Referencia - ICPC

Mathgic

Marzo 2023

Github
ACTUALIZADO HASTA LA SECCIÓN 9. Falta 10, 11, etc.

Índice

1. Estructuras básicas

1.1. Min stack

```cpp
struct min_stack{
    stack<pair<int, int>> st;
    min_stack(){st.push(make_pair(INT_MAX, INT_MAX));}
    void push(int v){st.push(make_pair(v, min(v, st.top().second)));}
    int top(){return st.top().first;}
    void pop(){if(st.size() > 1)st.pop();}
    int minV(){return st.top().second;}
    int size(){return st.size() -1;}
    bool empty(){return size() == 0;}
};
```

1.2. Min queue

```cpp
struct min_queue{
    min_stack p_in;
    min_stack p_out;
    void push(int v){p_in.push(v);}
    int front(){transfer(); return p_out.top();}
    void pop(){transfer(); p_out.pop();}
    int size(){return p_in.size() + p_out.size();}
    int minV() {return min(p_in.minV(), p_out.minV());}
    bool empty(){ return size() == 0;}
    void transfer(){
        if(p_out.size()) return;
        while(p_in.size()){
            p_out.push(p_in.top());
            p_in.pop();
        }
    }
};
```

2. Ordenamiento

2.1. Bucket sort

Complejidad: Tiempo $O(n)$ - Memoria extra $O(\texttt{MAXVAL})$.
`MAXVAL` es el valor máximo del arreglo.

```cpp
void bucketSort(int arr[], int n){
    int cub[MAXVAL + 1] = {};
    for(int i = 0; i < n; ++i)
        cub[ arr[i] ]++;
    int idx = 0;
    for(int i = 0; i <= MAXVAL; ++i){
        while( cub[i] ){
            arr[idx] = i;
            cub[i]--;
            idx++;
        }
    }
}
```

2.2. Merge sort

Complejidad: Tiempo $O(n \log n)$ - Memoria extra $O(n)$.

```cpp
void mergeSort(int arr[], int ini, int fin){
    if(ini == fin) return;
    int mitad = (ini + fin) / 2;
    mergeSort(arr, ini, mitad);
    mergeSort(arr, mitad + 1, fin);

    int tam1 = mitad - ini + 1, tam2 = fin - mitad;
    int mitad1[tam1], mitad2[tam2];
    for(int i = ini, idx = 0; i <= mitad; ++i, idx++)
        mitad1[idx] = arr[i];
    for(int i = mitad + 1, idx = 0; i <= fin; ++i, idx++)
        mitad2[idx] = arr[i];

    for(int i = ini, idx1 = 0, idx2 = 0; i <= fin; ++i){
        if(idx1 < tam1 && idx2 < tam2){ /// si quedan elementos en ambas mitades
            arr[i] = mitad1[idx1] < mitad2[idx2] ? mitad1[idx1++] : mitad2[idx2++];
        } else if(idx1 < tam1){ /// si solo hay elementos en mitad1
            arr[i] = idx1 < tam1 ? mitad1[idx1++] : mitad2[idx2++];
        }
    }
}
```

3. Matemáticas

3.1. Criba de Eratóstenes

Complejidad: Tiempo $O(n \log \log n)$ - Memoria extra $O(n)$.
Calcula los primos menores o iguales a n.

```cpp
void criba(int n, vector<int> &primos){
    primos.clear();
    if(n < 2) return;
    vector<bool> no_primo(n + 1);
    no_primo[0] = no_primo[1] = true;
    for(long long i = 3; i * i <= n; i += 2){
        if(no_primo[i]) continue;
        for(long long j = i * i; j <= n; j += 2 * i)
            no_primo[j] = true;
    }
    primos.push_back(2);
    for(int i = 3; i <= n; i += 2){
        if(!no_primo[i])
            primos.push_back(i);
    }
}
```

3.2. Criba sobre un rango

Complejidad: Tiempo $O(\sqrt{b} \log \log \sqrt{b} + (b-a) \log \log (b-a))$ - Memoria extra $O(\sqrt{b} + b - a)$.
Calcula los primos en el rango $[a, b]$.

```cpp
void cribaSobreRango(long long a, long long b, vector<long long> &primos){
    a = max(a, 0ll);
    b = max(b, 0ll);
    long long tam = b - a + 1;
    vector<int> primosRaiz;
    criba(sqrt(b) + 1, primosRaiz);
    bool no_primo[tam] = {};
    primos.clear();
    for(long long p : primosRaiz){
        long long ini = p * max(p, (a + p - 1) / p);
        for(long long m = ini; m <= b; m += p){
            no_primo[m - a] = true;
        }
    }
    for(long long i = 0; i < tam; ++i){
        if(no_primo[i] || i + a < 2) continue;
        primos.push_back(i + a);
    }
}
```

3.3. Criba segmentada

Complejidad: Tiempo $O(\sqrt{n}\log\log\sqrt{n} + n\log\log n)$ - Memoria extra $O(\sqrt{n} + S)$.
Cuenta la cantidad de primos menores o iguales a n.

```cpp
int cuentaPrimos(int n){
    if(n < 2) return 0;
    const int S = sqrt(n);
    vector<int> primosRaiz;
    criba(sqrt(n) + 1, primosRaiz);
    int ans = 0;
    vector<char> no_primo(S);
    for(int ini = 0; ini <= n; ini += S){
        fill(no_primo.begin(), no_primo.end(), false);
        for(int p : primosRaiz){
            int m = p * max(p, (ini + p - 1) / p) - ini;
            for(; m <= S; m += p)
                no_primo[m] = true;
        }
        for(int i = 0; i < S && i + ini <= n; ++i)
            if(!no_primo[i] && 1 < i + ini)
                ans++;
    }
    return ans;
}
```

3.4. Criba lineal

Complejidad: Tiempo $O(n)$ - Memoria extra $O(n)$.
Calcula los primos menores o iguales a n y el menor primo que divide a cada entero en $[2, n]$.

```cpp
void cribaLineal(int n, vector<int> &primos){
    primos.clear();
    if(n < 2) return;
    vector<int> lp(n + 1);
    for(long long i = 2; i <= n; ++i){
        if(!lp[i]){
            lp[i] = i;
```

```
            primos.push_back(i);
        }
        for(int j = 0; i * (long long)primos[j] <= n; ++j){
            lp[i * primos[j]] = primos[j];
            if(primos[j] == lp[i])
                break;
        }
    }
}
```

3.5. Algoritmo extendido de Euclides

Complejidad: Tiempo $O(\log(\text{máx}(a,b)))$ - Memoria extra $O(1)$.
Encuentra una solución a la ecuación $ax + by = \gcd(a,b)$.

```
int gcdExtendido(int a, int b, int &x, int &y){
    if(!b){
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int g = gcdExtendido(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return g;
}
```

3.6. Solución de ecuaciones diofánticas lineales

Complejidad: Tiempo $O(\log(\text{máx}(a,b)))$ - Memoria extra $O(1)$.
Encuentra una solución a la ecuación $ax + by = c$ o determina si no existe solución.

```
bool encuentra_solucion(int a, int b, int c, int &x, int &y){
    int g = gcdExtendido(abs(a), abs(b), x, y);
    if(c % g) return false;
    x *= c / g;
    y *= c / g;
    if(a < 0) x = -x;
    if(b < 0) y = -y;
    return true;
}
```

Cambia a la siguiente (anterior) solución abs(cnt) veces. $g := gcd(a,b)$.

```
void cambia_solucion(int &x, int &y, int a, int b, int cnt, int g = 1) {
    x += cnt * b / g;
    y -= cnt * a / g;
}
```

Cuenta la cantidad de soluciones x, y con $x \in [minx, maxx]$ y $y \in [miny, maxy]$.

```
int cuenta_soluciones(int a, int b, int c, int minx, int maxx, int miny, int maxy) {
    int x, y, g;
    if(!encuentra_solucion(a, b, c, x, y, g)) return 0;
    /// ax + by = c ssi (a/g)x + (b/g)y = c/g
    /// Dividimos entre g para simplificar y no dividir a cada rato
    a /= g;
    b /= g;
```

```
    /// Signos de a, b nos sirven para pasar a la
    /// siguiente (anterior) solucion
    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;
    /// pasa a la minima solucion tal que minx <= x
    cambia_solucion(x, y, a, b, (minx - x) / b);
    /// si x < minx, pasa a la siguiente para que minx <= x
    if(x < minx) cambia_solucion(x, y, a, b, sign_b);
    if(x > maxx) return 0; /// si x > maxx, entonces no hay x solucion tal que x in [minx, maxx]
    int lx1 = x;
    /// pasa a la maxima solucion tal que x <= maxx
    cambia_solucion(x, y, a, b, (maxx - x) / b);
    if(x > maxx) cambia_solucion(x, y, a, b, -sign_b); /// si x > maxx, pasa a la solucion anterior
    int rx1 = x;
    /// hace todo lo anterior pero con y
    cambia_solucion(x, y, a, b, -(miny - y) / a);
    if(y < miny) cambia_solucion(x, y, a, b, -sign_a);
    if(y > maxy) return 0;
    int lx2 = x;
    cambia_solucion(x, y, a, b, -(maxy - y) / a);
    if(y > maxy) cambia_solucion(x, y, a, b, sign_a);
    int rx2 = x;
    /// como al encontrar las x tomando y como criterio no nos asegura
    /// que esten ordenadas, entonces las ordenamos
    if(lx2 > rx2) swap(lx2, rx2);
    /// obtenemos la interseccion de los intervalos
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);
    if(lx > rx) return 0; /// no existen soluciones, interseccion vacia
    /// las soluciones (por x) van de b en b (b/g en b/g pero dividimos al principio)
    return (rx - lx) / abs(b) + 1;
}
```

3.7. Función Phi de Euler

Complejidad: Tiempo $O(d)$ - Memoria extra $O(n)$. d es la cantidad de factores primos de n. `lp[i]` es el menor primo que divide a `i`.

Cuenta la cantidad de coprimos con n menores a n.

```
int phi(int n){
    if(n <= 1) return 1;
    if(!dp[n]){
        int pot = 1, p = lp[n], n0 = n;
        while(n0 % p == 0){
            pot *= p;
            n0 /= p;
        }
        pot /= p;
        dp[n] = pot * (p - 1) * phi(n0);
    }
    return dp[n];
}
```

3.8. Función sigma

Sigma 0 (σ_0). Complejidad: Tiempo $O(d)$ - Memoria extra $O(n)$. d es la cantidad de factores primos de n. `lp[i]` es el menor primo que divide a `i`.

Cuenta la cantidad de divisores de n.

```
long long sigma0(int n){
    if(n <= 1) return 1;
    if(!dp[n]){
        long long exp = 0, p = lp[n], n0 = n;
        while(n0 % p == 0){
            exp++;
            n0 /= p;
        }
        dp[n] = (exp + 1) * sigma0(n0);
    }
    return dp[n];
}
```

Sigma 1 (σ_1). Complejidad: Tiempo $O(d)$ - Memoria extra $O(n)$. d es la cantidad de factores primos de n. `lp[i]` es el menor primo que divide a `i`.

Calcula la suma de los divisores de n.

```
long long sigma1(int n){
    if(n <= 1) return 1;
    if(!dp[n]){
        long long pot = 1, p = lp[n], n0 = n;
        while(n0 % p == 0){
            pot *= p;
            n0 /= p;
        }
        pot *= p;
        dp[n] = (pot - 1) / (p - 1) * sigma1(n0);
    }
    return dp[n];
}
```

3.9. Función de Moebius

Complejidad: Tiempo $O(d)$ - Memoria extra $O(n)$. d es la cantidad de factores primos de n. `lp[i]` es el menor primo que divide a `i`.

Devuelve 0 si n no es divisible por algún cuadrado. Devuelve 1 o -1 si n es divisible por al menos un cuadrado. Devuelve 1 si n tiene una cantidad par de factores primos. Devuelve -1 si n tiene una cantidad impar de factores primos.

```
int moebius(int n){
    if(n <= 1) return 1;
    if(dp[n] == -7){
        int exp = 0, p = lp[n], n0 = n;
        while(n0 % p == 0){
            exp++;
            n0 /= p;
        }
        dp[n] = (exp > 1 ? 0 : -1 * moebius(n0));
    }
    return dp[n];
}
```

3.10. Exponenciación binaria

Iterativa. Complejidad: Tiempo $O(\log b)$ - Memoria extra $O(1)$.

```cpp
int binExp(int a, int b){
    int ans = 1;
    while(b){
        if(b % 2) ans *= a;
        a *= a;
        b /= 2;
    }
    return ans;
}
```

Recursiva. Complejidad: Tiempo $O(\log b)$ - Memoria extra $O(1)$.

```cpp
int binExp(int a, int b){
    if(!b) return 1;
    int tmp = binExp(a, b / 2);
    if(b % 2) return tmp * tmp * a;
    return tmp * tmp;
}
```

4. Sparse table

Complejidad: Tiempo de precalculo $O(n \log n)$ - Tiempo en responder $O(\log(r - l + 1))$ - Tiempo en responder para operaciones idempotentes $O(1)$ - Memoria extra $O(n \log n)$. `LOGN` es $\lceil \log_2(\texttt{MAXN}) \rceil$.

```cpp
struct sparse_table{
    int n, NEUTRO, ST[LOGN][MAXN], lg2[LOGN];
    int f(int a, int b){
        return a + b;
    }
    sparse_table(int n_size, int nums[]){
        n = n_size;
        NEUTRO = lg2[1] = 0;
        for(int i = 2; i <= n; ++i)
            lg2[i] = lg2[i / 2] + 1;
        for(int i = 0; i < n; ++i)
            ST[0][i] = nums[i];
        for(int k = 1; k <= lg2[n]; ++k){
            int fin = (1 << k) - 1;
            for(int i = 0; i + fin < n; ++i)
                ST[k][i] = f(ST[k - 1][i], ST[k - 1][i + (1 << (k - 1))]);
        }
    }
    int query(int l, int r){
        int ans = NEUTRO;
        for(int k = lg2[n]; 0 <= k; --k){
            if( r - l + 1 < (1 << k) ) continue;
            ans = f(ans, ST[k][l]);
            l += 1 << k;
        }
        return ans;
    }
    int queryIdem(int l, int r){
        int lg = lg2[r - l + 1];
        return f(ST[lg][l], ST[lg][r - (1 << lg) + 1]);
    }
};
```

5. Fenwick Tree

Complejidad: Tiempo en responder $O(\log n)$ - Tiempo de actualización $O(\log n)$ - Memoria extra $O(n)$.

```
struct FenwickTree{
    int n, BIT[MAXN];
    FenwickTree(int n_size){
        n = n_size;
        memset(BIT, 0, sizeof(BIT));
    }
    void add(int pos, int x){
        while(pos <= n){
            BIT[pos] += x;
            pos += pos & -pos;
        }
    }
    int sum(int pos){
        int ret = 0;
        while(pos){
            ret += BIT[pos];
            pos -= pos & -pos;
        }
        return ret;
    }
};
```

6. Segment Tree

Nodo del Segment Tree:

```
struct nodo{
    int val, lazy;
    nodo():val(0), lazy(0){}/// inicializa con el neutro y sin lazy pendiente
    nodo(int x, int lz = 0):val(x), lazy(lz){}
    const nodo operator+(const nodo &b)const{
        return nodo(val + b.val);
    }
}
```

6.1. Actualizaciones puntuales

Complejidad: Tiempo de precalculo $O(n)$ - Tiempo en responder $O(\log n)$ - Tiempo de actualización $O(\log n)$ - Memoria extra $O(n)$.

```
struct segment_tree{
    struct node{...}nodes[4 * MAXN + 1];
    segment_tree(int n, int data[]){
        build(1, n, data);
    }
    void build(int left, int right, int data[], int pos = 1){
        if(left == right){
            nodes[pos].val = data[left];
            return;
        }
        int mid = (left + right) / 2;
        build(left, mid, data, pos * 2);
```

```
        build(mid + 1, right, data, pos * 2 + 1);
        nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
    }
    void update(int x, int idx, int left, int right, int pos = 1){
        if(idx < left || right < idx) return;
        if(left == right){
            nodes[pos].val += x;
            return;
        }
        int mid = (left + right) / 2;
        update(x, idx, left, mid, pos * 2);
        update(x, idx, mid + 1, right, pos * 2 + 1);
        nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
    }
    node query(int l, int r, int left, int right, int pos = 1){
        if(r < left || right < l) return node(); /// Devuelve el neutro
        if(l <= left && right <= r) return nodes[pos];
        int mid = (left + right) / 2;
        return query(l, r, left, mid, pos * 2) + query(l, r, mid + 1, right, pos * 2 + 1);
    }
};
```

6.2. Actualizaciones sobre rangos

Complejidad: Tiempo de precalculo $O(n)$ - Tiempo en responder $O(\log n)$ - Tiempo de actualización $O(\log n)$ - Memoria extra $O(n)$.

```
struct segment_tree{
    struct node{...}nodes[4 * MAXN + 1];
    segment_tree(int n, int data[]){...}
    void build(int left, int right, int data[], int pos = 1){...}
    void combineLazy(int lz, int pos){nodes[pos].lazy += lz;}
    void applyLazy(int pos, int tam){
        nodes[pos].val += nodes[pos].lazy * tam;
        nodes[pos].lazy = 0;
    }
    void pushLazy(int pos, int left, int right){
        int tam = abs(right - left + 1);
        if(1 < tam){
            combineLazy(nodes[pos].lazy, pos * 2);
            combineLazy(nodes[pos].lazy, pos * 2 + 1);
        }
        applyLazy(pos, tam);
    }
    void update(int x, int l, int r, int left, int right, int pos = 1){
        pushLazy(pos, left, right);
        if(r < left || right < l) return;
        if(l <= left && right <= r){
            combineLazy(x, pos);
            pushLazy(pos, left, right);
            return;
        }
        int mid = (left + right) / 2;
        update(x, l, r, left, mid, pos * 2);
        update(x, l, r, mid + 1, right, pos * 2 + 1);
        nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
    }
```

```
    node query(int l, int r, int left, int right, int pos = 1){...}
};
```

7. Sqrt decomposition

7.1. Algoritmo de MO

Complejidad: Tiempo en responder $O\left((n+q)\sqrt{n}F + q\log(q)\right)$, donde $O(F)$ es la complejidad de `add()` y `remove()`.

```
const int block_size = 300; /// Ajustable
struct query {
    int l, r, block, i;
    bool operator<(const query &b) const {
        if(block == b.block) return r < b.r;
        return block < b.block;
    }
};
void add(int idx){/**TO-DO*/}
void remove(int idx){/**TO-DO*/}
int get_answer(){return 0; /**TO-DO*/}
vector<int> solve(vector<query> &queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());
    int l_act = 0;
    int r_act = -1;
    for(query q : queries){
        while(l_act > q.l) add(--l_act);
        while(r_act < q.r) add(++r_act);
        while(l_act < q.l) remove(l_act++);
        while(r_act > q.r) remove(r_act--);
        answers[q.i] = get_answer();
    }
    return answers;
}
```

8. DSU

Complejidad: Tiempo $O(\log(n))$ - Memoria $O(n)$, donde n es la cantidad total de elementos. La complejidad temporal es por cada función.

`P[MAXN]`: guarda el representante para cada nodo.

`RA[MAXN]`: guarda el rango (peso) del conjunto de cada representante para el *small to large*.

```
struct dsu{
    int RA[MAXN], P[MAXN];
    dsu(int n){
        for(int i = 0; i < n; ++i){
            RA[i] = 1;
            P[i] = i;
        }
    }
    int root(int x){
        if(x == P[x]) return x;
```

```
        return P[x] = root(P[x]);
    }
    void join(int x, int y){
        int Px = root(x);
        int Py = root(y);
        if(Px == Py) return;
        if(RA[Px] < RA[Py]){
            RA[Py] += RA[Py];
            P[Px] = Py;
        } else {
            RA[Px] += RA[Py];
            P[Py] = Px;
        }
    }
};
```

9. Grafos

```
struct edge{
    int from, to;
    int64_t w;
    const bool operator<(const edge &b)const{
        return w > b.w;
    }
};
struct pos{
    int from;
    int64_t c;
    const bool operator<(const pos &b)const{
        return c > b.c;
    }
};
```

9.1. Caminos mínimos

9.1.1. Dijkstra

Complejidad: Tiempo $O(|E| \log |V|)$ - Memoria extra $O(|E|)$. `dist[MAXN]` es el arreglo de distancias mínimas desde el nodo inicial a todos los demás.

```
int64_t dijkstra(int a, int b, vector<edge> graph[]){
    int64_t dist[MAXN];
    bool vis[MAXN];
    fill(dist, dist + MAXN, LLONG_MAX);
    memset(vis, 0, sizeof(vis));
    priority_queue<pos> q;
    q.push(pos{a, 0});
    dist[a] = 0;
    while(!q.empty()){
        pos act = q.top();
        q.pop();
        if(vis[act.from]) continue;
        vis[act.from] = true;
        for(edge &e : grafo[act.from]){
            if(dist[e.to] < dist[act.from] + e.w) continue;
```

```
            dist[e.to] = dist[act.from] + e.w;
            q.push(pos{e.to, dist[e.to]});
        }
    }
    return dist[b];
}
```

9.1.2. Bellman-Ford

```
vector<int> bellman_ford(int s, int n, vector<edge> &edges, bool cycles = false){
    vector<int> d(n, (cycles ? 0 : INT_MAX));
    d[s] = 0;
    vector<int> P(n, -1); /// Predecesor
    for(int i = 0; i < n - 1; ++i){
        for(edge &e : edges){
            if(d[e.from] == INT_MAX) continue;
            if(d[e.to] > d[e.from] + e.w){
                d[e.to] = d[e.from] + e.w;
                P[e.to] = e.from;
            }
        }
    }
    int last_relax = -1;
    for(edge &e : edges){
        if(d[e.from] == INT_MAX) continue;
        if(d[e.to] > d[e.from] + e.w){
            d[e.to] = d[e.from] + e.w;
            P[e.to] = e.from;
            last_relax = e.to;
        }
    }
    if(last_relax == -1) return d;
    return {}; /// VACIO
}
```

9.2. Árboles

9.2.1. MST

Prim. Complejidad: Tiempo $O(|E| \log |V|)$ - Memoria extra $O(|E|)$. `eCost[MAXN]` es el arreglo de costos mínimos de cada nodo para incluirlo en el MST.

```
int64_t prim(vector<edge> grafo[]){
    int64_t eCost[MAXN];
    bool vis[MAXN];
    memset(vis, 0, sizeof(vis));
    fill(eCost, eCost + MAXN, LLONG_MAX);
    int64_t ans = 0;
    priority_queue<edge> q;
    q.push(edge{1, 1, 0});
    while(q.size()){
        int node = q.top().to;
        int64_t w = q.top().w;
        q.pop();
        if(vis[node]) continue;
        visitado[node] = true;
        ans += w;
        for(edge &it : grafo[node]){
```

```
            if(visitado[it.to] || eCost[it.to] <= it.w) continue;
            eCost[it.to] = it.w;
            q.push(it);
        }
    }
    return ans;
}
```

9.2.2. LCA

9.2.3. Sack

Complejidad: Tiempo $O(n \log n)$.

```
vector<int> tree[MAXN];
int subtree_size[MAXN], depth[MAXN];
bool big[MAXN];
void precalc(int node, int p = 0){
    subtree_size[node] = 1;
    depth[node] = depth[p] + 1;
    for(int v : tree[node]){
        if(v == p) continue;
        precalc(v, node);
        subtree_size[node] += subtree_size[v];
    }
}
void add(int node, int x, int p = 0){
    /// add node here
    /// add subtree
    for(int v: tree[node])
        if(v != p && !big[v])
            add(v, x, node);
}
void dfs(int node, bool keep, int p = 0){
    int maxi = -1, big_child = -1;
    for(int v : tree[node]) /// Search for big_child
       if(v != p && subtree_size[v] > maxi)
          maxi = subtree_size[v], big_child = v;
    for(int v : tree[node])
        if(v != p && v != big_child)
            dfs(v, false, node);  /// run a dfs on small childs and clear them
    if(big_child != -1)
        dfs(big_child, true, node), big[big_child] = 1;  /// big_child marked as big and not cleared
    add(node, 1, p);
    /// answer queries here
    if(big_child != -1) big[big_child] = 0;
    if(!keep) add(node, -1, p);
}
```

9.3. Máximo flujo

Complejidad: Ford-Fulkerson $O(|E| \cdot maxFlow)$, Edmonds-Karp $\left(|V||E|^2\right)$.

```
struct edge {
    int64_t c; // capacity
    int64_t f; // flow
    int to;
};
```

```cpp
class ford_fulkerson {
public:
    ford_fulkerson (vector<vector<pair<int, int64_t>>> &graph) : graph(graph){}
    int64_t get_max_flow(int s, int t){
        init();
        int64_t f = 0;
        while(find_and_update(s, t, f)){}
        return f;
    }
private:
    vector<vector<pair<int, int64_t>>> graph; // graph (to, capacity)
    vector<edge> edges; // List of edges (including the inverse ones)
    vector<vector<int>> edge_indexes; // indexes of edges going out from each vertex
    void init(){
        edges.clear();
        edge_indexes.clear(); edge_indexes.resize(graph.size());
        for(int i = 0; i < graph.size(); i++){
            for(int j = 0; j < graph[i].size(); j++){
                edges.push_back({graph[i][j].second, 0, graph[i][j].first});
                edges.push_back({0, 0, i});
                edge_indexes[i].push_back(edges.size() - 2);
                edge_indexes[graph[i][j].first].push_back(edges.size() - 1);
            }
        }
}
    bool find_and_update(int s, int t, int64_t &flow){
        // Encontrar camino desat con BFS
        queue<int> q;
        // Desde donde llego y con que arista
        vector<pair<int, int>> from(graph.size(), make_pair(-1, -1));
        q.push(s);
        from[s] = make_pair(s, -1);
        bool found = false;
        while(q.size() && (!found)){
            int u = q.front(); q.pop();
            for(int i = 0; i < edge_indexes[u].size(); i++){
                int eI = edge_indexes[u][i];
                if((edges[eI].c > edges[eI].f) && (from[edges[eI].to].first == -1)){
                    from[edges[eI].to] = make_pair(u, eI);
                    q.push(edges[eI].to);
                    if(edges[eI].to == t) found = true;
                }
            }
        }
        if(!found) return false;
        // Encontrar cap. minima del camino de aumento
        int64_t u_flow = LLONG_MAX;
        int current = t;
        while(current != s) {
            u_flow = min(u_flow, edges[from[current].second].c - edges[from[current].second].f);
            current = from[current].first;
        }
        current = t;
        // Actualizar flujo
        while(current != s){
            edges[from[current].second].f += u_flow;
```

```
            edges[from[current].second^1].f -= u_flow; // Arista inversa
            current = from[current].first;
        }
        flow += u_flow ;
        return true;
    }
};
```

10. Treap

AGREGAR PEQUEÑA DESCRIPCIÓN.

```
struct treap{
    typedef struct _node{
        long long x;
        int freq, cnt;
        long long p;
        _node *l, *r;
        _node(long long _x): x(_x), p(((long long)(rand()) << 32 )^rand()),
        cnt(1), freq(1), l(nullptr), r(nullptr){}
        ~_node(){delete l; delete r;}
        void recalc(){
            cnt = freq;
            cnt += ((l) ? (l->cnt) : 0);
            cnt += ((r) ? (r->cnt) : 0);
        }
    }* node;
    node root;
    node merge(node l, node r){
        if(!l || !r) return l ? l : r;
        if(l->p < r->p){
            r->l = merge(l, r->l);
            r->recalc();
            return r;
        } else {
            l->r = merge(l->r, r);
            l->recalc();
            return l;
        }
    }
    void split_by_value(node n, long long d, node &l, node &r){
        l = r = nullptr;
        if(!n) return;
        if(n->x < d){
            split_by_value(n->r, d, n->r, r);
            l = n;
        } else {
            split_by_value(n->l, d, l, n->l);
            r = n;
        }
        n->recalc();
    }
    void split_by_pos(node n, int pos, node &l, Node &r, int l_nodes = 0){
        l = r = NULL;
        if(!n) return;
```

```
        int cur_pos = (n->l) ? (l_nodes + n->l->cnt) : l_nodes;
        if(cur_pos < pos){
            splitFirstNodes(n->r, pos, n->r, r, cur_pos + 1);
            l = n;
        } else {
            splitFirstNodes(n->l, pos, l, n->l, l_nodes);
            r = n;
        }
        n->recalc();
    }
    treap(): root(NULL){}
    void insert_value(long long x){
        node l, m, r;
        split_by_value(root, x, l, m);
        split_by_value(m, x + 1, m, r);
        if(m){
            m->freq++;
            m->cnt++;
        } else m = new _node(x);
        root = merge(merge(l, m), r);
    }
    void erase_value(long long x){
        node l, m, r;
        split_by_value(root, x, l, m);
        split_by_value(m, x + 1, m, r);
        if(!m || m->freq == 1){
            delete m;
            m = nullptr;
        } else {
            m->freq--;
            m->cnt--;
        }
        root = merge(merge(l, m), r);
    }
};
```

11. Strings

11.1. KMP

Complejidad: Tiempo $O(|s|)$ - Memoria extra $O(|s|)$.

```
vector<int> prefix_function(string s){
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j]) j = pi[j-1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

11.2. Suffix array

11.2.1. Construcción

Complejidad: Tiempo $O(|s|\log(|s|))$ - Memoria $O(|s|)$. Calcula la permutación que corresponde a los sufijos ordenados lexicográficamente. `SA[i]` es el índice en el cual empieza el i-ésimo sufijo ordenado.

```
int SA[MAXN], mrank[MAXN];
int tmpSA[MAXN], tmpMrank[MAXN];
void countingSort(int k, int n){
    int freqs[MAXN] = {};
    for(int i = 0; i < n; ++i){
        if(i + k < n) freqs[ mrank[i + k] ]++;
        else freqs[0]++;
    }
    int m = max(100, n);
    for(int i = 0, sfs = 0; i < m; ++i){
        int f = freqs[i];
        freqs[i] = sfs;
        sfs += f;
    }
    for(int i = 0; i < n; ++i){
        if(SA[i] + k < n) tmpSA[ freqs[mrank[ SA[i] + k ]]++ ] = SA[i];
        else tmpSA[ freqs[0]++ ] = SA[i];
    }
    for(int i = 0; i < n; ++i) SA[i] = tmpSA[i];
}

void buildSA(string &str){
    int n = str.size();
    for(int i = 0; i < n; ++i){
        mrank[i] = str[i] - '#';
        SA[i] = i;
    }
    for(int k = 1; k < n; k <<= 1){
        countingSort(k, n);
        countingSort(0, n);
        int r = 0;
        tmpMrank[ SA[0] ] = 0;
        for(int i = 1; i < n; ++i){
            if(mrank[ SA[i] ] != mrank[ SA[i - 1] ] || mrank[ SA[i] + k ] != mrank[ SA[i - 1] + k ])
                tmpMrank[ SA[i] ] = ++r;
            else
                tmpMrank[ SA[i] ] = r;
        }
        for(int i = 0; i < n; ++i) mrank[i] = tmpMrank[i];
    }
}
inline bool suff_compare1(int idx,const string &pattern) {
    return (s.substr(idx).compare(0, pattern.size(), pattern) < 0);
}
inline bool suff_compare2(const string &pattern,int idx) {
    return (s.substr(idx).compare(0, pattern.size(), pattern) > 0);
}
pair<int,int> match(const string &pattern) {
    int *low = lower_bound (SA, SA + s.size(), pattern, suff_compare1);
    int *up = upper_bound (SA, SA + s.size(), pattern, suff_compare2);
    return make_pair((int)(low - SA),(int)(up - SA));
```

```
}
```

11.2.2. Prefijo común más largo

Complejidad: Tiempo $O(|s|)$ - Memoria $O(|s|)$. Calcula la longitud del prefijo común más largo entre dos sufijos consecutivos (lexicográficamente) de `s`. $lcp[i]$ guarda la respuesta para el i-ésimo sufijo y el $(i-1)$-ésimo sufijo.

```
int lcp[MAXN];
void buildLCP(string &str){
    int n = str.size();
    int phi[n];
    phi[SA[0]] = -1;
    for(int i = 1; i < n; ++i) phi[ SA[i] ] = SA[i - 1];
    int plcp[n];
    int k = 0;
    for(int i = 0; i < n; ++i){
        if(phi[i] == -1){
            plcp[i] = 0;
            continue;
        }
        while(i + k < n && phi[i] + k < n && str[i + k] == str[phi[i] + k]) k++;
        plcp[i] = k;
        k = max(k - 1, 0);
    }
    for(int i = 0; i < n; ++i) lcp[i] = plcp[SA[i]];
}
```

11.3. Suffix tree

COPIADO Y PEGADO POR

```
/// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ porcuestionesdetiempo
const int inf = 1e9;
const int maxn = 1e6;
int s[maxn];
map<int, int> to[maxn];
//Root is the vertex 0
//f_pos[i] is the initial index with the letter of the edge that goes from the parent of i to i
//len[i] is the number of letters in the edge that enters in i
//slink[i] is the suffix link
int len[maxn], f_pos[maxn], slink[maxn];
int node, pos;
int sz = 1, n = 0;

int make_node(int _pos, int _len){
    f_pos[sz] = _pos;
    len [sz] = _len;
    return sz++;
}

void go_edge(){
    while(pos > len[to[node][s[n - pos]]]){
        node = to[node][s[n - pos]];
        pos -= len[node];
    }
```

```
}

void add_letter(int c){
    s[n++] = c;
    pos++;
    int last = 0;
    while(pos > 0){
        go_edge();
        int edge = s[n - pos];
        int &v = to[node][edge];
        int t = s[f_pos[v] + pos - 1];
        if(v == 0){
            v = make_node(n - pos, inf);
            //v = make_node(n - pos, 1);
            slink[last] = node;
            last = 0;
        } else if(t == c) {
            slink[last] = node;
            return;
        } else {
            int u = make_node(f_pos[v], pos - 1);
            to[u][c] = make_node(n - 1, inf);
            to[u][t] = v;
            f_pos[v] += pos - 1;
            len [v] -= pos - 1;
            v = u;
            slink[last] = u;
            last = u;
        }
        if(node == 0) pos--;
        else node = slink[node];
    }
}

void correct(int s_size){
    len[0] = 0;
    for (int i = 1; i < sz; i++){
        if (f_pos[i] + len[i] - 1 >= s_size){
            len[i] = (s_size - f_pos[i]);
        }
    }
}

void print_suffix_tree(int from){
    cout << "Edge entering in " << from << " has size " << len[from];
    cout << " and starts in " << f_pos[from] << endl;
    cout << "Node " << from << " goes to: ";
    for (auto u : to[from]){
        cout << u.second << " with " << (char)u.first << " ";
    }
    cout << endl;
    for (auto u : to[from]){
        print_suffix_tree(u.second);
    }
}

```

```
void build(string &s){
    for (int i = 0; i < sz; i++){
        to[i].clear();
    }
    sz = 1;
    node = pos = n = 0;
    len[0] = inf;
    for(int i = 0; i < s.size(); i++)
        add_letter(s[i]);
    correct(s.size());
}

void cutGeneralized(vector<int> &finishPoints){
    for (int i = 0; i < sz; i++){
        int init = f_pos[i];
        int end = f_pos[i] + len[i] - 1;
        int idx = lower_bound(finishPoints.begin(), finishPoints.end(), init) - finishPoints.begin();
        if ((idx != finishPoints.size()) && (finishPoints[idx] <= end)){//Must be cut
            len[i] = (finishPoints[idx] - f_pos[i] + 1);
            to[i].clear();
        }
    }
}


void build_generalized(vector<string> &ss){
    for (int i = 0; i < sz; i++){
        to[i].clear();
    }
    sz = 1;
    node = pos = n = 0;
    len[0] = inf;
    int sep = 256;
    vector<int> finishPoints;
    int next = 0;
    for (int i = 0; i < ss.size(); i++){
        for (int j = 0; j < ss[i].size(); j++){
            add_letter(ss[i][j]);
        }
        next += ss[i].size();
        finishPoints.push_back(next);
        add_letter(sep++);
        next++;
    }
    correct(next);
    cutGeneralized(finishPoints);
}
```

12. Geometría

12.1. Convex hull

Complejidad: $O(n \log n)$. AGREGAR PEQUEÑA DESCRIPCIÓN.

```
/// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ porcuestionesdetiempo
struct pt {
```

```
    double x, y;
};
int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}
bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }
void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }
    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }
    a = st;
}
```

13. Utilidades

13.1. Plantilla tree

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less(int), rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int main(){
    ordered_set S;
    S.insert(1);
    S.insert(4);
    S.insert(0);
    S.insert(-10);
    cout << S.order_of_key(1) << '\n'; /// 2
    cout << S.order_of_key(4) << '\n'; /// 3
```

```
    cout << *(S.find_by_order(1)) << '\n'; /// 0
}
```

13.2. Números aleatorios

`mt19937_64` genera números de 64 bits.

```
random_device rd; // Inicializa el generador de numeros aleatorios
mt19937_64 generator(rd());  // Crea un generador Mersenne Twister con la semilla de random_device
uniform_int_distribution<long long> distribution(1, 1e18);
cout << distribution(generator) << '\n';
```

13.3. Bitmask

```
#define isOn(S, j) (S & (1ll << j))
#define setBit(S, j) (S |= (1ll << j))
#define clearBit(S, j) (S &= ~(1ll << j))
#define toggleBit(S, j) (S ^= (1ll << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1ll << n) - 1ll)
#define modulo(S, N) ((S) & (N - 1)) // S % N, N potencia de 2
#define isPowerOfTwo(S) (!((S) & (S - 1)))
#define nearestPowerOfTwo(S) ((int)pow(2, (int)((log((double)S) / log(2)) + 0.5)) )
```