index_type a, index_type b)
    {
        return find_root(a) == find_root(b);
    }

    size_type num_components() const { return m_num_components; }

    index_type size() const { return parent.size(); }

    auto& parents() const { return parent; }

private:
    // returns ORIGINAL parent of x
    index_type set_parent(index_type x, index_type p)
    {
        while (x != parent[x])
        {
            index_type t = parent[x];
            parent[x] = p;
            x = t;
        }
        parent[x] = p;
        return x;
    }

    std::vector<index_type> parent;
    size_type m_num_components;
};

int main()
{
    disjoint_sets D(4);

    std::cout << "Num components: " << D.num_components() << std::endl;
    D.merge(0, 1);
    std::cout << "Num components: " << D.num_components() << std::endl;
    D.merge(2, 3);
    std::cout << "Num components: " << D.num_components() << std::endl;
```

```
    D.merge(0, 3);
    std::cout << "Num components: " << D.num_components() << std::endl;
    D.merge(1, 2);
    std::cout << "Num components: " << D.num_components() << std::endl;

    return 0;
}
```

**Output**:

```
Num components: 4
Num components: 3
Num components: 2
Num components: 1
Num components: 1
```

# Disjoint Intervals

Disjoint Intervals es una estructura de datos que representa una unión de intervalos cerrado-abiertos disjuntos de $\mathbb{R}$.

- Tiempo para insertar: $O(\log(n))$.
- Tiempo para buscar si existe: $O(\log(n))$.

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <set>
#include <vector>

// Closed-open interval [L,R)
template <class T>
struct Interval
{
    using value_type = T;
    Interval() : L(0), R(0) {}
    Interval(T l, T r) : L(l), R(r) {}
    T L;
    T R;
    T size() const { return R - L; }
};

template <class T>
bool operator<(const Interval<T>& A, const Interval<T>& B)
{
    if (A.L != B.L)
        return A.L < B.L;
    return A.R < B.R;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const Interval<T>& I)
{
    os << "[" << I.L << ", " << I.R << ")";
    return os;
}

using namespace std;

template <class T>
class DisjointIntervals
```

```
{
public:
    using value_type = Interval<T>;
    using iterator = typename std::set<Interval<T>>::iterator;
    using const_iterator = typename std::set<Interval<T>>::const_iterator;

    static constexpr T INF = std::numeric_limits<T>::max();

    const_iterator Insert(T a, T b) { return Insert({a, b}); }

    const_iterator FirstThatContainsOrEndsAt(T x)
    {
        auto first = lower_bound({x, x});

        if (first == m_data.begin())
            return first;

        // guaranteed to exist, since first != m_data.begin()
        auto prev = std::prev(first);

        if (prev->R >= x)
            return prev;

        return first;
    }

    const_iterator Insert(const Interval<T>& I)
    {
        auto L = I.L;
        auto R = I.R;

        //   L-----R
        // ---          <- This is the first that
        // could intersect (if it exists)
        auto first_possible = FirstThatContainsOrEndsAt(L);

        if (first_possible == m_data.end() || first_possible->L > R)
            return m_data.insert(I).first;

        L = std::min(L, first_possible->L);

        //   L-----R
        //           ---      <- First whose left
        //           is strictly > R
        auto last_possible = upper_bound({R, INF});
```

```
        // guaranteed to exist, since first_possible != m_data.end()
        auto last_intersected = std::prev(last_possible);
        R = std::max(R, last_intersected->R);

        // Erase the whole range that intersects [L,R)
        m_data.erase(first_possible, last_possible);

        return m_data.insert({L, R}).first;
    }

    const_iterator lower_bound(const Interval<T>& I) const
    {
        return m_data.lower_bound(I);
    }

    const_iterator upper_bound(const Interval<T>& I) const
    {
        return m_data.upper_bound(I);
    }

    const auto& Intervals() const { return m_data; }

private:
    std::set<Interval<T>> m_data;
};

template <class T>
std::ostream& operator<<(std::ostream& os, const DisjointIntervals<T>& D)
{
    auto& I = D.Intervals();

    auto it = I.begin();
    if (it == I.end())
    {
        os << "empty";
        return os;
    }

    os << *it;
    ++it;

    for (; it != I.end(); ++it)
    {
        os << " U " << *it;
```

```
    }
    return os;
}

using namespace std;
// Example program
int main()
{
    DisjointIntervals<int> D;
    D.Insert(0, 4);
    cout << D << endl;
    D.Insert(2, 8); // Intersects on the right
    cout << D << endl;
    D.Insert(-2, 1); // Intersects on the left
    cout << D << endl;
    D.Insert(-3, 9); // Contains
    cout << D << endl;

    D.Insert(15, 24); // Doesn't intersect at all
    cout << D << endl;

    D.Insert(10, 12); // In between, no intersect
    cout << D << endl;

    D.Insert(12, 15); // Joins two existing ones.
    cout << D << endl;

    return 0;
}
```

**Output**:

```
[0, 4)
[0, 8)
[-2, 8)
[-3, 9)
[-3, 9) U [15, 24)
[-3, 9) U [10, 12) U [15, 24)
[-3, 9) U [10, 24)
```

# Range Minimum Query

Dada una lista, permite preprocesarla para poder contestar preguntas de tipo "¿Cuál es el índice con el valor mínimo en el rango [L,R)?"

- Tiempo de preprocesamiento: $O(n \log(n))$
- Tiempo para contestar pregunta: $O(1)$.

Permite definir qué significa "menor qué".

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

template <typename RAContainer,
          typename Compare = std::less<typename RAContainer::value_type>>
class range_min_query
{
    using index_type = std::make_signed_t<size_t>;
    using Row = std::vector<index_type>;
    using value_type = typename RAContainer::value_type;

public:
    range_min_query(const RAContainer& A,
                    Compare comp = std::less<value_type>())
        : A_(A), T(A.size(), Row(std::log2(A.size()) + 1, -1)), comp_(comp)
    {
        index_type n = A.size();
        index_type max_h = T[0].size();

        for (index_type x = 0; x < n; ++x)
        {
            T[x][0] = x;
        }

        for (index_type h = 1; h < max_h; ++h)
        {
            for (index_type x = 0; x < n; ++x)
            {
                if (x + (1 << h) <= n)
                {
                    index_type mid = x + (1 << (h - 1));
                    T[x][h] = best(T[x][h - 1], T[mid][h - 1]);
```

```
                }
            }
        }
    }

    // Get min index in range [L,R)
    index_type GetMinIndex(index_type L, index_type R) const
    {
        assert(0 <= L && L < R && R <= A_.size());
        index_type h = std::log2(R - L);

        index_type min_index_starting_at_L = T[L][h];
        index_type min_index_ending_at_R = BestEndingAt(R - 1, h);

        return best(min_index_starting_at_L, min_index_ending_at_R);
    }

private:
    // A reference to the original container
    const RAContainer& A_;

    // T[x][i] contains the index of the
    // minimum of range [x,x+1,...,x+2^i)
    std::vector<Row> T;

    Compare comp_;

    index_type best(index_type i, index_type j) const
    {
        if (comp_(A_[j], A_[i]))
            return j;
        return i;
    }

    index_type BestEndingAt(index_type R, index_type h) const
    {
        return T[R - (1 << h) + 1][h];
    }
};

// This function is deprecated with C++17, but useful in c++14 and 11
template <typename RAContainer,
          typename Compare = std::less<typename RAContainer::value_type>>
range_min_query<RAContainer, Compare> make_range_min_query(
  const RAContainer& A,
```

```cpp
  Compare comp = std::less<typename RAContainer::value_type>())
{
    return range_min_query<RAContainer, Compare>(A, comp);
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    std::vector<int> A = {1, 5, 3,  9, 6, 10, 1, 5, 7, 9, 8, 0, 7,
                          4, 2, 10, 2, 3, 8,  6, 5, 7, 8, 9, 9};

    auto RMQ = make_range_min_query(A);
    auto GRMQ = make_range_min_query(A, std::greater<>());

    cout << "A = " << A << endl;

    cout << "Min value between index 5 and index 15 is at: "
         << RMQ.GetMinIndex(5, 15) << " with val " << A[RMQ.GetMinIndex(5, 15)]
         << std::endl;

    cout << "And the max value is at: " << GRMQ.GetMinIndex(5, 15)
         << " with val " << A[GRMQ.GetMinIndex(5, 15)] << endl;
}
```

**Output**:

```
A = 1 5 3 9 6 10 1 5 7 9 8 0 7 4 2 10 2 3 8 6 5 7 8 9 9
Min value between index 5 and index 15 is at: 11 with val 0
And the max value is at: 5 with val 10
```

# Linear optimization (Linear Programming) simplex

**NO ESCRITO POR MI**.

Este programa resuelve problemas de optimización lineal de la forma:

$$\text{Maximiza } c^T \cdot x$$

$$\text{Sujeto a } Ax \leq b$$

$$x \geq 0$$

```
// This program was written by jaehyunp and distributed under the MIT license.
// Taken from: https://github.com/jaehyunp/stanfordacm/blob/master/code/

// It has been slightly modified (modernized to C++, mainly) by mraggi

// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//          above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <cmath>
#include <iomanip>
#include <iostream>
#include <limits>
#include <vector>

using namespace std;

using DOUBLE = long double; // change to double to trade accuracy for speed.
using Row = vector<DOUBLE>;
using Matrix = vector<Row>;
using VI = vector<int>;
```

```
const DOUBLE EPS = 1e-9;

struct LPSolver
{
    int m, n;
    VI B, N;
    Matrix D;

    LPSolver(const Matrix& A, const Row& b, const Row& c)
        : m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, Row(n + 2))
    {
        for (int i = 0; i < m; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                D[i][j] = A[i][j];
            }
        }

        for (int i = 0; i < m; ++i)
        {
            B[i] = n + i;
            D[i][n] = -1;
            D[i][n + 1] = b[i];
        }

        for (int j = 0; j < n; ++j)
        {
            N[j] = j;
            D[m][j] = -c[j];
        }

        N[n] = -1;
        D[m + 1][n] = 1;
    }

    void Pivot(int r, int s)
    {
        double inv = 1.0/D[r][s];

        for (int i = 0; i < m + 2; ++i)
        {
            if (i != r)
            {
                for (int j = 0; j < n + 2; ++j)
```

```
            {
                if (j != s)
                    D[i][j] -= D[r][j]*D[i][s]*inv;
            }
        }
    }

    for (int j = 0; j < n + 2; ++j)
        if (j != s)
            D[r][j] *= inv;

    for (int i = 0; i < m + 2; ++i)
        if (i != r)
            D[i][s] *= -inv;

    D[r][s] = inv;
    swap(B[r], N[s]);
}

bool Simplex(int phase)
{
    int x = phase == 1 ? m + 1 : m;
    while (true)
    {
        int s = -1;
        for (int j = 0; j <= n; ++j)
        {
            if (phase == 2 && N[j] == -1)
                continue;
            if (s == -1 || D[x][j] < D[x][s] ||
                (D[x][j] == D[x][s] && N[j] < N[s]))
                s = j;
        }

        if (D[x][s] > -EPS)
            return true;

        int r = -1;
        for (int i = 0; i < m; ++i)
        {
            if (D[i][s] < EPS)
                continue;
            if (r == -1 || D[i][n + 1]/D[i][s] < D[r][n + 1]/D[r][s] ||
                ((D[i][n + 1]/D[i][s]) == (D[r][n + 1]/D[r][s]) &&
                 B[i] < B[r]))
```

```
                r = i;
        }

        if (r == -1)
            return false;

        Pivot(r, s);
    }
}

DOUBLE Solve(Row& x)
{
    int r = 0;
    for (int i = 1; i < m; ++i)
    {
        if (D[i][n + 1] < D[r][n + 1])
            r = i;
    }

    if (D[r][n + 1] < -EPS)
    {
        Pivot(r, n);

        if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
            return -numeric_limits<DOUBLE>::infinity();

        for (int i = 0; i < m; ++i)
        {
            if (B[i] == -1)
            {
                int s = -1;
                for (int j = 0; j <= n; ++j)
                    if (s == -1 || D[i][j] < D[i][s] ||
                        (D[i][j] == D[i][s] && N[j] < N[s]))
                        s = j;
                Pivot(i, s);
            }
        }
    }

    if (!Simplex(2))
        return numeric_limits<DOUBLE>::infinity();

    x = Row(n);
```

```
        for (int i = 0; i < m; ++i)
        {
            if (B[i] < n)
                x[B[i]] = D[i][n + 1];
        }

        return D[m][n + 1];
    }
};

int main()
{
    Matrix A = {{6, -1, 0}, {-1, -5, 0}, {1, 5, 1}, {-1, -5, -1}};
    Row b = {10, -4, 5, -5};
    Row c = {1, -1, 0};

    LPSolver solver(A, b, c);
    Row x;
    DOUBLE value = solver.Solve(x);

    cout << "VALUE: " << value << endl; // VALUE: 1.29032
    cout << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
    for (auto t : x)
        cout << ' ' << t;
    cout << endl;
    return 0;
}
```

**Output**:

```
VALUE: 1.29032
SOLUTION: 1.74194 0.451613 1
```

# Números Naturales

Clase muy simple para iterar en el rango $n = \{0, 1, ..., n - 1\}$. Otras clases la utilizan.

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>

template <class IntType>
class basic_natural_number
{
public:
    using difference_type = long long;
    using size_type = long long;
    using value_type = IntType;
    class iterator;
    using const_iterator = iterator;

public:
    explicit basic_natural_number(IntType n) : m_n(n) { assert(n >= 0); }

    size_type size() const { return m_n; }

    class iterator
    {
    public:
        using iterator_category = std::random_access_iterator_tag;
        using value_type = IntType;
        using difference_type = long long;
        using pointer = IntType const*;
        using reference = const IntType&;

        explicit iterator(IntType t = 0) : m_ID(t) {}

        inline iterator& operator++()
        {
            ++m_ID;
            return *this;
        }
        inline iterator& operator--()
        {
            --m_ID;
            return *this;
        }
```

```
        inline const IntType& operator*() const { return m_ID; }

        inline iterator& operator+=(difference_type n)
        {
            m_ID += n;
            return *this;
        }

        inline iterator& operator-=(difference_type n)
        {
            return operator+=(-n);
        }

        inline bool operator==(const iterator& it) { return *it == m_ID; }

        inline bool operator!=(const iterator& it) { return *it != m_ID; }

        inline difference_type operator-(const iterator& it)
        {
            return *it - m_ID;
        }

    private:
        IntType m_ID{0};

        friend class basic_natural_number;
    }; // end class iterator

    iterator begin() const { return iterator(0); }

    iterator end() const { return iterator(m_n); }

    IntType operator[](size_type m) const { return m; }

    template <class Pred>
    IntType partition_point(Pred p)
    {
        return *std::partition_point(begin(), end(), p);
    }

private:
    IntType m_n;
}; // end class basic_natural_number

template <class IntType>
```

```
inline typename basic_natural_number<IntType>::iterator
operator+(typename basic_natural_number<IntType>::iterator it,
          typename basic_natural_number<IntType>::difference_type n)
{
    it += n;
    return it;
}

template <class IntType>
inline typename basic_natural_number<IntType>::iterator
operator-(typename basic_natural_number<IntType>::iterator it,
          typename basic_natural_number<IntType>::difference_type n)
{
    it -= n;
    return it;
}

using natural_number = basic_natural_number<int>;
using big_natural_number = basic_natural_number<long long>;

template <class Container, class T = typename Container::size_type>
basic_natural_number<T> indices(const Container& C)
{
    return basic_natural_number<T>(C.size());
}

int main()
{
    using std::cout;
    using std::endl;

    for (int i : natural_number(5))
        cout << i << ' ';

    cout << endl;

    std::vector<int> W = {2, 4, 6, 8};
    for (auto i : indices(W))
        cout << i << ": " << W[i] << endl;

    return 0;
}
```

**Output**:

```
0 1 2 3 4
```

```
0: 2
1: 4
2: 6
3: 8
```

# Graph

Clase que representa un grafo. Por sí solo no hace nada.

**REQUIERE:** NaturalNumber

**REQUERIDO POR:** Bipartite, MinSpanningTree, Shortest Paths, etc.

```cpp
#pragma once

#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>

#include "NaturalNumber.hpp"

template <class Iter, class T>
Iter find_binary(const Iter& first, const Iter& last, const T& t)
{
    auto it = std::lower_bound(first, last, t);

    if (it == last || *it != t)
        return last;

    return it;
}

// simple undirected graph
class Graph
{
public:
    using size_type = long long;

    using Vertex = std::int64_t;

    enum WORKAROUND_UNTIL_CPP17
    {
        INVALID_VERTEX = -1
    };
    //  inline static constexpr Vertex INVALID_VERTEX = -1; // Uncomment with
    // c++17

    using weight_t = std::int64_t;

    // something larger than weight_t, for when you have that weight_t doesn't
```

```
// properly hold a sum of weight_t (for example, if weight_t = char).
using sumweight_t = std::int64_t;

struct Neighbor; // Represents a half-edge (vertex,weight)

struct Edge; // (from,to,weight)

using neighbor_list = std::vector<Neighbor>;
using neighbor_const_iterator = neighbor_list::const_iterator;
using neighbor_iterator = neighbor_list::iterator;

explicit Graph(Vertex numberOfVertices = 0)
    : m_numvertices(std::max<Vertex>(0, numberOfVertices))
    , m_graph(m_numvertices)
{}

size_type degree(Vertex a) const { return m_graph[a].size(); }

// Graph modification functions
Vertex add_vertex()
{
    m_graph.emplace_back(); // empty vector
    return m_numvertices++;
}

void add_edge(Vertex from, Vertex to, weight_t w = 1)
{
    m_graph[from].emplace_back(to, w);
    m_graph[to].emplace_back(from, w);
    ++m_numedges;
    m_neighbors_sorted = false;
}

void add_edge(const Edge& e) { add_edge(e.from, e.to, e.weight()); }

template <class EdgeContainer>
void add_edges(const EdgeContainer& edges)
{
    for (auto& e : edges)
        add_edge(e);
}

void add_edges(const std::initializer_list<Edge>& edges)
{
    for (auto& e : edges)
```

```
        add_edge(e);
}

bool add_edge_no_repeat(Vertex from, Vertex to, weight_t w = 1)
{
    if (is_neighbor(from, to))
        return false;

    add_edge(from, to, w);
    return true;
}

void sort_neighbors()
{
    if (m_neighbors_sorted)
        return;

    for (auto& adj_list : m_graph)
        sort(adj_list.begin(), adj_list.end());

    m_neighbors_sorted = true;
}

// Get Graph Info
Vertex num_vertices() const { return m_numvertices; }
size_type num_edges() const { return m_numedges; }

inline const neighbor_list& neighbors(Vertex n) const { return m_graph[n]; }
inline const neighbor_list& outneighbors(Vertex n) const
{
    return m_graph[n];
}
inline const neighbor_list& inneighbors(Vertex n) const
{
    return m_graph[n];
}

using all_vertices = basic_natural_number<Vertex>;
auto vertices() const { return all_vertices(num_vertices()); }

std::vector<Edge> edges() const
{
    std::vector<Edge> total;

    for (auto u : vertices())
```

```
    {
        for (auto v : m_graph[u])
        {
            if (v > u)
                total.emplace_back(u, v, v.weight());
        }
    }

    return total;
}

bool is_neighbor(Vertex from, Vertex to) const
{
    if (degree(from) > degree(to))
        std::swap(from, to);

    auto& NF = neighbors(from);

    if (m_neighbors_sorted)
        return std::binary_search(NF.begin(), NF.end(), to);

    for (auto& a : NF)
    {
        if (a == to)
            return true;
    }

    return false;
}

weight_t edge_value(Vertex from, Vertex to) const
{
    if (degree(from) > degree(to))
        std::swap(from, to);

    auto neigh = get_neighbor(from, to);

    if (neigh == neighbors(from).end() || *neigh != to)
        return 0;

    return neigh->weight();
}

neighbor_const_iterator get_neighbor(Vertex from, Vertex to) const
{
```

```
    auto first = m_graph[from].begin();
    auto last = m_graph[from].end();

    if (m_neighbors_sorted)
        return find_binary(first, last, to);

    return std::find(first, last, to);
}

neighbor_iterator get_neighbor(Vertex from, Vertex to)
{
    auto first = m_graph[from].begin();
    auto last = m_graph[from].end();

    if (m_neighbors_sorted)
        return find_binary(first, last, to);

    return std::find(first, last, to);
}

// Start class definitions
struct Neighbor
{
    explicit Neighbor() : vertex(INVALID_VERTEX), m_weight(0) {}

    explicit Neighbor(Vertex v, weight_t w = 1) : vertex(v), m_weight(w) {}

    inline operator Vertex() const { return vertex; }

    weight_t weight() const { return m_weight; }

    void set_weight(weight_t w) { m_weight = w; }

    Vertex vertex{INVALID_VERTEX};

private:
    // comment out if not needed, and make set_weight do nothing, and make
    // weight() return 1
    weight_t m_weight{1};
};

struct Edge
{
    Vertex from{INVALID_VERTEX};
    Vertex to{INVALID_VERTEX};
```

```
        Edge() : m_weight(0) {}
        Edge(Vertex f, Vertex t, weight_t w = 1) : from(f), to(t), m_weight(w)
        {}

        Vertex operator[](bool i) const { return i ? to : from; }

        // replace by "return 1" if weight doesn't exist
        weight_t weight() const { return m_weight; }
        void change_weight(weight_t w) { m_weight = w; }

        bool operator==(const Edge& E) const
        {
            return ((from == E.from && to == E.to) ||
                    (from == E.to && to == E.from)) &&
              m_weight == E.m_weight;
        }

    private:
        weight_t m_weight{1};
    };

private:
    // Graph member variables
    size_type m_numvertices{0};
    size_type m_numedges{0};

    std::vector<neighbor_list> m_graph{};
    bool m_neighbors_sorted{false};
};
```

# Connected Components

Encuentra las componentes conexas de un grafo. Regresa un vector cuyo $i$-ésimo valor es la componente conexa a la que pertence el vértice $i$.

**REQUIERE:** Graph

- Tiempo de ejecución: $O(E)$.

```cpp
#include <stack>

#include "Graph.hpp"

using Vertex = Graph::Vertex;

// connected_components(G)[i] = connected component of the i-th vertex.
std::vector<int> connected_components(const Graph& G)
{
    auto n = G.num_vertices();
    std::vector<int> components(n, -1);
    int current_component = 0;

    for (auto v : G.vertices())
    {
        if (components[v] != -1)
            continue;

        std::stack<Vertex> frontier;
        frontier.emplace(v);

        while (!frontier.empty())
        {
            auto p = frontier.top();
            frontier.pop();

            if (components[p] != -1)
                continue;

            components[p] = current_component;

            for (auto u : G.neighbors(p))
            {
                if (components[u] == -1)
                    frontier.emplace(u);
            }
        }
```

```cpp
        ++current_component;
    }

    return components;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    Graph G(5);

    G.add_edge(0, 1);
    G.add_edge(2, 3);

    std::cout << connected_components(G) << std::endl;

    return 0;
}
```

**Output**:

```
0 0 1 1 2
```

# Algoritmos básicos en árboles.

Funciones de utilidad para cuando un grafo es árbol. La función `set_root` regresa un vector con el padre de cada vértice, (-1 es el padre de la raíz).

La función `height_map` regresa la altura del vértice. Equivalente (pero más rápido) a correr dijkstra.

**REQUIERE:** Graph

```
#include "Graph.hpp"

#include <cmath>
#include <set>
#include <stack>

using Vertex = Graph::Vertex;

std::vector<Vertex> set_root(const Graph& G, Vertex root)
{
    std::vector<Vertex> parent(G.num_vertices());

    parent[root] = Graph::INVALID_VERTEX;

    std::stack<Vertex> frontier;
    frontier.emplace(root);

    while (!frontier.empty())
    {
        auto p = frontier.top();
        frontier.pop();

        for (auto u : G.neighbors(p))
        {
            if (parent[p] == u)
                continue;
            parent[u] = p;
            frontier.emplace(u);
        }
    }
    return parent;
}

std::vector<int> height_map(const Graph& G, Vertex root)
{
    std::vector<int> level(G.num_vertices(), -1);
```

```
    level[root] = 0;

    std::stack<Vertex> frontier;
    frontier.emplace(root);

    while (!frontier.empty())
    {
        auto p = frontier.top();
        frontier.pop();
        int current_level = level[p];

        for (auto u : G.neighbors(p))
        {
            if (level[u] != -1)
                continue;
            level[u] = current_level + 1;
            frontier.emplace(u);
        }
    }

    return level;
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    Graph tree(5);
    tree.add_edge(1, 0);
    tree.add_edge(1, 2);
    tree.add_edge(2, 3);
    tree.add_edge(2, 4);

    auto parents = set_root(tree, 1);
    std::cout << "Parents: " << parents << std::endl;
```

```
    auto height = height_map(tree, 1);
    std::cout << "Heights: " << height << std::endl;

    return 0;
}
```

**Output**:

```
Parents: 1 -1 1 2 2
Heights: 1 0 1 2 2
```

# Árbol Generador de Peso Mínimo (MST)

Dado un grafo, encuentra el árbol generador de peso mínimo.

Se incluyen dos algoritmos: Prim y Kruskal. En la práctica es más rápido Prim, aunque hay varios problemas que se resuelven con un algoritmo que sea una modificación de uno de ellos.

- Tiempo: $O(E \log(E))$

**REQUIERE:** Graph, DisjointSets (para kruskal)

```
#include "DisjointSets.hpp"
#include "Graph.hpp"

#include <cmath>
#include <queue>
#include <set>
#include <stack>

using Vertex = Graph::Vertex;
using Edge = Graph::Edge;

struct by_reverse_weight // for prim
{
    template <class T>
    bool operator()(const T& a, const T& b)
    {
        return a.weight() > b.weight();
    }
};

struct by_weight // for kruskal
{
    template <class T>
    bool operator()(const T& a, const T& b)
    {
        return a.weight() < b.weight();
    }
};

std::vector<Graph::Edge> prim(const Graph& G)
{
    auto n = G.num_vertices();

    std::vector<Edge> T;
    if (n < 2)
```

```
        return T;

    Vertex num_tree_edges = n - 1;

    T.reserve(num_tree_edges);

    std::vector<bool> explored(n, false);

    std::priority_queue<Edge, std::vector<Edge>, by_reverse_weight>
      EdgesToExplore;

    explored[0] = true;
    for (auto v : G.neighbors(0))
    {
        EdgesToExplore.emplace(0, v, v.weight());
    }

    while (!EdgesToExplore.empty())
    {
        Edge s = EdgesToExplore.top();
        EdgesToExplore.pop();

        if (explored[s.to])
            continue;

        T.emplace_back(s);

        --num_tree_edges;
        if (num_tree_edges == 0)
            return T;

        explored[s.to] = true;
        for (auto v : G.neighbors(s.to))
        {
            if (!explored[v])
                EdgesToExplore.emplace(s.to, v, v.weight());
        }
    }
    return T;
}

std::vector<Graph::Edge> kruskal(const Graph& G)
{
    auto n = G.num_vertices();
    Vertex num_tree_edges = n - 1;
```

```cpp
    std::vector<Graph::Edge> T;
    T.reserve(num_tree_edges);

    auto E = G.edges();

    std::sort(E.begin(), E.end(), by_weight{});

    disjoint_sets D(G.num_vertices());

    for (auto& e : E)
    {
        Vertex a = e.from;
        Vertex b = e.to;

        if (!D.are_in_same_connected_component(a, b))
        {
            D.merge(a, b);
            T.emplace_back(e);
            --num_tree_edges;
            if (num_tree_edges == 0)
                return T;
        }
    }

    return T;
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    Graph G(5);
    G.add_edge(0, 1, 5);
    G.add_edge(0, 2, 2);
    G.add_edge(0, 3, 4);
    G.add_edge(1, 2, 1);
```

```cpp
    G.add_edge(1, 3, 8);
    G.add_edge(1, 4, 7);
    G.add_edge(2, 3, 3);
    G.add_edge(2, 4, 2);
    G.add_edge(3, 4, 9);

    cout << "Prim has the following edges:" << endl;
    for (auto e : prim(G))
    {
        cout << "(" << e.from << "," << e.to << "," << e.weight() << ")\n";
    }

    cout << "\nKruskal has the following edges:" << endl;
    for (auto e : kruskal(G))
    {
        cout << "(" << e.from << "," << e.to << "," << e.weight() << ")\n";
    }

    return 0;
}
```

**Output**:

```
Prim has the following edges:
(0,2,2)
(2,1,1)
(2,4,2)
(2,3,3)

Kruskal has the following edges:
(1,2,1)
(0,2,2)
(2,4,2)
(2,3,3)
```

# Lowest Common Ancestors

Una clase que, dado un árbol, puede responder a la pregunta “¿quién es el ancestro común más cercano de dos vértices $u$ y $v$?” rápidamente.

Se incluyen sólo la implementación de los $2^i$-ancestros. Hay una mejor pero más complicada de escribir.

- Tiempo de preprocesamiento: $O(n \log(n))$.
- Tiempo para pregunta: $O(\log(n))$

**REQUIERE:** Graph, Tree

```
#include <stack>

#include "Graph.hpp"
#include "Tree.hpp"

using Vertex = Graph::Vertex;

class LCA
{
public:
    using Vertex = Graph::Vertex;
    LCA(const Graph& G, Vertex root)
        : L(height_map(G, root))
        , A(G.num_vertices(),
            std::vector<Vertex>(
              std::log2(*std::max_element(L.begin(), L.end()) + 1) + 1, -1))
    {
        auto parents = set_root(G, root);

        // The 2^0-th ancestor of v is simply the parent of v
        for (auto v : G.vertices())
            A[v][0] = parents[v];

        for (int i = 2; i < log_height(); ++i)
        {
            for (auto v : G.vertices())
            {
                // My 2^i-th ancestor is the 2^{i-1} ancestor of my 2^{i-1}
                // ancestor!
                if (A[v][i - 1] != -1)
                    A[v][i] = A[A[v][i - 1]][i - 1];
            }
        }
```

```
    }

    Vertex FindLCA(Vertex u, Vertex v) const
    {
        if (L[u] < L[v])
            std::swap(u, v);

        u = AncestorAtLevel(u, L[v]);

        if (u == v)
            return u;

        for (int i = std::log2(L[u]); i >= 0; --i)
        {
            if (A[u][i] != -1 && A[u][i] != A[v][i])
            {
                u = A[u][i];
                v = A[v][i];
            }
        }

        return A[u][0]; // which is = A[v][0]
    }

    const std::vector<std::vector<Vertex>>& Ancestors() const { return A; }
    const auto& Levels() const { return L; }

private:
    // L[v] is the level (distance to root) of vertex v
    std::vector<int> L;

    // A[v][i] is the 2^i ancestor of vertex v
    std::vector<std::vector<Vertex>> A;

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

    Vertex AncestorAtLevel(Vertex u, int lvl) const
    {
        int d = L[u] - lvl;
        assert(d >= 0);

        while (d > 0)
        {
            int h = std::log2(d);
            u = A[u][h];
```

```
            d -= (1 << h);
        }

        return u;
    }
};

int main()
{
    Graph tree(5);
    tree.add_edge(1, 0);
    tree.add_edge(1, 2);
    tree.add_edge(2, 3);
    tree.add_edge(2, 4);

    LCA lca(tree, 1);

    std::cout << "LCA of 0 and 4: " << lca.FindLCA(0, 4) << std::endl;
    std::cout << "LCA of 3 and 4: " << lca.FindLCA(3, 4) << std::endl;

    return 0;
}
```

**Output**:

```
LCA of 0 and 4: 1
LCA of 3 and 4: 2
```

# Shortest Paths

Dado un grafo y un vértice inicial, encuentra el camino de menor peso a un objetivo.

Se incluyen dos algoritmos: Dijkstra y A*.

**REQUIERE:** Graph

```
#include "Graph.hpp"

#include <cmath>
#include <deque>
#include <queue>
#include <set>
#include <stack>

using Vertex = Graph::Vertex;
using Edge = Graph::Edge;
using Distance = Graph::sumweight_t;

const auto INF = std::numeric_limits<Distance>::max();

// Used by both A* and Dijkstra
template <class Path = std::deque<Graph::Neighbor>>
inline Path PathFromParents(Vertex origin,
                            Vertex destination,
                            const std::vector<Distance>& distance,
                            const std::vector<Vertex>& parent)
{
    Path P;

    if (origin == destination)
    {
        P.emplace_front(origin, 0);
        return P;
    }

    auto remaining = distance[destination];

    if (remaining == INF)
        return P;

    do
    {
        auto previous = destination;
        destination = parent[destination];
```

```
        auto d = distance[previous] - distance[destination];
        P.emplace_front(previous, d);
    } while (destination != origin);

    P.emplace_front(origin, 0);

    return P;
}

//------------------ Start Dijsktra Searcher

struct DummyPath
{
    DummyPath(Vertex v, Distance d) : last(v), length(d) {}
    Vertex last;
    Distance length;
};

bool operator<(const DummyPath& a, const DummyPath& b)
{
    return a.length > b.length;
}

class DijkstraSearcher
{
public:
    // If destination is invalid, it constructs all single-source shortest
    // paths. If destination is a specific vertex, the searcher stops when it
    // finds it.
    DijkstraSearcher(const Graph& G,
                     Vertex origin_,
                     Vertex destination_ = Graph::INVALID_VERTEX)
        : origin(origin_)
        , destination(destination_)
        , distance(G.num_vertices(), INF)
        , parent(G.num_vertices(), -1)
    {
        distance[origin] = 0;

        std::priority_queue<DummyPath> frontier;

        frontier.emplace(origin, 0);

        while (!frontier.empty())
        {
```

```
            auto P = frontier.top();
            frontier.pop();

            if (P.length > distance[P.last])
                continue;

            if (P.last == destination)
                break;

            for (auto& v : G.neighbors(P.last))
            {
                auto d = P.length + v.weight();
                if (distance[v] > d)
                {
                    distance[v] = d;
                    parent[v] = P.last;
                    frontier.emplace(v, d);
                }
            }
        }
    }

    // dest might be different from destination, if and only if, either
    // destination is Graph::INVALID_VERTEX or distance from origin to dest is
    // smaller than distance to destination.
    template <class Path = std::deque<Graph::Neighbor>>
    Path GetPath(Vertex dest = Graph::INVALID_VERTEX) const
    {
        if (dest == Graph::INVALID_VERTEX)
            dest = destination;

        assert(dest != Graph::INVALID_VERTEX);

        return PathFromParents<Path>(origin, dest, distance, parent);
    }

    Vertex Origin() const { return origin; }
    Vertex Destination() const { return destination; }

    const std::vector<Distance>& Distances() const { return distance; }
    const std::vector<Vertex>& Parents() const { return parent; }

private:
    Vertex origin;
    Vertex destination;
```

```
    std::vector<Distance> distance;
    std::vector<Vertex> parent;
};

template <class Path = std::deque<Graph::Neighbor>>
Path Dijkstra(const Graph& G, Vertex origin, Vertex destination)
{
    return DijkstraSearcher(G, origin, destination).GetPath<Path>();
}

//------------------- START A* searcher

struct DummyPathWithHeuristic
{
    DummyPathWithHeuristic(Vertex v, Distance c, Distance h)
        : last(v), cost(c), heuristic(h)
    {}
    Vertex last;
    Distance cost;
    Distance heuristic;

    Distance cost_plus_heuristic() const { return cost + heuristic; }
};

inline bool operator<(const DummyPathWithHeuristic& A,
                      const DummyPathWithHeuristic& B)
{
    if (A.cost_plus_heuristic() != B.cost_plus_heuristic())
        return A.cost_plus_heuristic() > B.cost_plus_heuristic();

    if (A.heuristic != B.heuristic)
        return A.heuristic > B.heuristic;

    // if same cost plus heuristic, whatever.
    return A.last > B.last;
}

// Managed to make this not a template by having templated constructors.
class AstarSearcher
{
public:
    // Finds a path from origin to destination using heuristic h
    template <class Heuristic>
    AstarSearcher(const Graph& G,
                  Vertex origin_,
```

```cpp
                  Vertex destination_,
                  Heuristic h)
        : origin(origin_)
        , destination(destination_)
        , distance(G.num_vertices(), INF)
        , parent(G.num_vertices(), Graph::INVALID_VERTEX)
    {
        auto objective = [destination_](Vertex v) { return v == destination_; };
        Init(G, objective, h);
    }

    // Finds a path from origin to some destination that satisfies predicte
    // objective, using heuristic h
    template <class Objective, class Heuristic>
    AstarSearcher(const Graph& G,
                  Vertex origin_,
                  Objective objective,
                  Heuristic h)
        : origin(origin_)
        , destination(Graph::INVALID_VERTEX)
        , distance(G.num_vertices(), INF)
        , parent(G.num_vertices(), Graph::INVALID_VERTEX)
    {
        Init(G, objective, h);
    }

    template <class Path = std::deque<Graph::Neighbor>>
    Path GetPath() const
    {
        return PathFromParents<Path>(origin, destination, distance, parent);
    }

    Vertex Origin() const { return origin; }
    Vertex Destination() const { return destination; }

    Distance PathCost() const { return distance[destination]; }

    const std::vector<Distance>& Distances() const { return distance; }
    const std::vector<Vertex>& Parents() const { return parent; }

private:
    Vertex origin;
    Vertex destination;
    std::vector<Distance> distance;
    std::vector<Vertex> parent;
```

```
    template <class Heuristic, class Objective>
    void Init(const Graph& G, Objective objective, Heuristic h)
    {
        using std::cout;
        using std::endl;
        distance[origin] = 0;

        std::priority_queue<DummyPathWithHeuristic> frontier;

        frontier.emplace(origin, 0, h(origin));

        while (!frontier.empty())
        {
            auto P = frontier.top();
            frontier.pop();

            if (P.cost > distance[P.last])
                continue;

            if (objective(P.last))
            {
                destination = P.last;
                return;
            }

            for (auto& v : G.neighbors(P.last))
            {
                auto d = P.cost + v.weight();
                if (distance[v] > d)
                {
                    distance[v] = d;
                    parent[v] = P.last;
                    frontier.emplace(v, d, h(v));
                }
            }
        }
    }
};

template <class Objective,
          class Heuristic,
          class Path = std::deque<Graph::Neighbor>>
Path Astar(const Graph& G, Vertex origin, Objective objective, Heuristic h)
{
```

```cpp
    return AstarSearcher(G, origin, objective, h).GetPath<Path>();
}

int main()
{
    using std::cout;
    using std::endl;

    Graph G(5);
    G.add_edge(0, 1, 5);
    G.add_edge(0, 2, 9);
    G.add_edge(1, 2, 3);
    G.add_edge(2, 3, 4);
    G.add_edge(3, 4, 5);

    Vertex s = 0;
    Vertex t = 4;

    std::vector<int> heuristic = {5, 5, 4, 4, 0};

    cout << "Dijsktra produces the following path:\n\t";
    for (auto e : Dijkstra(G, s, t))
    {
        cout << "----(w = " << e.weight() << ")----> " << e.vertex << " ";
    }
    cout << endl << endl;

    auto h = [&heuristic](Vertex v) { return heuristic[v]; };

    cout << "A* produces the following path:\n\t";
    for (auto e : Astar(G, s, t, h))
    {
        cout << "----(w = " << e.weight() << ")----> " << e.vertex << " ";
    }

    return 0;
}
```

**Output**:

```
Dijsktra produces the following path:
    ----(w = 0)----> 0 ----(w = 5)----> 1 ----(w = 3)----> 2 ----(w = 4)----> 3 ----(w =

A* produces the following path:
    ----(w = 0)----> 0 ----(w = 5)----> 1 ----(w = 3)----> 2 ----(w = 4)----> 3 ----(w =
```

# BipartiteGraph

Clase que representa un grafo bipartito. Por sí solo no hace nada.

**REQUIERE**: Graph, NaturalNumber

**REQUERIDO POR:** BipartiteMatcher

```
#pragma once

// maybe not needed, only "Neighbor" and "Edge" are needed.
#include "Graph.hpp"

class BipartiteGraph
{
public:
    using size_type = std::int64_t;
    using Vertex = std::int64_t;
    using weight_t = std::int64_t;

    // something larger than weight_t, for when you have that weight_t doesn't
    // properly hold a sum of weight_t (for example, if weight_t = char).
    using sumweight_t = std::int64_t;

    using Neighbor = Graph::Neighbor; // Represents a half-edge (vertex,weight)

    using Edge = Graph::Edge; // (from,to,weight)

    using neighbor_list = std::vector<Neighbor>;
    using neighbor_const_iterator = neighbor_list::const_iterator;
    using neighbor_iterator = neighbor_list::iterator;
    /****** END using definitions *****/

public:
    BipartiteGraph(size_type x, size_type y) : m_X(x), m_Y(y) {}

    size_type degreeX(Vertex x) const { return m_X[x].size(); }
    size_type degreeY(Vertex y) const { return m_Y[y].size(); }

    size_type num_verticesX() const { return m_X.size(); }
    size_type num_verticesY() const { return m_Y.size(); }

    size_type num_vertices() const { return num_verticesX() + num_verticesY(); }

    using all_vertices = basic_natural_number<Vertex>;
    auto verticesX() const { return all_vertices(num_verticesX()); }
```

```cpp
auto verticesY() const { return all_vertices(num_verticesY()); }

const auto& X() const { return m_X; }
const auto& Y() const { return m_Y; }

const neighbor_list& neighborsX(Vertex a) const { return m_X[a]; }
const neighbor_list& neighborsY(Vertex a) const { return m_Y[a]; }

void add_edge(Vertex x, Vertex y, weight_t w = 1)
{
    m_X[x].emplace_back(y, w);
    m_Y[y].emplace_back(x, w);
    ++m_numedges;
    m_neighbors_sorted = false;
}

void add_edge(const Edge& E) { add_edge(E.from, E.to, E.weight()); }

template <class EdgeContainer>
void add_edges(const EdgeContainer& edges)
{
    for (auto& e : edges)
        add_edge(e);
}

void add_edges(const std::initializer_list<Edge>& edges)
{
    for (auto& e : edges)
        add_edge(e);
}

void FlipXandY() { std::swap(m_X, m_Y); }

void sort_neighbors()
{
    if (m_neighbors_sorted)
        return;

    for (auto& x : m_X)
        std::sort(std::begin(x), std::end(x));

    for (auto& y : m_Y)
        std::sort(std::begin(y), std::end(y));

    m_neighbors_sorted = true;
```

```
    }

    Graph UnderlyingGraph() const
    {
        Graph G(num_vertices());

        for (Vertex v = 0; v < num_verticesX(); ++v)
        {
            for (auto u : neighborsX(v))
            {
                G.add_edge(v, u + num_verticesX(), u.weight());
            }
        }

        return G;
    }

private:
    std::vector<neighbor_list> m_X{};
    std::vector<neighbor_list> m_Y{};
    size_type m_numedges{0};
    bool m_neighbors_sorted{false};
};
```

# Bipartite Matching

Encuentra el apareamiento máximo en una gráfica bipartita.

**REQUIERE:** BipartiteGraph

- Tiempo de ejecución: $O(VE)$, pero en general es muuucho más rápido que eso.

**Nota**: Encuentra el apareamiento de cardinalidad máxima, no el de peso máximo. Si se requiere max weight matching, mejor usar max flow con el truco de agregar dos vértices fantasmas.

```
#include "BipartiteGraph.hpp"
#include <deque>
#include <queue>
#include <stack>

class BipartiteMatcher
{
public:
    using Vertex = BipartiteGraph::Vertex;
    using Edge = Graph::Edge;

    BipartiteMatcher(const BipartiteGraph& G)
        : m_Xmatches(G.num_verticesX(), -1), m_Ymatches(G.num_verticesY(), -1)
    {
        CreateInitialMatching(G);
        Augment(G);
    }

    // MatchX(x) returns the matched vertex to x (-1 if none).
    Vertex MatchX(Vertex x) const { return m_Xmatches[x]; }
    Vertex MatchY(Vertex y) const { return m_Ymatches[y]; }

    int size() const { return m_size; }

    std::vector<Edge> Edges() const
    {
        std::vector<Edge> matching;
        matching.reserve(size());

        for (auto x : indices(m_Xmatches))
        {
            auto y = MatchX(x);
            if (y >= 0)
                matching.emplace_back(x, y);
```

```
        }

        return matching;
    }

private:
    void CreateInitialMatching(const BipartiteGraph& G)
    {
        m_unmatched_in_X.reserve(G.num_verticesX());

        for (auto x : G.verticesX())
        {
            for (auto y : G.neighborsX(x))
            {
                if (m_Ymatches[y] < 0)
                {
                    m_Xmatches[x] = y;
                    m_Ymatches[y] = x;
                    ++m_size;
                    break;
                }
            }

            if (m_Xmatches[x] < 0)
                m_unmatched_in_X.emplace_back(x);
        }
    }

    // returns false if no augmenting path was found
    void Augment(const BipartiteGraph& G)
    {
        size_t num_without_augment = 0;
        auto it = m_unmatched_in_X.begin();

        while (num_without_augment < m_unmatched_in_X.size())
        {
            // Imagine this a circular buffer.
            if (it == m_unmatched_in_X.end())
                it = m_unmatched_in_X.begin();

            if (FindAugmentingPath(G, *it))
            {
                // The following two lines erase it quickly by replacing it with
                // the last element of m_unmatched_in_X
                *it = m_unmatched_in_X.back();
```

```
            m_unmatched_in_X.pop_back();
            num_without_augment = 0;
        }
        else
        {
            ++it;
            ++num_without_augment;
        }
    }
}

bool FindAugmentingPath(const BipartiteGraph& G, Vertex x)
{
    const Vertex not_seen = -1;
    // In order to reconstruct the augmenting path.
    std::vector<Vertex> parent(G.num_verticesY(), -1);

    std::queue<Vertex> frontier; // BFS
    frontier.emplace(x);

    while (!frontier.empty())
    {
        auto current_x = frontier.front();
        frontier.pop();

        for (Vertex y : G.neighborsX(current_x))
        {
            if (parent[y] != not_seen)
                continue;

            parent[y] = current_x;

            auto new_x = m_Ymatches[y];
            if (new_x == -1)
            {
                ApplyAugmentingPath(y, parent);
                assert(m_Xmatches[x] != -1);
                return true;
            }

            frontier.emplace(new_x);
        }
    }

    return false;
```

```
    }

    void ApplyAugmentingPath(Vertex y, const std::vector<Vertex>& parent)
    {
        ++m_size;

        Vertex x = parent[y];
        do
        {
            auto new_y = m_Xmatches[x]; // save it because I'll erase it

            // new matches
            m_Ymatches[y] = x;
            m_Xmatches[x] = y;

            y = new_y;
            x = parent[y];
            assert(x != -1);
        } while (y != -1);
    }

    int m_size{0};
    std::vector<Vertex> m_Xmatches{}; // -1 if not matched
    std::vector<Vertex> m_Ymatches{}; // -1 if not matched

    std::vector<Vertex> m_unmatched_in_X{};
};

using namespace std;

int main()
{
    BipartiteGraph G(4, 4);
    G.add_edge(0, 3);
    G.add_edge(0, 1);
    G.add_edge(1, 0);
    G.add_edge(1, 1);
    G.add_edge(2, 0);
    G.add_edge(2, 1);
    G.add_edge(3, 0);
    G.add_edge(3, 2);
    G.add_edge(3, 3);

    BipartiteMatcher BM(G);
    cout << "Best match has size " << BM.size() << ", which is:" << endl;
```

```
    for (auto edge : BM.Edges())
    {
        cout << '(' << edge.from << ',' << edge.to << ')' << endl;
    }

    return 0;
}
```

**Output**:

```
0 0 1 1 2
```

# Maximum flow

ESTE CÓDIGO NO LO ESCRIBÍ YO.

Dada una gráfica dirigida con capacidades, una fuente y un pozo, encuentra el máximo flujo. Puede usarse para resolver mínimo corte, con el teorema de mínimo corte y máximo flujo, simplemente considerando todas las parejas de flujo $(0, v)$ con $v > 0$.

```
// This program was written by jaehyunp and distributed under the MIT license.
// Taken from: https://github.com/jaehyunp/stanfordacm/blob/master/code/

// It has been slightly modified (modernized to C++, mainly) by mraggi

#include <algorithm>
#include <iostream>
#include <numeric>
#include <queue>
#include <vector>

struct Edge
{
    long from, to, cap, flow, index_of_twin;
    Edge(long from, long to, long cap, long flow, long index_of_twin)
        : from(from), to(to), cap(cap), flow(flow), index_of_twin(index_of_twin)
    {}
};

class PushRelabel
{
public:
    PushRelabel(long N)
        : N(N), G(N), excess(N), dist(N), active(N), count(2*N)
    {}

    void AddEdge(long from, long to, long cap)
    {
        G[from].emplace_back(from, to, cap, 0, G[to].size());

        if (from == to)
            ++G[from].back().index_of_twin;

        G[to].emplace_back(to, from, 0, 0, G[from].size() - 1);
    }

    long GetMaxFlow(long s, long t)
```

```cpp
    {
        count[0] = N - 1;
        count[N] = 1;
        dist[s] = N;
        active[s] = active[t] = true;

        for (auto& edge : G[s])
        {
            excess[s] += edge.cap;
            Push(edge);
        }

        while (!Q.empty())
        {
            long v = Q.front();
            Q.pop();
            active[v] = false;
            Discharge(v);
        }

        long totflow = 0;

        for (auto& edge : G[s])
            totflow += edge.flow;

        return totflow;
    }

private:
    long N;
    std::vector<std::vector<Edge>> G;
    std::vector<long> excess;
    std::vector<long> dist, active, count;
    std::queue<long> Q;

    void Enqueue(long v)
    {
        if (!active[v] && excess[v] > 0)
        {
            active[v] = true;
            Q.push(v);
        }
    }

    void Push(Edge& e)
```

```
{
    long amt = std::min<long>(excess[e.from], e.cap - e.flow);

    if (dist[e.from] <= dist[e.to] || amt == 0)
        return;

    e.flow += amt;
    G[e.to][e.index_of_twin].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
}

void Gap(long k)
{
    for (long v = 0; v < N; ++v)
    {
        if (dist[v] < k)
            continue;

        --count[dist[v]];
        dist[v] = std::max(dist[v], N + 1);
        ++count[dist[v]];
        Enqueue(v);
    }
}

void Relabel(long v)
{
    --count[dist[v]];
    dist[v] = 2*N;

    for (auto& edge : G[v])
    {
        if (edge.cap - edge.flow > 0)
            dist[v] = std::min(dist[v], dist[edge.to] + 1);
    }

    ++count[dist[v]];
    Enqueue(v);
}

void Discharge(long v)
{
    for (auto& edge : G[v])
```

```
        {
            if (excess[v] <= 0)
                break;
            Push(edge);
        }

        if (excess[v] > 0)
        {
            if (count[dist[v]] == 1)
                Gap(dist[v]);
            else
                Relabel(v);
        }
    }
};

int main()
{
    PushRelabel G(5);

    G.AddEdge(0, 1, 8);
    G.AddEdge(0, 2, 3);
    G.AddEdge(1, 2, 2);
    G.AddEdge(1, 4, 4);
    G.AddEdge(1, 3, 1);
    G.AddEdge(3, 4, 4);

    std::cout << "Max flow: " << G.GetMaxFlow(0, 4) << std::endl;

    return 0;
}
```

**Output**:

```
Max flow: 5
```