

Referencia - ICPC

Mathgic

Marzo 2024

Github

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

Índice

0.1. OJO	4
1. Estructuras básicas	4
1.1. Min stack	4
1.2. Min queue	4
2. Ordenamiento	4
2.1. Bucket sort	4
2.2. Merge sort	5
3. Matemáticas	5
3.1. Criba de Eratóstenes	5
3.2. Criba sobre un rango	5
3.3. Criba segmentada	6
3.4. Criba lineal	6
3.5. Algoritmo extendido de Euclides	7
3.6. Solución de ecuaciones diofánticas lineales	7
3.7. Función Phi de Euler	8
3.8. Función sigma	8
3.9. Función de Moebius	9
3.10. Exponenciación binaria	9
4. Sparse table	10
5. Fenwick Tree	11
6. Segment Tree	11
6.1. Actualizaciones puntuales	11
6.2. Actualizaciones sobre rangos	12
7. Sqrt decomposition	13
7.1. Algoritmo de MO	13
8. DSU	13
9. Grafos	14
9.1. Caminos mínimos	14
9.1.1. Dijkstra	14
9.1.2. Bellman-Ford	15
9.2. Árboles	15
9.2.1. MST	15
9.2.2. LCA	16
9.2.3. Sack	17
9.3. Máximo flujo	18
9.4. SCC	19
9.4.1. Kosajaru	19
9.5. 2-Sat	20
10. Treap	21
11. Strings	22
11.1. KMP	22
11.2. Suffix array	22
11.2.1. Construcción	22
11.2.2. Prefijo común más largo	23
11.3. Suffix tree	24

12.Geometría	26
12.1. Convex hull	26
13.Utilidades	27
13.1. Plantilla tree	27
13.2. Números aleatorios	27
13.3. Bitmask	27

0.1. OJO

- a) Se usan macros (MAXN, LOGN, etc) para más comodidad, pero puede causar RTE o MLE cuando los valores son grandes. Pensar en usar `vector<>` (STL) cuando sea conveniente.
- b) agréguele errores/consejos que hay que tener en cuenta sobre las implementaciones y no estemos mucho tiempo tratando de encontrar el error.

1. Estructuras básicas

1.1. Min stack

```
1 struct min_stack{
2     stack<pair<int, int>> st;
3     min_stack(){st.push(make_pair(INT_MAX, INT_MAX));}
4     void push(int v){st.push(make_pair(v, min(v, st.top().second)));}
5     int top(){return st.top().first;}
6     void pop(){if(st.size() > 1)st.pop();}
7     int minV(){return st.top().second;}
8     int size(){return st.size() - 1;}
9     bool empty(){return size() == 0;}
10 };
```

1.2. Min queue

```
1 struct min_queue{
2     min_stack p_in;
3     min_stack p_out;
4     void push(int v){p_in.push(v);}
5     int front(){transfer(); return p_out.top();}
6     void pop(){transfer(); p_out.pop();}
7     int size(){return p_in.size() + p_out.size();}
8     int minV() {return min(p_in.minV(), p_out.minV());}
9     bool empty(){ return size() == 0;}
10    void transfer(){
11        if(p_out.size()) return;
12        while(p_in.size()){
13            p_out.push(p_in.top());
14            p_in.pop();
15        }
16    }
17 };
```

2. Ordenamiento

2.1. Bucket sort

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

```
1 void bucketSort(int arr[], int n){
2     int cub[MAXVAL + 1] = {};
3     for(int i = 0; i < n; ++i) cub[ arr[i] ]++;
4     int idx = 0;
5     for(int i = 0; i <= MAXVAL; ++i){
6         while( cub[i] ){
7             arr[idx++] = i;
8             cub[i]--;
9         }
10    }
```

```

9         }
10     }
11 }

```

2.2. Merge sort

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

```

1 void mergeSort(int arr[], int ini, int fin){
2     if(ini == fin) return;
3     int mitad = (ini + fin) / 2;
4     mergeSort(arr, ini, mitad);
5     mergeSort(arr, mitad + 1, fin);
6
7     int tam1 = mitad - ini + 1, tam2 = fin - mitad;
8     int mitad1[tam1], mitad2[tam2];
9     for(int i = ini, idx = 0; i <= mitad; ++i, idx++)
10         mitad1[idx] = arr[i];
11     for(int i = mitad + 1, idx = 0; i <= fin; ++i, idx++)
12         mitad2[idx] = arr[i];
13
14     for(int i = ini, idx1 = 0, idx2 = 0; i <= fin; ++i){
15         if(idx1 < tam1 && idx2 < tam2){ /// si quedan elementos en ambas mitades
16             arr[i] = mitad1[idx1] < mitad2[idx2] ? mitad1[idx1++] : mitad2[idx2++];
17         } else if(idx1 < tam1){ /// si solo hay elementos en mitad1
18             arr[i] = idx1 < tam1 ? mitad1[idx1++] : mitad2[idx2++];
19         }
20     }
21 }

```

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 .

```

1 void criba(int n, vector<int> &primos){
2     primos.clear();
3     if(n < 2) return;
4     vector<bool> no_primo(n + 1);
5     no_primo[0] = no_primo[1] = true;
6     for(long long i = 3; i * i <= n; i += 2){
7         if(no_primo[i]) continue;
8         for(long long j = i * i; j <= n; j += 2 * i)
9             no_primo[j] = true;
10    }
11    primos.push_back(2);
12    for(int i = 3; i <= n; i += 2){
13        if(!no_primo[i])
14            primos.push_back(i);
15    }
16 }

```

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 $[a, b]$.

```

1 void cribaSobreRango(long long a, long long b, vector<long long> &primos){
2     a = max(a, 0ll);
3     b = max(b, 0ll);
4     long long tam = b - a + 1;
5     vector<int> primosRaiz;
6     criba(sqrt(b) + 1, primosRaiz);
7     bool no_primo[tam] = {};
8     primos.clear();
9     for(long long p : primosRaiz){
10         long long ini = p * max(p, (a + p - 1) / p);
11         for(long long m = ini; m <= b; m += p){
12             no_primo[m - a] = true;
13         }
14     }
15     for(long long i = 0; i < tam; ++i){
16         if(no_primo[i] || i + a < 2) continue;
17         primos.push_back(i + a);
18     }
19 }

```

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 .

```

1 int cuentaPrimos(int n){
2     if(n < 2) return 0;
3     const int S = sqrt(n);
4     vector<int> primosRaiz;
5     criba(sqrt(n) + 1, primosRaiz);
6     int ans = 0;
7     vector<char> no_primo(S);
8     for(int ini = 0; ini <= n; ini += S){
9         fill(no_primo.begin(), no_primo.end(), false);
10        for(int p : primosRaiz){
11            int m = p * max(p, (ini + p - 1) / p) - ini;
12            for(; m <= S; m += p)
13                no_primo[m] = true;
14        }
15        for(int i = 0; i < S && i + ini <= n; ++i)
16            if(!no_primo[i] && 1 < i + ini)
17                ans++;
18    }
19    return ans;
20 }

```

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 $[2, n]$.

```

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

```

```

8         primos.push_back(i);
9     }
10    for(int j = 0; i * (long long)primos[j] <= n; ++j){
11        lp[i * primos[j]] = primos[j];
12        if(primos[j] == lp[i])
13            break;
14    }
15 }
16 }

```

3.5. Algoritmo extendido de Euclides

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

```

1 int gcdExtendido(int a, int b, int &x, int &y){
2     if(!b){
3         x = 1;
4         y = 0;
5         return a;
6     }
7     int x1, y1;
8     int g = gcdExtendido(b, a % b, x1, y1);
9     x = y1;
10    y = x1 - y1 * (a / b);
11    return g;
12 }

```

3.6. Solución de ecuaciones diofánticas lineales

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

```

1 bool encuentra_solucion(int a, int b, int c, int &x, int &y){
2     int g = gcdExtendido(abs(a), abs(b), x, y);
3     if(c % g) return false;
4     x *= c / g;
5     y *= c / g;
6     if(a < 0) x = -x;
7     if(b < 0) y = -y;
8     return true;
9 }

```

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

```

1 void cambia_solucion(int &x, int &y, int a, int b, int cnt, int g = 1) {
2     x += cnt * b / g;
3     y -= cnt * a / g;
4 }

```

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

```

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

```

```

8  /// Signos de a, b nos sirven para pasar a la
9  /// siguiente (anterior) solucion
10 int sign_a = a > 0 ? +1 : -1;
11 int sign_b = b > 0 ? +1 : -1;
12 /// pasa a la minima solucion tal que minx <= x
13 cambia_solucion(x, y, a, b, (minx - x) / b);
14 /// si x < minx, pasa a la siguiente para que minx <= x
15 if(x < minx) cambia_solucion(x, y, a, b, sign_b);
16 if(x > maxx) return 0; /// si x > maxx, entonces no hay x solucion tal que x in [minx, maxx]
17 int lx1 = x;
18 /// pasa a la maxima solucion tal que x <= maxx
19 cambia_solucion(x, y, a, b, (maxx - x) / b);
20 if(x > maxx) cambia_solucion(x, y, a, b, -sign_b); /// si x > maxx, pasa a la solucion anterior
21 int rx1 = x;
22 /// hace todo lo anterior pero con y
23 cambia_solucion(x, y, a, b, -(miny - y) / a);
24 if(y < miny) cambia_solucion(x, y, a, b, -sign_a);
25 if(y > maxy) return 0;
26 int lx2 = x;
27 cambia_solucion(x, y, a, b, -(maxy - y) / a);
28 if(y > maxy) cambia_solucion(x, y, a, b, sign_a);
29 int rx2 = x;
30 /// como al encontrar las x tomando y como criterio no nos asegura
31 /// que esten ordenadas, entonces las ordenamos
32 if(lx2 > rx2) swap(lx2, rx2);
33 /// obtenemos la interseccion de los intervalos
34 int lx = max(lx1, lx2);
35 int rx = min(rx1, rx2);
36 if(lx > rx) return 0; /// no existen soluciones, interseccion vacia
37 /// las soluciones (por x) van de b en b (b/g en b/g pero dividimos al principio)
38 return (rx - lx) / abs(b) + 1;
39 }

```

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 .

```

1  int phi(int n){
2      if(n <= 1) return 1;
3      if(!dp[n]){
4          int pot = 1, p = lp[n], n0 = n;
5          while(n0 % p == 0){
6              pot *= p;
7              n0 /= p;
8          }
9          pot /= p;
10         dp[n] = pot * (p - 1) * phi(n0);
11     }
12     return dp[n];
13 }

```

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 .


```

1  long long sigma0(int n){
2      if(n <= 1) return 1;
3      if(!dp[n]){
4          long long exp = 0, p = lp[n], n0 = n;
5          while(n0 % p == 0){
6              exp++;
7              n0 /= p;
8          }
9          dp[n] = (exp + 1) * sigma0(n0);
10     }
11     return dp[n];
12 }

```

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 .

```

1  long long sigma1(int n){
2      if(n <= 1) return 1;
3      if(!dp[n]){
4          long long pot = 1, p = lp[n], n0 = n;
5          while(n0 % p == 0){
6              pot *= p;
7              n0 /= p;
8          }
9          pot *= p;
10         dp[n] = (pot - 1) / (p - 1) * sigma1(n0);
11     }
12     return dp[n];
13 }

```

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.

```

1  int moebius(int n){
2      if(n <= 1) return 1;
3      if(dp[n] == -7){
4          int exp = 0, p = lp[n], n0 = n;
5          while(n0 % p == 0){
6              exp++;
7              n0 /= p;
8          }
9          dp[n] = (exp > 1 ? 0 : -1 * moebius(n0));
10     }
11     return dp[n];
12 }

```

3.10. Exponenciación binaria

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

```

1  int binExp(int a, int b){
2      int ans = 1;
3      while(b){
4          if(b % 2) ans *= a;

```

```

5         a *= a;
6         b /= 2;
7     }
8     return ans;
9 }

```

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

```

1 int binExp(int a, int b){
2     if(!b) return 1;
3     int tmp = binExp(a, b / 2);
4     if(b % 2) return tmp * tmp * a;
5     return tmp * tmp;
6 }

```

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 $\lceil \log_2(\text{MAXN}) \rceil$.

```

1 struct sparse_table{
2     int n, NEUTRO;
3     vector<vector<int>> ST;
4     vector<int> lg2;
5     int f(int a, int b){return a + b;}
6     sparse_table(int _n, int data[]){
7         n = _n;
8         NEUTRO = 0;
9         lg2.resize(n + 1);
10        lg2[1] = 0;
11        for(int i = 2; i <= n; ++i) lg2[i] = lg2[i / 2] + 1;
12        ST.resize(lg2[n] + 1, vector<int>(n + 1, NEUTRO));
13        for(int i = 0; i < n; ++i) ST[0][i] = data[i];
14        for(int k = 1; k <= lg2[n]; ++k){
15            int fin = (1 << k) - 1;
16            for(int i = 0; i + fin < n; ++i)
17                ST[k][i] = f(ST[k - 1][i], ST[k - 1][i + (1 << (k - 1))]);
18        }
19    }
20    int query(int l, int r){
21        if(l > r) return NEUTRO;
22        int ans = NEUTRO;
23        for(int k = lg2[n]; 0 <= k; --k){
24            if( r - l + 1 < (1 << k) ) continue;
25            ans = f(ans, ST[k][l]);
26            l += 1 << k;
27        }
28        return ans;
29    }
30    int queryIdem(int l, int r){
31        if(l > r) return NEUTRO;
32        int lg = lg2[r - l + 1];
33        return f(ST[lg][l], ST[lg][r - (1 << lg) + 1]);
34    }
35 };

```

5. Fenwick Tree

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

```
1 struct FenwickTree{
2     int n, BIT[MAXN];
3     FenwickTree(int n_size){
4         n = n_size;
5         memset(BIT, 0, sizeof(BIT));
6     }
7     void add(int pos, int x){
8         while(pos <= n){
9             BIT[pos] += x;
10            pos += pos & -pos;
11        }
12    }
13    int sum(int pos){
14        int ret = 0;
15        while(pos){
16            ret += BIT[pos];
17            pos -= pos & -pos;
18        }
19        return ret;
20    }
21 };
```

6. Segment Tree

Nodo del Segment Tree:

```
1 struct nodo{
2     int val, lazy;
3     nodo():val(0), lazy(0){} /// inicializa con el neutro y sin lazy pendiente
4     nodo(int x, int lz = 0):val(x), lazy(lz){}
5     const nodo operator+(const nodo &b)const{
6         return nodo(val + b.val);
7     }
8 }
```

6.1. Actualizaciones puntuales

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

```
1 struct segment_tree{
2     struct node{...}nodes[4 * MAXN + 1];
3     segment_tree(int n, int data[]){
4         build(1, n, data);
5     }
6     void build(int left, int right, int data[], int pos = 1){
7         if(left == right){
8             nodes[pos].val = data[left];
9             return;
10        }
11        int mid = (left + right) / 2;
12        build(left, mid, data, pos * 2);
13        build(mid + 1, right, data, pos * 2 + 1);
14        nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
15    }
```

```

15     }
16     void update(int x, int idx, int left, int right, int pos = 1){
17         if(idx < left || right < idx) return;
18         if(left == right){
19             nodes[pos].val += x;
20             return;
21         }
22         int mid = (left + right) / 2;
23         update(x, idx, left, mid, pos * 2);
24         update(x, idx, mid + 1, right, pos * 2 + 1);
25         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
26     }
27     node query(int l, int r, int left, int right, int pos = 1){
28         if(r < left || right < l) return node(); /// Devuelve el neutro
29         if(l <= left && right <= r) return nodes[pos];
30         int mid = (left + right) / 2;
31         return query(l, r, left, mid, pos * 2) + query(l, r, mid + 1, right, pos * 2 + 1);
32     }
33 };

```

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 $O(n)$.

```

1  struct segment_tree{
2      struct node{...}nodes[4 * MAXN + 1];
3      segment_tree(int n, int data[]){...}
4      void build(int left, int right, int data[], int pos = 1){...}
5      void combineLazy(int lz, int pos){nodes[pos].lazy += lz;}
6      void applyLazy(int pos, int tam){
7          nodes[pos].val += nodes[pos].lazy * tam;
8          nodes[pos].lazy = 0;
9      }
10     void pushLazy(int pos, int left, int right){
11         int tam = abs(right - left + 1);
12         if(1 < tam){
13             combineLazy(nodes[pos].lazy, pos * 2);
14             combineLazy(nodes[pos].lazy, pos * 2 + 1);
15         }
16         applyLazy(pos, tam);
17     }
18     void update(int x, int l, int r, int left, int right, int pos = 1){
19         pushLazy(pos, left, right);
20         if(r < left || right < l) return;
21         if(l <= left && right <= r){
22             combineLazy(x, pos);
23             pushLazy(pos, left, right);
24             return;
25         }
26         int mid = (left + right) / 2;
27         update(x, l, r, left, mid, pos * 2);
28         update(x, l, r, mid + 1, right, pos * 2 + 1);
29         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
30     }
31     node query(int l, int r, int left, int right, int pos = 1){...}
32 };

```

7. Sqrt decomposition

7.1. Algoritmo de MO

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

```
1  const int block_size = 300; /// Ajustable
2  struct query {
3      int l, r, block, i;
4      bool operator<(const query &b) const {
5          if(block == b.block) return r < b.r;
6          return block < b.block;
7      }
8  };
9  void add(int idx){/**TO-DO*/}
10 void remove(int idx){/**TO-DO*/}
11 int get_answer(){return 0; /**TO-DO*/}
12 vector<int> solve(vector<query> &queries) {
13     vector<int> answers(queries.size());
14     sort(queries.begin(), queries.end());
15     int l_act = 0;
16     int r_act = -1;
17     for(query q : queries){
18         while(l_act > q.l) add(--l_act);
19         while(r_act < q.r) add(++r_act);
20         while(l_act < q.l) remove(l_act++);
21         while(r_act > q.r) remove(r_act--);
22         answers[q.i] = get_answer();
23     }
24     return answers;
25 }
```

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*.

```
1  struct dsu{
2      struct action{
3          int x_p, y_p;
4          int rank_y;
5      };
6      int RA[MAXN], P[MAXN];
7      vector<action> actions;
8      dsu(int n){
9          for(int i = 0; i < n; ++i){
10             RA[i] = 1;
11             P[i] = i;
12         }
13     }
14     int root(int x){
15         return (x == P[x] ? x : P[x] = root(P[x]));
16     }
17     void join(int x, int y, bool recording){
```

```

18     x = root(x);
19     y = root(y);
20     if(x == y) return;
21     if(RA[x] >= RA[y]) swap(x, y);
22     if(recording) actions.push_back({x, y, RA[y]});
23     RA[y] += RA[x];
24     P[x] = y;
25 }
26 void rollback(int times){
27     while(times > 0 && actions.size()){
28         action act = actions.back();
29         actions.pop_back();
30         sets.RA[act.y_p] = act.rank_y;
31         sets.P[act.x_p] = act.x_p;
32     }
33 }
34 };

```

9. Grafos

```

1 struct edge{
2     int from, to;
3     int64_t w;
4     const bool operator<(const edge &b)const{
5         return w > b.w;
6     }
7 };
8 struct pos{
9     int from;
10    int64_t c;
11    const bool operator<(const pos &b)const{
12        return c > b.c;
13    }
14 };

```

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.

```

1 int64_t dijkstra(int a, int b, vector<edge> graph[]){
2     int64_t dist[MAXN];
3     bool vis[MAXN];
4     fill(dist, dist + MAXN, LLONG_MAX);
5     memset(vis, 0, sizeof(vis));
6     priority_queue<pos> q;
7     q.push(pos{a, 0});
8     dist[a] = 0;
9     while(!q.empty()){
10        pos act = q.top();
11        q.pop();
12        if(vis[act.from]) continue;
13        vis[act.from] = true;
14        for(edge &e : grafo[act.from]){
15            if(dist[e.to] < dist[act.from] + e.w) continue;

```

```

16         dist[e.to] = dist[act.from] + e.w;
17         q.push(pos{e.to, dist[e.to]});
18     }
19 }
20 return dist[b];
21 }

```

9.1.2. Bellman-Ford

```

1 vector<int> bellman_ford(int s, int n, vector<edge> &edges, bool cycles = false){
2     vector<int> d(n, (cycles ? 0 : INT_MAX));
3     d[s] = 0;
4     vector<int> P(n, -1); /// Predecesor
5     for(int i = 0; i < n - 1; ++i){
6         for(edge &e : edges){
7             if(d[e.from] == INT_MAX) continue;
8             if(d[e.to] > d[e.from] + e.w){
9                 d[e.to] = d[e.from] + e.w;
10                P[e.to] = e.from;
11            }
12        }
13    }
14    int last_relax = -1;
15    for(edge &e : edges){
16        if(d[e.from] == INT_MAX) continue;
17        if(d[e.to] > d[e.from] + e.w){
18            d[e.to] = d[e.from] + e.w;
19            P[e.to] = e.from;
20            last_relax = e.to;
21        }
22    }
23    if(last_relax == -1) return d;
24    return {}; /// VACIO
25 }

```

9.2. Árboles

9.2.1. MST

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

```

1 int64_t prim(vector<edge> graph[]){
2     int64_t eCost[MAXN];
3     bool vis[MAXN];
4     memset(vis, 0, sizeof(vis));
5     fill(eCost, eCost + MAXN, LLONG_MAX);
6     int64_t ans = 0;
7     priority_queue<edge> q;
8     q.push(edge{1, 1, 0});
9     while(q.size()){
10        int node = q.top().to;
11        int64_t w = q.top().w;
12        q.pop();
13        if(vis[node]) continue;
14        vis[node] = true;
15        ans += w;
16        for(edge &e : graph[node]){

```

```

17         if(vis[e.to] || eCost[e.to] <= e.w) continue;
18         eCost[e.to] = e.w;
19         q.push(e);
20     }
21 }
22 return ans;
23 }

```

Kruskal. Complejidad: Tiempo $O(|E| \log |E|)$.

```

1 int64_t kruskal(vector<edge> &edges, int n){
2     sort(edges.begin(), edges.end());
3     dsu mset(n);
4     int64_t res = 0;
5     for(edge &e : edges){
6         if(mset.root(e.from) == mset.root(e.to)) continue;
7         mset.join(e.from, e.to);
8         res += e.w;
9     }
10    return res;
11 }

```

Boruvka. Complejidad: Tiempo $O(|E| \log |V|)$. $|V| = n$. `dsu.join()` devuelve `true` si la unión se llevó a cabo o `false` en otro caso.

```

1 int64_t boruvka(vector<edge> &edges, int n){
2     dsu mset(n);
3     int min_edge[n];
4     int64_t res = 0;
5     while(mset.cnt_comp > 1){
6         fill(min_edge, min_edge + n, -1);
7         for(int i = 0; i < edges.size(); ++i){
8             int u = mset.root(edges[i].from);
9             int v = mset.root(edges[i].to);
10            if(u == v) continue;
11            if(min_edge[u] == -1 || edges[i].w < edges[min_edge[u]].w) min_edge[u] = i;
12            if(min_edge[v] == -1 || edges[i].w < edges[min_edge[v]].w) min_edge[v] = i;
13        }
14        for(int i = 0; i < n; ++i){
15            int idx_e = min_edge[i];
16            if(idx_e == -1) continue;
17            res += mset.join(edges[idx_e].from, edges[idx_e].to) * edges[idx_e].w;
18        }
19    }
20    return res;
21 }

```

9.2.2. LCA

Complejidad: Tiempo de preproceso $O(|V| \log |V|)$. Tiempo de LCA y n -ésimo ancestro $O(\log |V|)$

```

1 void precalc(int node, int p = 0, int d = 1){
2     depth[node] = d;
3     P[0][node] = p;
4     for(int k = 1; k <= LOGN; ++k)
5         P[k][node] = P[k - 1][P[k - 1][node]];
6     for(int child : tree[node])
7         if(p != child) precalc(child, node, d + 1);
8 }

```



```

9  int LCA(int a, int b){
10     if(depth[b] < depth[a]) swap(a, b);
11     int dif = depth[b] - depth[a];
12     for(int k = LOGN; 0 <= k; --k)
13         if(is_on(dif, k)) b = P[k][b];
14     if(a == b) return a;
15     for(int k = LOGN; 0 <= k; --k){
16         if(P[k][a] != P[k][b]){
17             a = P[k][a];
18             b = P[k][b];
19         }
20     }
21     return P[0][a];
22 }
23 int nth_ancestor(int u, int n){
24     for(int k = LOGN; 0 <= k; --k)
25         if(is_on(n, k)) u = P[k][u];
26     return u;
27 }

```

9.2.3. Sack

Complejidad: Tiempo $O(|V| \log |V|)$.

```

1  void precalc(int node, int p = 0){
2      subtree_size[node] = 1;
3      depth[node] = depth[p] + 1;
4      for(int v : tree[node]){
5          if(v == p) continue;
6          precalc(v, node);
7          subtree_size[node] += subtree_size[v];
8      }
9  }
10 void add(int node, int x, int p = 0){
11     /// add node here
12     /// add subtree
13     for(int v : tree[node])
14         if(v != p && !big[v])
15             add(v, x, node);
16 }
17 void dfs(int node, bool keep, int p = 0){
18     int maxi = -1, big_child = -1;
19     for(int v : tree[node]) /// Search for big_child
20         if(v != p && subtree_size[v] > maxi)
21             maxi = subtree_size[v], big_child = v;
22     for(int v : tree[node])
23         if(v != p && v != big_child)
24             dfs(v, false, node); /// run a dfs on small childs and clear them
25     if(big_child != -1)
26         dfs(big_child, true, node), big[big_child] = 1; /// big_child marked as big and not cleared
27     add(node, 1, p);
28     /// answer queries here
29     if(big_child != -1) big[big_child] = 0;
30     if(!keep) add(node, -1, p);
31 }

```

9.3. Máximo flujo

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

```
1 struct edge {
2     int64_t c; // capacity
3     int64_t f; // flow
4     int to;
5 };
6 class ford_fulkerson {
7 public:
8     ford_fulkerson (vector<vector<pair<int, int64_t>>> &graph) : graph(graph){}
9     int64_t get_max_flow(int s, int t){
10         init();
11         int64_t f = 0;
12         while(find_and_update(s, t, f)){
13             return f;
14         }
15 private:
16     vector<vector<pair<int, int64_t>>> graph; // graph (to, capacity)
17     vector<edge> edges; // List of edges (including the inverse ones)
18     vector<vector<int>> edge_indexes; // indexes of edges going out from each vertex
19     void init(){
20         edges.clear();
21         edge_indexes.clear(); edge_indexes.resize(graph.size());
22         for(int i = 0; i < graph.size(); i++){
23             for(int j = 0; j < graph[i].size(); j++){
24                 edges.push_back({graph[i][j].second, 0, graph[i][j].first});
25                 edges.push_back({0, 0, i});
26                 edge_indexes[i].push_back(edges.size() - 2);
27                 edge_indexes[graph[i][j].first].push_back(edges.size() - 1);
28             }
29         }
30     }
31     bool find_and_update(int s, int t, int64_t &flow){
32         // Encontrar camino desat con BFS
33         queue<int> q;
34         // Desde donde llego y con que arista
35         vector<pair<int, int>> from(graph.size(), make_pair(-1, -1));
36         q.push(s);
37         from[s] = make_pair(s, -1);
38         bool found = false;
39         while(q.size() && (!found)){
40             int u = q.front(); q.pop();
41             for(int i = 0; i < edge_indexes[u].size(); i++){
42                 int eI = edge_indexes[u][i];
43                 if((edges[eI].c > edges[eI].f) && (from[edges[eI].to].first == -1)){
44                     from[edges[eI].to] = make_pair(u, eI);
45                     q.push(edges[eI].to);
46                     if(edges[eI].to == t) found = true;
47                 }
48             }
49         }
50         if(!found) return false;
51         // Encontrar cap. minima del camino de aumento
52         int64_t u_flow = LLONG_MAX;
53         int current = t;
```

```

54     while(current != s) {
55         u_flow = min(u_flow, edges[from[current].second].c - edges[from[current].second].f);
56         current = from[current].first;
57     }
58     current = t;
59     // Actualizar flujo
60     while(current != s){
61         edges[from[current].second].f += u_flow;
62         edges[from[current].second^1].f -= u_flow; // Arista inversa
63         current = from[current].first;
64     }
65     flow += u_flow ;
66     return true;
67 }
68 };

```

9.4. SCC

9.4.1. Kosajaru

Complejidad: Tiempo $O(n)$.

```

1  void dfs(int node, vector<int> &topo_ord){
2      if(vis[node]) return;
3      vis[node] = true;
4      for(int v : graph[node]) dfs(v, topo_ord);
5      topo_ord.push_back(node);
6  }
7  void assign_scc(int node, const int id){
8      if(vis[node]) return;
9      vis[node] = true;
10     scc[node] = id;
11     for(int v : inv_graph[node]) assign_scc(v, id);
12 }
13 int kosajaru(int n){ /// devuelve la cantidad de scc.
14     memset(vis, 0, sizeof(vis));
15     vector<int> topo_ord;
16     for(int i = 1; i <= n; ++i) dfs(i, topo_ord);
17     reverse(topo_ord.begin(), topo_ord.end());
18     memset(vis, 0, sizeof(vis));
19     int id = 0;
20     for(int u : topo_ord) if(!vis[u]) assign_scc(u, id++);
21     return id;
22 }
23 void build_scc_graph(int n, int n_scc){
24     for(int u = 0; u < n; ++u)
25         for(int v : graph[u])
26             if(scc[u] != scc[v])
27                 scc_graph[scc[u]].push_back(scc[v]);
28     for(int u = 0; u < n_scc; ++u){
29         sort(scc_graph[u].begin(), scc_graph[u].end());
30         auto it = unique(scc_graph[u].begin(), scc_graph[u].end());
31         scc_graph[u].resize(it - scc_graph[u].begin());
32         for(int v : scc_graph[u])
33             inv_scc_graph[v].push_back(u);
34     }
35 }

```

9.5. 2-Sat

Complejidad: Tiempo en responder $O(n)$.

```
1 struct two_sat{
2     int n;
3     vector<vector<int>> graph, inv_graph;
4     vector<int> scc, ans;
5     vector<bool> vis;
6     two_sat(){}
7     two_sat(int _n){
8         n = _n;
9         graph.resize(2 * n);
10        inv_graph.resize(2 * n);
11        scc.resize(2 * n);
12        vis.resize(2 * n);
13        ans.resize(n);
14    }
15    void add_edge(int u, int v){
16        graph[u].push_back(v);
17        inv_graph[v].push_back(u);
18    }
19    /// al menos una es verdadera
20    void add_or(int p, bool val_p, int q, bool val_q){
21        add_edge(p + (val_p ? n : 0), q + (val_q ? 0 : n));
22        add_edge(q + (val_q ? n : 0), p + (val_p ? 0 : n));
23    }
24    /// exactamente una es verdadera
25    void add_xor(int p, bool val_p, int q, bool val_q){
26        add_or(p, val_p, q, val_q);
27        add_or(p, !val_p, q, !val_q);
28    }
29    /// p y q tienen el mismo valor
30    void add_and(int p, bool val_p, int q, bool val_q){
31        add_xor(p, !val_p, q, val_q);
32    }
33    /// Kosajaru
34    void dfs(int node, vector<int> &topo_ord){...}
35    void assign_scc(int node, const int id){...}
36    /// construye respuesta
37    bool build_ans(){
38        fill(vis.begin(), vis.end(), false);
39        vector<int> topo_ord;
40        for(int i = 0; i < 2 * n; ++i) dfs(i, topo_ord);
41        fill(vis.begin(), vis.end(), false);
42        reverse(topo_ord.begin(), topo_ord.end());
43        int id = 0;
44        for(int u : topo_ord) if(!vis[u]) assign_scc(u, id++);
45        for(int i = 0; i < n; ++i){
46            if(scc[i] == scc[i + n]) return false;
47            ans[i] = (scc[i] < scc[i + n] ? 0 : 1);
48        }
49        return true;
50    }
51 };
```

10. Treap

AGREGAR PEQUEÑA DESCRIPCIÓN.

```
1 struct treap{
2     typedef struct _node{
3         long long x;
4         int freq, cnt;
5         long long p;
6         _node *l, *r;
7         _node(long long _x): x(_x), p(((long long)(rand()) << 32 )^rand()),
8         cnt(1), freq(1), l(nullptr), r(nullptr){}
9         ~_node(){delete l; delete r;}
10        void recalc(){
11            cnt = freq;
12            cnt += ((l) ? (l->cnt) : 0);
13            cnt += ((r) ? (r->cnt) : 0);
14        }
15    }* node;
16    node root;
17    node merge(node l, node r){
18        if(!l || !r) return l ? l : r;
19        if(l->p < r->p){
20            r->l = merge(l, r->l);
21            r->recalc();
22            return r;
23        } else {
24            l->r = merge(l->r, r);
25            l->recalc();
26            return l;
27        }
28    }
29    void split_by_value(node n, long long d, node &l, node &r){
30        l = r = nullptr;
31        if(!n) return;
32        if(n->x < d){
33            split_by_value(n->r, d, n->r, r);
34            l = n;
35        } else {
36            split_by_value(n->l, d, l, n->l);
37            r = n;
38        }
39        n->recalc();
40    }
41    void split_by_pos(node n, int pos, node &l, Node &r, int l_nodes = 0){
42        l = r = NULL;
43        if(!n) return;
44        int cur_pos = (n->l) ? (l_nodes + n->l->cnt) : l_nodes;
45        if(cur_pos < pos){
46            splitFirstNodes(n->r, pos, n->r, r, cur_pos + 1);
47            l = n;
48        } else {
49            splitFirstNodes(n->l, pos, l, n->l, l_nodes);
50            r = n;
51        }
52        n->recalc();
53    }
```

```

54     treap(): root(NULL){}
55     void insert_value(long long x){
56         node l, m, r;
57         split_by_value(root, x, l, m);
58         split_by_value(m, x + 1, m, r);
59         if(m){
60             m->freq++;
61             m->cnt++;
62         } else m = new _node(x);
63         root = merge(merge(l, m), r);
64     }
65     void erase_value(long long x){
66         node l, m, r;
67         split_by_value(root, x, l, m);
68         split_by_value(m, x + 1, m, r);
69         if(!m || m->freq == 1){
70             delete m;
71             m = nullptr;
72         } else {
73             m->freq--;
74             m->cnt--;
75         }
76         root = merge(merge(l, m), r);
77     }
78 };

```

11. Strings

11.1. KMP

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

```

1  vector<int> prefix_function(string s){
2      int n = (int)s.length();
3      vector<int> pi(n);
4      for (int i = 1; i < n; i++) {
5          int j = pi[i-1];
6          while (j > 0 && s[i] != s[j]) j = pi[j-1];
7          if (s[i] == s[j]) j++;
8          pi[i] = j;
9      }
10     return pi;
11 }

```

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.

```

1  int SA[MAXN], mrank[MAXN];
2  int tmpSA[MAXN], tmpMrank[MAXN];
3  void countingSort(int k, int n){
4      int freqs[MAXN] = {};
5      for(int i = 0; i < n; ++i){
6          if(i + k < n) freqs[ mrank[i + k] ]++;
7          else freqs[0]++;

```

```

8     }
9     int m = max(100, n);
10    for(int i = 0, sfs = 0; i < m; ++i){
11        int f = freqs[i];
12        freqs[i] = sfs;
13        sfs += f;
14    }
15    for(int i = 0; i < n; ++i){
16        if(SA[i] + k < n) tmpSA[ freqs[mrank[ SA[i] + k ]]]++ ] = SA[i];
17        else tmpSA[ freqs[0]]++ ] = SA[i];
18    }
19    for(int i = 0; i < n; ++i) SA[i] = tmpSA[i];
20 }
21
22 void buildSA(string &str){
23     int n = str.size();
24     for(int i = 0; i < n; ++i){
25         mrank[i] = str[i] - '#';
26         SA[i] = i;
27     }
28     for(int k = 1; k < n; k <= 1){
29         countingSort(k, n);
30         countingSort(0, n);
31         int r = 0;
32         tmpMrank[ SA[0] ] = 0;
33         for(int i = 1; i < n; ++i){
34             if(mrank[ SA[i] ] != mrank[ SA[i - 1] ] || mrank[ SA[i] + k ] != mrank[ SA[i - 1] + k ])
35                 tmpMrank[ SA[i] ] = ++r;
36             else
37                 tmpMrank[ SA[i] ] = r;
38         }
39         for(int i = 0; i < n; ++i) mrank[i] = tmpMrank[i];
40     }
41 }
42 inline bool suff_compare1(int idx, const string &pattern) {
43     return (s.substr(idx).compare(0, pattern.size(), pattern) < 0);
44 }
45 inline bool suff_compare2(const string &pattern, int idx) {
46     return (s.substr(idx).compare(0, pattern.size(), pattern) > 0);
47 }
48 pair<int, int> match(const string &pattern) {
49     int *low = lower_bound (SA, SA + s.size(), pattern, suff_compare1);
50     int *up = upper_bound (SA, SA + s.size(), pattern, suff_compare2);
51     return make_pair((int)(low - SA), (int)(up - SA));
52 }

```

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.

```

1  int lcp[MAXN];
2  void buildLCP(string &str){
3      int n = str.size();
4      int phi[n];
5      phi[SA[0]] = -1;

```

```

6     for(int i = 1; i < n; ++i) phi[ SA[i] ] = SA[i - 1];
7     int plcp[n];
8     int k = 0;
9     for(int i = 0; i < n; ++i){
10         if(phi[i] == -1){
11             plcp[i] = 0;
12             continue;
13         }
14         while(i + k < n && phi[i] + k < n && str[i + k] == str[phi[i] + k]) k++;
15         plcp[i] = k;
16         k = max(k - 1, 0);
17     }
18     for(int i = 0; i < n; ++i) lcp[i] = plcp[SA[i]];
19 }

```

11.3. Suffix tree

COPIADO Y PEGADO POR

```

1  /// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ por cuestiones de tiempo
2  const int inf = 1e9;
3  const int maxn = 1e6;
4  int s[maxn];
5  map<int, int> to[maxn];
6  //Root is the vertex 0
7  //f_pos[i] is the initial index with the letter of the edge that goes from the parent of i to i
8  //len[i] is the number of letters in the edge that enters in i
9  //slink[i] is the suffix link
10 int len[maxn], f_pos[maxn], slink[maxn];
11 int node, pos;
12 int sz = 1, n = 0;
13
14 int make_node(int _pos, int _len){
15     f_pos[sz] = _pos;
16     len [sz] = _len;
17     return sz++;
18 }
19
20 void go_edge(){
21     while(pos > len[to[node][s[n - pos]]]){
22         node = to[node][s[n - pos]];
23         pos -= len[node];
24     }
25 }
26
27 void add_letter(int c){
28     s[n++] = c;
29     pos++;
30     int last = 0;
31     while(pos > 0){
32         go_edge();
33         int edge = s[n - pos];
34         int &v = to[node][edge];
35         int t = s[f_pos[v] + pos - 1];
36         if(v == 0){
37             v = make_node(n - pos, inf);
38             //v = make_node(n - pos, 1);

```



```

39         slink[last] = node;
40         last = 0;
41     } else if(t == c) {
42         slink[last] = node;
43         return;
44     } else {
45         int u = make_node(f_pos[v], pos - 1);
46         to[u][c] = make_node(n - 1, inf);
47         to[u][t] = v;
48         f_pos[v] += pos - 1;
49         len[v] -= pos - 1;
50         v = u;
51         slink[last] = u;
52         last = u;
53     }
54     if(node == 0) pos--;
55     else node = slink[node];
56 }
57 }
58
59 void correct(int s_size){
60     len[0] = 0;
61     for (int i = 1; i < sz; i++){
62         if (f_pos[i] + len[i] - 1 >= s_size){
63             len[i] = (s_size - f_pos[i]);
64         }
65     }
66 }
67
68 void print_suffix_tree(int from){
69     cout << "Edge entering in " << from << " has size " << len[from];
70     cout << " and starts in " << f_pos[from] << endl;
71     cout << "Node " << from << " goes to: ";
72     for (auto u : to[from]){
73         cout << u.second << " with " << (char)u.first << " ";
74     }
75     cout << endl;
76     for (auto u : to[from]){
77         print_suffix_tree(u.second);
78     }
79 }
80
81 void build(string &s){
82     for (int i = 0; i < sz; i++){
83         to[i].clear();
84     }
85     sz = 1;
86     node = pos = n = 0;
87     len[0] = inf;
88     for(int i = 0; i < s.size(); i++)
89         add_letter(s[i]);
90     correct(s.size());
91 }
92
93 void cutGeneralized(vector<int> &finishPoints){
94     for (int i = 0; i < sz; i++){

```

```

95         int init = f_pos[i];
96         int end = f_pos[i] + len[i] - 1;
97         int idx = lower_bound(finishPoints.begin(), finishPoints.end(), init) - finishPoints.begin();
98         if ((idx != finishPoints.size()) && (finishPoints[idx] <= end)){//Must be cut
99             len[i] = (finishPoints[idx] - f_pos[i] + 1);
100             to[i].clear();
101         }
102     }
103 }
104
105
106 void build_generalized(vector<string> &ss){
107     for (int i = 0; i < sz; i++){
108         to[i].clear();
109     }
110     sz = 1;
111     node = pos = n = 0;
112     len[0] = inf;
113     int sep = 256;
114     vector<int> finishPoints;
115     int next = 0;
116     for (int i = 0; i < ss.size(); i++){
117         for (int j = 0; j < ss[i].size(); j++){
118             add_letter(ss[i][j]);
119         }
120         next += ss[i].size();
121         finishPoints.push_back(next);
122         add_letter(sep++);
123         next++;
124     }
125     correct(next);
126     cutGeneralized(finishPoints);
127 }

```

12. Geometría

12.1. Convex hull

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

```

1  /// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ por cuestiones de tiempo
2  struct pt {
3      double x, y;
4  };
5  int orientation(pt a, pt b, pt c) {
6      double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
7      if (v < 0) return -1; // clockwise
8      if (v > 0) return +1; // counter-clockwise
9      return 0;
10 }
11 bool cw(pt a, pt b, pt c, bool include_collinear) {
12     int o = orientation(a, b, c);
13     return o < 0 || (include_collinear && o == 0);
14 }
15 bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }
16 void convex_hull(vector<pt>& a, bool include_collinear = false) {
17     pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {

```

```

18     return make_pair(a.y, a.x) < make_pair(b.y, b.x);
19 });
20 sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
21     int o = orientation(p0, a, b);
22     if (o == 0)
23         return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
24             < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
25     return o < 0;
26 });
27 if (include_collinear) {
28     int i = (int)a.size()-1;
29     while (i >= 0 && collinear(p0, a[i], a.back())) i--;
30     reverse(a.begin()+i+1, a.end());
31 }
32 vector<pt> st;
33 for (int i = 0; i < (int)a.size(); i++) {
34     while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
35         st.pop_back();
36     st.push_back(a[i]);
37 }
38 a = st;
39 }

```

13. Utilidades

13.1. Plantilla tree

```

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

```

13.2. Números aleatorios

mt19937_64 genera números de 64 bits.

```

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

```

13.3. Bitmask

```

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

```