

Notas de programación competitiva

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

Última revisión: 19 de octubre de 2023

Índice

1. General	2
2. Tokenización de líneas	2
3. Construcción de cadenas concatenando valores arbitrarios	2
4. Potencia rápida de naturales	2
5. Potencia rápida de naturales con aritmética modular	3
6. Fibonacci matricial en tiempo logarítmico	3
7. Distancia entre dos puntos	3
8. Cerco convexo	3
9. Algoritmos de Dijkstra y Prim	4
10. Algoritmo de Kruskal (unión-pertenencia)	4
11. Algoritmo de Floyd (distancias más cortas entre cualquier pareja de vértices)	5
12. Puentes de una gráfica	5
13. Coeficiente binomial iterativo	5
14. Inverso modular	5
15. Coeficiente binomial modular	6
16. Criba de Eratóstenes	6
17. Primalidad Miller Rabin	6
18. Factorización de números grandes	6
19. Búsqueda binaria generalizada	7
20. Búsqueda de subcadenas en tiempo logarítmico con arreglo de sufijos	7
21. Búsqueda de subcadenas en tiempo lineal con Knuth-Morris-Pratt	8
22. Subsecuencia creciente más larga	8
23. Generación de permutaciones usando la biblioteca	9
24. Subsecuencia común más larga con programación dinámica	9

25.Subsecuencia común más larga con memoria lineal	9
26.Código Lehmer ($i \Leftrightarrow i$ -ésima permutación)	9
27.Generación de cadenas binarias usando recursión	10
28.Generación de cadenas numéricas usando recursión	10
29.Generación de permutaciones usando recursión	10
30.Trie	10
31.Árbol de Fenwick	11
32.Árbol de Fenwick multidimensional	11
33.Árbol de segmentos	12
34.Árbol de segmentos perezoso	13
35.Árbol de segmentos persistente	15
36.Acoplamiento bipartito de cardinalidad máxima	17
37.Acoplamiento bipartito de cardinalidad máxima con costo mínimo	17
38.Flujo máximo	18

1 General

```
#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 (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
ios_base::sync_with_stdio(false);
cin.tie(nullptr);
```

2 Tokenización de líneas

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

```
// también se puede extraer con terminador
// getline(flujo, s, '@'); // separa por @
}
```

3 Construcción de cadenas concatenando valores arbitrarios

```
int main( ) {
    int a = 57;
    char c = '@';
    double f = 3.14;
    ostringstream bufer;
    bufer << a << " " << c << " " << f;
    string cadena = bufer.str( );
    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 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 {
    array<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(array<array<int, 2>, 2> m) : mat(m) {}

    void operator*=(const matriz& m) {
        array<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];
}
```

7 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 sqrt(dx * dx + dy * dy);
}
```

8 Cerco convexo

```
auto producto_cruz(const auto& a, const auto& b, const auto& c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

template<typename RI1, typename RI2>
auto 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) { // < 0 para permitir empates en línea recta
            --bw;
        }
    }
    return bw;
}

template<typename RI>
auto cerco_convexo(RI ai, RI af) {
    if (af - ai <= 2) {
        return vector<typename iterator_traits<RI>::value_type>(ai, af);
    }
    vector<typename iterator_traits<RI>::value_type> res(2 * (af - ai));
    auto iter1 = cerco_parcial(ai, af, res.begin() - 1);
    auto iter2 = cerco_parcial(make_reverse_iterator(af), make_reverse_iterator(ai), iter1) - 1;
    res.resize(iter2 - res.begin());
    return res;
}

auto distancia(const auto& a, const auto& b) {
```

```

    return hypot(a.x - b.x, a.y - b.y); // hypot
    puede ser más lento que hacerlo manualmente
    con sqrt
}

template<typename T>
auto perimetro(const vector<T>& puntos) {
    double res = 0;
    for (int i = 0; i < puntos.size(); ++i) {
        res += distancia(puntos[i], puntos[(i + 1) %
            puntos.size()]);
    }
    return res;
}

struct punto {
    double x, y; // si no necesitan doubles,
    pasarlos a int porque es más rápido
    bool operator<(const punto& p) const {
        return pair(x, y) < pair(p.x, p.y);
    }
};

int main() {
    int n;
    cin >> n;

    vector<punto> v(n);
    for (auto& p : v) {
        cin >> p.x >> p.y;
    }

    sort(v.begin(), v.end()); // importante
    vector<punto> cerco = cerco_convexo(v.begin(),
        v.end());

    for (auto p : cerco) {
        cout << p.x << " " << p.y << "\n";
    }
}

```

9 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;
    cin >> n >> m;

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

```

```

    adyacencia[y].push_back({ x, c });
}

priority_queue<entrada> cp;
cp.push({0, 0});
int distancia[n];
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 });
        } // quitar la suma para Prim (dejar
        vecino.costo)
    }
} while (!cp.empty());

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

```

10 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;
    cin >> v >> a;

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

```

```

    return a.costo < b.costo);
});

int tabla[v];
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);
    }
}
}

```

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

```

12 Puentes de una gráfica

```

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

void dfs(int actual, int anterior, const vector<
int> adyacencia[], vector<bool>& visitado, int
& id, vector<int>& tin, vector<int>& low,
vector<pair<int, int>>& res) {
    visitado[actual] = true, tin[actual] = low[
actual] = id++;
    for (int vecino : adyacencia[actual]) {
        if (vecino != anterior) {
            continue;
        }
        if (visitado[vecino]) {
            low[actual] = min(low[actual], tin[vecino]
);
        } else {
            dfs(vecino, actual, adyacencia, visitado,
id, tin, low, res);
            low[actual] = min(low[actual], low[vecino]
);
            if (low[vecino] > tin[actual]) {
                res.emplace_back(min(actual, vecino),
max(actual, vecino));
            }
        }
    }
}
}

```

```

vector<pair<int, int>> calcula_puentes(const
vector<int> adyacencia[], int n) {
    vector<bool> visitado(n); int id;
    vector<int> tin(n, -1), low(n, -1);
    vector<pair<int, int>> res;
    for (int i = 0; i < n; ++i) {
        dfs(i, -1, adyacencia, visitado, id, tin,
low, res);
    }
    return res;
}

```

```

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

    vector<int> adyacencia[n];
    for (int i = 0; i < m; ++i) {
        int x, y;
        cin >> x >> y;
        adyacencia[x - 1].emplace_back(y - 1);
        adyacencia[y - 1].emplace_back(x - 1);
    }

    auto puentes = calcula_puentes(adyacencia, n);
    for (auto [x, y] : puentes) {
        cout << x << " " << y << "\n";
    }
}

```

13 Coeficiente binomial iterativo

```

unsigned long long binomial(int n, int k) {
    __uint128_t res = 1;
    // unsigned long long es más rápido pero falla
    en casos raros
    for (int i = 1; i <= k; ++i) {
        res *= n + 1 - i;
        res /= i;
    }
    return res;
}

```

14 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;
}

```

15 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);
    }
    //...
}

```

16 Criba de Eratóstenes

```

struct criba {
    int64_t tope;
    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
    cout << c.factor[i]; // un factor de i
    // si i es primo, factor[i] == i
}

```

17 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;
}

```

18 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 = gcd(x - y, n);
        if (d != 1) {
            return d;
        }
        if (i == k) {
            y = x, k *= 2;
        }
    }
}

// usar una criba con un tope razonable
vector<int64_t> factoriza(int64_t n, const criba&
    c) { // suposición: n >= 2
    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 = ceil(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) {
            vector<int64_t> factores1 = factoriza(
checar, c);
            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;
}

```

19 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+1, [](
int checar) {
        return checar >= 50;
    });
}

```

20 Búsqueda de subcadenas en tiempo logarítmico con arreglo de sufijos

```

template<typename RI>
vector<int> suffix_ranking(RI si, RI sf) {
    vector<int> rank(si, sf), indices(sf - si);
    iota(indices.begin( ), indices.end( ), 0);
    for (int t = 1; t <= sf - si; t *= 2) {
        auto pred = [&, rank](int i1, int i2) {
            return make_pair(rank[i1], (i1 + t < sf -
si ? rank[i1 + t] : -1)) < make_pair(rank[i2
], (i2 + t < sf - si ? rank[i2 + t] : -1));
        };
        sort(indices.begin( ), indices.end( ), 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>
vector<RI> suffix_array(RI si, RI sf, const vector
<int>& rank) {
    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>
vector<int> longest_prefix(RI si, RI sf, const
vector<int>& rank, const vector<RI>& suffix) {
    vector<int> res(sf - si);
    for (int i = 0, t = 0; i < rank.size( ); ++i) {
        if (rank[i] + 1 != sf - si) {
            t += 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 vector
<RI1>& suffix, const vector<int>& lcp, RI2 bi,
RI2 bf) {
    auto xi = suffix.begin( ), xf = suffix.end( );
    auto li = lcp.begin( ), lf = lcp.end( );

```



```

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

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

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

```

23 Generación de permutaciones usando la biblioteca

```

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

```

24 Subsecuencia común más larga con programación dinámica

```

auto lcs(const string& a, const 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] = max(mem[i][j + 1], mem[i +
1][j]);
            }
        }
    }

    vector<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;
}

```

25 Subsecuencia común más larga con memoria lineal

```

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

```

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

```

constexpr size_t factorial(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, size_t indice) {
    iota(ini, fin, size_t(0));
    size_t n = fin - ini, r = factorial(n - 1);
    for (T i = ini; i != fin; ++i) {
        size_t d = indice / r; indice %= r;
        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
size_t indice(T ini, T fin) {
    size_t n = fin - ini, r = factorial(n - 1), res
    = 0;
    for (T i = ini; i != fin; ++i) {
        res += r * count_if(i + 1, fin, [&](size_t v
        ) { return v < *i; });
        if (fin - i - 1 != 0) {
            r /= fin - i - 1;
        }
    }
    return res;
}

```

27 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) {
            cout << arr[i];
        }
        cout << "\n";
    } else {
        arr[i] = false;
        cadenas_binarias(i + 1);
        arr[i] = true;
        cadenas_binarias(i + 1);
    }
}

```

28 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) {
            cout << arr[i] << " ";
        }
        cout << "\n";
    } else {
        for (int d = 0; d <= 9; ++d) {
            arr[i] = d;
            cadenas_numericas(i + 1);
        }
    }
}

```

29 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) {
            cout << arr[i] << " ";
        }
        cout << "\n";
    } else {
        for (int j = i; j < n; ++j) {
            swap(arr[i], arr[j]);
            permutaciones(i + 1);
            swap(arr[i], arr[j]);
        }
    }
}

```

30 Trie

```

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

    bool prefijo(const string& s) {

```

```

        auto pos = posicion(s);
        return pos != nullptr;
    }

```

```

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

```

31 Árbol de Fenwick

```

#include <bit> // sólo si se necesita
        min_prefix
#include <iostream>
#include <vector>

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
        para lograr un acumulado >= v;
        int i = 0; // si es
        imposible lograr dicha suma, regresa .size( )
        + 1
        for (int j = 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:
    vector<T> mem_;
};

int main( ) {
    fenwick_tree<int> arbol(10); // inicialmente
    todo en cero

    for (int i = 0; i < 10; ++i) {
        arbol.replace(i, i);
    }
    for (int i = 0; i < 10; ++i) {
        cout << i << ": " << arbol[i] << "\n";
    }
    cout << "\n";

    cout << arbol.query(0, 10) << "\n";
    cout << arbol.query_until(10) << "\n";
    cout << arbol.query(8, 10) << "\n";
    cout << "\n";

    arbol.modify_add(0, +6);
    for (int i = 0; i < 10; ++i) {
        cout << i << ": " << arbol[i] << "\n";
    }
    cout << "\n";

    cout << arbol.query(0, 10) << "\n";
    cout << arbol.query_until(10) << "\n";
    cout << arbol.query(8, 10) << "\n";
    cout << "\n";

    for (int i = 0; i < 10; ++i) {
        cout << arbol.query_until(i + 1) << " ";
    }
    cout << "\n";
    for (int i = 0; i < 100; ++i) {
        cout << i << ": " << arbol.min_prefix(i) <<
            "\n";
    }
}

```

32 Árbol de Fenwick multidimensional

```

#include <array>
#include <bit>
#include <iostream>
#include <tuple>
#include <vector>

```

```

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 operator[](const P&... x) { // C
        ++23
        return query(x..., (x + 1)...);
    }

    template<typename... P>
    T operator()(const P&... x) { // C
        ++20 o menor
        return query(x..., (x + 1)...);
    }

    template<typename... P>
    T query(const P&... x) {
        static_assert(sizeof...(P) == 2 * D);
        return query(make_index_sequence<D>(),
            array<int, 2 * D>{ x... });
    }

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

private:
    template<size_t... I, typename... P>
    T query(index_sequence<I...> i, const array<int,
        2 * D>& indices) {
        T res = 0;
        for (unsigned i = 0; i < (1 << D); ++i) {
            res += (popcount(i) % 2 == D % 2 ? +1 :
                -1) * query_until(indices[bool(i & (1 << I)) *
                    D + I]...);
        }
        return res;
    }

    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 query_until() const {
        return v_;
    }

private:
    T v_ = 0;
};

int main() {
    prueba_1d: {
        fenwick_tree<int, 1> arbol(10);
        arbol.modify_add(2, 1);
        arbol.modify_add(4, 1);

        for (int i = 0; i < 10; ++i) {
            cout << arbol[i] << " ";
        }
        cout << "\n";
        for (int i = 0; i < 10; ++i) {
            cout << arbol.query_until(i + 1) << " ";
        }
        cout << "\n";
    }

    cout << "\n\n";

    prueba_2d: {
        fenwick_tree<int, 2> arbol(10, 10);
        arbol.modify_add(2, 2, 1);
        arbol.modify_add(4, 4, 1);
        for (int i = 0; i < 10; ++i) {
            for (int j = 0; j < 10; ++j) {
                cout << arbol[i, j] << " ";
            }
            cout << "\n";
        }
        cout << "\n\n";
        for (int i = 0; i < 10; ++i) {
            for (int j = 0; j < 10; ++j) {
                cout << arbol.query_until(i + 1, j +
                    1) << " ";
            }
            cout << "\n";
        }
        cout << "\n";
    }
}

template<typename T, typename F = const T&(*)(&const T&, const T&)>
class segment_tree {

```

33 Árbol de segmentos

```

public:
    segment_tree(T n, F f)
    : pisos_(1), neutro_(move(n)), funcion_(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(max(p +
1, int(pisos_.size( ))))) {
            pisos_[p].push_back(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] = 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:
    vector<vector<T>> pisos_;
    F funcion_;
    T neutro_;
};

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

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

```

34 Árbol de segmentos perezoso

```

template<typename T, typename U, typename F1 =
    const T&(*)(const T&, const T&), typename F2 =
    bool(*)(T&, int n, const U&), typename F3 =
    const U&(*)(const U&, const U&)>
class lazy_segment_tree {
    struct nodo {
        T valor;
        U lazy;
    };

public:
    template<typename I>
    lazy_segment_tree(T n1, U n2, F1 f1, F2 f2, F3
f3, I&& entrada, int t)
    : mem_(2 * t), neutro1_(move(n1)), neutro2_(
move(n2)), funcion1_(move(f1)), funcion2_(move
(f2)), funcion3_(move(f3)), tam_(t) {
        construye(0, 0, t, entrada);
    }

    template<typename RI>
    lazy_segment_tree(T n1, U n2, F1 f1, F2 f2, F3
f3, RI ini, RI fin)
    : lazy_segment_tree(move(n1), move(n2), move(f1
), move(f2), move(f3), [&]( ) { return *ini++;
}, fin - ini) {
    }

    int size( ) const {
        return tam_;
    }

    T operator[](int i) const {

```

```

        return query(i, i + 1);
    }

    T query(int ini, int fin) const {
        T res = neutro1_;
        visit(0, ini, fin, 0, tam_, [&](const nodo&
actual, int tam) {
            res = funcion1_(res, actual.valor);
        });
        return res;
    }

    void modify_apply(int ini, int fin, const U& v)
    {
        visit(0, ini, fin, 0, tam_, [&](nodo& actual
, int tam) {
            modifica(actual, tam, v);
        }, true);
    }

    template<typename V>
    void visit(int ini, int fin, V&& vis) const {
        visit(0, ini, fin, 0, tam_, [&](const nodo&
actual, int tam) {
            vis(actual.valor);
        });
    }

private:
    void modifica(nodo& actual, int tam, const U& v
) const {
        if (v != neutro2_ && funcion2_(actual.valor,
tam, v)) {
            actual.lazy = (actual.lazy != neutro2_ ?
funcion3_(actual.lazy, v) : v);
        }
    }

    template<typename I>
    void construye(int i, int ini, int fin, I&
entrada) {
        if (ini == fin) {
            return;
        } else if (fin - ini == 1) {
            mem_[i] = { entrada(), neutro2_ };
        } else {
            int tam = fin - ini, mitad = ini + tam /
2, izq = i + 1, der = i + 2 * (tam / 2);
            construye(izq, ini, mitad, entrada);
            construye(der, mitad, fin, entrada);
            mem_[i] = { funcion1_(mem_[izq].valor,
mem_[der].valor), neutro2_ };
        }
    }

    template<typename V>
    void visit(int i, int qi, int qf, int ini, int
fin, V&& vis, bool actualizar = false) const {
        if (qi >= qf) {
            return;
        } else if (qi == ini && qf == fin) {
            vis(mem_[i], fin - ini);
        } else {
            int tam = fin - ini, mitad = ini + tam /
2, izq = i + 1, der = i + 2 * (tam / 2);
            modifica(mem_[izq], tam / 2, mem_[i].lazy
);
            modifica(mem_[der], tam - tam / 2, mem_[i
].lazy);
            mem_[i].lazy = neutro2_;

            visit(izq, qi, min(qf, mitad), ini, mitad
, vis, actualizar);
            visit(der, max(qi, mitad), qf, mitad, fin
, vis, actualizar);
            if (actualizar) {
                mem_[i].valor = funcion1_(mem_[izq].
valor, mem_[der].valor);
            }
        }
    }

    mutable vector<nodo> mem_;
    T neutro1_;
    U neutro2_;
    F1 funcion1_;
    F2 funcion2_;
    F3 funcion3_;
    int tam_;
};

int main( ) {
    auto imprime = [&](const auto& s) {
        for (int i = 0; i < s.size( ); ++i) {
            cout << s[i] << " ";
        }
        cout << "\n";
    };

    caso1: { // query: suma, update: suma
        auto s = lazy_segment_tree(
            0,
            0,
            plus( ),
            [](int& a, int n, int b) { return a += n
* b, true; },
            plus( ),
            [i = 0]( ) mutable {
                return i++;
            }, 50);
        imprime(s);

        cout << "original...\n";
        int arr[s.size( )];
        iota(arr, arr + s.size( ), 0);
        for (int i = 0; i <= s.size( ); ++i) {
            for (int f = i; f <= s.size( ); ++f) {
                if (accumulate(arr + i, arr + f, 0) !=
s.query(i, f)) {
                    cout << "X_X " << i << " " << f <<
"\n";
                }
            }
        }
        return 0;
    }
}

```

```

    }
}

cout << "[0,50)...\n";
for_each(arr, arr + 50, [](int& v) { v += 10; });
s.modify_apply(0, 50, 10);
for (int i = 0; i <= s.size(); ++i) {
    for (int f = i; f <= s.size(); ++f) {
        if (accumulate(arr + i, arr + f, 0) != s.query(i, f)) {
            cout << "X_X " << i << " " << f << "\n";
            return 0;
        }
    }
}

cout << "[10,40)...\n";
for_each(arr + 10, arr + 40, [](int& v) { v += 312; });
s.modify_apply(10, 40, 312);
for (int i = 0; i <= s.size(); ++i) {
    for (int f = i; f <= s.size(); ++f) {
        if (accumulate(arr + i, arr + f, 0) != s.query(i, f)) {
            cout << "X_X\n";
            return 0;
        }
    }
}

cout << ":\n";
}

caso2: { // query: max, update: asignación
    auto s = lazy_segment_tree(
        INT_MIN,
        optional<int>( ),
        max,
        [](int& a, int n, optional<int> b) {
            return a = *b, true; },
        [](optional<int> a, optional<int> b) {
            return b; },
        [i = 0]( ) mutable {
            return i++;
        }, 50);
    imprime(s);

    cout << "original...\n";
    int arr[s.size( )];
    iota(arr, arr + s.size( ), 0);
    for (int i = 0; i <= s.size( ); ++i) {
        for (int f = i; f <= s.size( ); ++f) {
            if (i != f && *max_element(arr + i, arr + f) != s.query(i, f)) {
                cout << "X_X " << i << " " << f << "\n";
                return 0;
            }
        }
    }
}

```

```

    }
}

cout << "[0,50)...\n";
for_each(arr, arr + 50, [](int& v) { v = 10; });
s.modify_apply(0, 50, 10);
for (int i = 0; i <= s.size( ); ++i) {
    for (int f = i; f <= s.size( ); ++f) {
        if (i != f && *max_element(arr + i, arr + f) != s.query(i, f)) {
            cout << "X_X " << i << " " << f << "\n";
            return 0;
        }
    }
}

cout << "[10,40)...\n";
for_each(arr + 10, arr + 40, [](int& v) { v = 312; });
s.modify_apply(10, 40, 312);
for (int i = 0; i <= s.size( ); ++i) {
    for (int f = i; f <= s.size( ); ++f) {
        if (i != f && *max_element(arr + i, arr + f) != s.query(i, f)) {
            cout << "X_X " << i << " " << f << "\n";
            return 0;
        }
    }
}

cout << ":\n";
}
}

```

35 Árbol de segmentos persistente

```

template<typename T, typename F = const T&(*)(<
    const T&, const T&>)
class persistent_segment_tree {
    struct nodo {
        T valor;
        nodo *izq, *der;
    };

public:
    template<typename I>
    persistent_segment_tree(T n, F f, I&& entrada,
        int t)
        : mem_(make_shared<deque<nodo>>()), neutro_(
            move(n)), funcion_(move(f)), tam_(t), raiz_(
                replace(nullptr, 0, tam_, 0, tam_, entrada)) {}

    template<typename RI>
    persistent_segment_tree(T n, F f, RI ini, RI
        fin)

```

```

: persistent_segment_tree(move(n), move(f),
 [&]( ) { return *ini++; }, fin - ini) {
}

int size( ) const {
    return tam_;
}

const T& operator[](int i) const {
    const T* res;
    visit(i, i + 1, [&](const T& actual) {
        res = &actual;
    });
    return *res;
}

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

persistent_segment_tree replace(int i, T v) {
    return { neutro_, funcion_, tam_, replace(
        raiz_, i, i + 1, 0, tam_, [&]( ) { return move
        (v); } ), mem_ };
}

template<typename V>
void visit(int ini, int fin, V&& vis) const {
    return visit(raiz_, ini, fin, 0, tam_, vis);
}

private:
persistent_segment_tree(T n, F f, int t, nodo*
r, shared_ptr<deque<nodo>>& m)
: mem_(m), neutro_(move(n)), funcion_(move(f)),
raiz_(r), tam_(t) {
}

template<typename I>
nodo* replace(nodo* p, int qi, int qf, int ini,
int fin, I&& entrada) {
    if (ini == fin || qi >= qf) {
        return p;
    } else if (fin - ini == 1) {
        return crea(entrada( ));
    } else {
        int mitad = ini + (fin - ini) / 2, tam =
fin - ini;
        auto izq = replace((p == nullptr ?
nullptr : p->izq), qi, min(qf, mitad), ini,
mitad, entrada);
        auto der = replace((p == nullptr ?
nullptr : p->der), max(qi, mitad), qf, mitad,
fin, entrada);
        return crea(funcion_(izq->valor, der->
valor), izq, der);
    }
}

}

template<typename V>
void visit(const nodo* p, int qi, int qf, int
ini, int fin, V& vis) const {
    if (qi >= qf) {
        return;
    } else if (qi == ini && qf == fin) {
        vis(p->valor);
    } else {
        int mitad = ini + (fin - ini) / 2;
        visit(p->izq, qi, min(qf, mitad), ini,
mitad, vis);
        visit(p->der, max(qi, mitad), qf, mitad,
fin, vis);
    }
}

template<typename... P>
nodo* crea(P&&... v) {
    return &*mem_->insert(mem_->end( ), nodo{
        forward<P>(v)...});
}

shared_ptr<deque<nodo>> mem_;
T neutro_;
F funcion_;
int tam_;
nodo* raiz_;
};

int main( ) {
    auto s1 = persistent_segment_tree(0, plus( ), [
        i = 0]( ) mutable {
            return i++;
        }, 50);
    auto s2 = s1.replace(13, 27);
    auto s3 = s2.replace(42, -14);

    auto imprime = [&](const auto& s) {
        for (int i = 0; i < s.size( ); ++i) {
            cout << s[i] << " ";
        }
        cout << "\n";
    };
    imprime(s1), imprime(s2), imprime(s3);

    int arr[s1.size( )];
    iota(arr, arr + s1.size( ), 0);
    for (int i = 0; i <= s1.size( ); ++i) {
        for (int f = i; f <= s1.size( ); ++f) {
            if (accumulate(arr + i, arr + f, 0) != s1
                .query(i, f)) {
                cout << "X_X\n";
            }
        }
    }

    arr[13] = 27;
    for (int i = 0; i <= s2.size( ); ++i) {
        for (int f = i; f <= s2.size( ); ++f) {

```



```

        if (accumulate(arr + i, arr + f, 0) != s2
        .query(i, f)) {
            cout << "X_X\n";
        }
    }
}

arr[42] = -14;
for (int i = 0; i <= s3.size(); ++i) {
    for (int f = i; f <= s3.size(); ++f) {
        if (accumulate(arr + i, arr + f, 0) != s3
        .query(i, f)) {
            cout << "X_X\n";
        }
    }
}

cout << ":\n";
}

```

36 Acoplamiento bipartito de cardinalidad máxima

```

namespace bipartito {
    bool aumenta(int a, vector<int> adyacencia_a[],
        vector<int>& nivel_a, vector<int>& pareja_a,
        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(vector<int> adyacencia_a[], int n,
    int m) {
    vector<int> nivel_a(n), pareja_a(n, -1),
    pareja_b(m, -1);
    for (;;) {
        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 - count(pareja_a.
        begin(), pareja_a.end(), -1);
        return tuple(move(pareja_a), 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
}

```

37 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

typedef vector<double> VD; // el algoritmo usa
double para los costos durante el cálculo
typedef vector<VD> VVD; // vector<vector<
double>> como matriz de adyacencia para los
costos
typedef vector<int> VI; // 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;
}

```

38 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).

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;
    cin >> n >> e;
    Dinic dinic(n);

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

```