



# Índice

<b>1. Referencia</b>	<b>3</b>	4.6. String Matching With Suffix Array . . . . .	16
<b>2. Estructuras</b>	<b>3</b>	4.7. LCP (Longest Common Prefix) . . . . .	16
2.1. RMQ (static) . . . . .	3	4.8. Corasick . . . . .	16
2.2. RMQ (dynamic) . . . . .	3	4.9. Suffix Automaton . . . . .	17
2.3. RMQ (lazy) . . . . .	4	4.10. Z Function . . . . .	18
2.4. RMQ (persistente) . . . . .	5	4.11. Palindrome . . . . .	18
2.5. Sliding window RMQ . . . . .	5	<b>5. Geometría</b>	<b>18</b>
2.6. Fenwick Tree . . . . .	5	5.1. Punto . . . . .	18
2.7. Union Find . . . . .	6	5.2. Orden radial de puntos . . . . .	19
2.8. Disjoint Intervals . . . . .	6	5.3. Line . . . . .	19
2.9. RMQ (2D) . . . . .	6	5.4. Segment . . . . .	19
2.10. Big Int . . . . .	7	5.5. Rectangle . . . . .	20
2.11. Modnum . . . . .	8	5.6. Polygon Area . . . . .	20
2.12. Treap para set . . . . .	9	5.7. Circle . . . . .	20
2.13. Treap para arreglo . . . . .	10	5.8. Point in Poly . . . . .	21
2.14. Convex Hull Trick . . . . .	10	5.9. Point in Convex Poly log(n) . . . . .	21
2.15. Convex Hull Trick (Dynamic) . . . . .	11	5.10. Convex Check CHECK . . . . .	21
2.16. Gain-Cost Set . . . . .	12	5.11. Convex Hull . . . . .	21
2.17. Set con índices . . . . .	12	5.12. Cut Polygon . . . . .	21
<b>3. Algoritmos</b>	<b>12</b>	5.13. Bresenham . . . . .	22
3.1. Longest Increasing Subsequence . . . . .	12	5.14. Rotate Matrix . . . . .	22
3.2. Alpha-Beta pruning . . . . .	13	5.15. Interseccion de Circulos en $n^3 \log(n)$ . . . . .	22
3.3. Mo's algorithm . . . . .	13	5.16. Cayley-Menger . . . . .	23
<b>4. Strings</b>	<b>13</b>	5.17. Heron's formula . . . . .	23
4.1. Hash . . . . .	13	<b>6. DP Opt</b>	<b>23</b>
4.2. Manacher . . . . .	14	6.1. Knuth . . . . .	23
4.3. KMP . . . . .	14	6.2. Chull . . . . .	23
4.4. Trie . . . . .	15	6.3. Divide & Conquer . . . . .	23
4.5. Suffix Array (largo, $n \log n$ ) . . . . .	15	<b>7. Matemática</b>	<b>24</b>
		7.1. Teoría de números . . . . .	24
		7.1.1. Funciones multiplicativas, función de Möbius . . . . .	24
		7.1.2. Teorema de Wilson . . . . .	24
		7.1.3. Pequeño teorema de Fermat . . . . .	24
		7.1.4. Teorema de Euler . . . . .	24
		7.2. Combinatoria . . . . .	24
		7.2.1. Burnside's lemma . . . . .	24
		7.2.2. Combinatorios . . . . .	24
		7.2.3. Lucas Theorem . . . . .	24
		7.2.4. Stirling . . . . .	24
		7.2.5. Bell . . . . .	25

7.2.6. Eulerian . . . . .	25	8.14. Euler Cycle . . . . .	40
7.2.7. Catalan . . . . .	25	8.15. Diametro árbol . . . . .	40
7.3. Sumatorias conocidas . . . . .	25	8.16. Chu-liu . . . . .	41
7.4. Ec. Característica . . . . .	25	8.17. Hungarian . . . . .	41
7.5. Aritmetica Modular . . . . .	25	8.18. Dynamic Conectivity . . . . .	42
7.6. Exp. de Numeros Mod. . . . .	25	<b>9. Flujo</b> . . . . .	<b>43</b>
7.7. Exp. de Matrices y Fibonacci en $\log(n)$ . . . . .	25	9.1. Dinic . . . . .	43
7.8. Matrices y determinante $O(n^3)$ . . . . .	26	9.2. Konig . . . . .	44
7.9. Primes and factorization . . . . .	26	9.3. Edmonds Karp's . . . . .	44
7.10. Euler's Phi . . . . .	26	9.4. Min-cost Max-flow . . . . .	45
7.11. Criba . . . . .	27	<b>10.Template</b> . . . . .	<b>46</b>
7.12. Funciones de primos . . . . .	27	<b>11.vimrc</b> . . . . .	<b>46</b>
7.13. Phollard's Rho - Miller-Rabin . . . . .	28	<b>12.misc</b> . . . . .	<b>46</b>
7.14. GCD . . . . .	28	<b>13.Ayudamemoria</b> . . . . .	<b>48</b>
7.15. LCM . . . . .	29		
7.16. Euclides extendido . . . . .	29		
7.17. Inversos . . . . .	29		
7.18. Ecuaciones diofánticas . . . . .	29		
7.19. Teorema Chino del Resto . . . . .	29		
7.20. Simpson . . . . .	29		
7.21. Fraction . . . . .	30		
7.22. Polinomio, Ruffini e interpolación de Lagrange . . . . .	30		
7.23. Ec. Lineales . . . . .	31		
7.24. FFT y NTT . . . . .	32		
7.25. Programación lineal: Simplex . . . . .	33		
7.26. Tablas y cotas (Primos, Divisores, Factoriales, etc) . . . . .	34		
<b>8. Grafos</b> . . . . .	<b>34</b>		
8.1. Teoremas y fórmulas . . . . .	34		
8.1.1. Teorema de Pick . . . . .	34		
8.1.2. Formula de Euler . . . . .	34		
8.2. Dijkstra . . . . .	35		
8.3. Bellman-Ford . . . . .	35		
8.4. Floyd-Warshall . . . . .	35		
8.5. Kruskal . . . . .	35		
8.6. Prim . . . . .	36		
8.7. 2-SAT + Tarjan SCC . . . . .	36		
8.8. Kosaraju . . . . .	37		
8.9. Articulation Points . . . . .	37		
8.10. Comp. Biconexas y Puentes . . . . .	38		
8.11. LCA + Climb . . . . .	38		
8.12. Heavy Light Decomposition . . . . .	39		
8.13. Centroid Decomposition . . . . .	39		

## 1. Referencia

Algoritmo	Parámetros	Función
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	<i>void</i> ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	<i>void</i> llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	<i>it</i> al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	<i>bool</i> esta elem en [f, l)
copy	f, l, resul	hace $resul+i=f+i \forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	<i>it</i> encuentra $i \in [f, l)$ tq. $i=elem$ , $pred(i)$ , $i \in [f2, l2)$
count, count_if	f, l, elem/pred	cuenta elem, $pred(i)$
search	f, l, f2, l2	busca $[f2, l2) \in [f, l)$
replace, replace_if	f, l, old / pred, new	cambia old / $pred(i)$ por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	$pred(i)$ ad, $!pred(i)$ atras
min_element, max_element	f, l, [comp]	<i>it</i> min, max de [f, l]
lexicographical_compare	f1, l1, f2, l2	<i>bool</i> con $[f1, l1)_i [f2, l2]$
next/prev_permutation	f, l	deja en [f, l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f, l), hace un heap de [f, l)
is_heap	f, l	<i>bool</i> es [f, l) un heap
accumulate	f, l, i, [op]	$T = \sum / oper$ de [f, l)
inner_product	f1, l1, f2, i	$T = i + [f1, l1) \cdot [f2, \dots)$
partial_sum	f, l, r, [op]	$r+i = \sum / oper$ de $[f, f+i) \forall i \in [f, l)$
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

## 2. Estructuras

### 2.1. RMQ (static)

```

1 // Dado un arreglo y una operacion asociativa idempotente:
2 // get(i, j) opera sobre el rango [i, j).
3 // Restriccion: 2^K > N. Usar [ ] para llenar
4 // el arreglo y luego build().
5 struct RMQ {
6     const static int K = ;
7     tipo vec[K][1 << K];
8     tipo &operator [](int p){ return vec[0][p]; }
9     tipo get(int i, int j){ // intervalo [i, j)
10         int p = 31 - __builtin_clz(j - i);
11         return min(vec[p][i], vec[p][j - (1 << p)]);
12     }
13     void build(int n){ // O(n log n)
14         int mp = 31 - __builtin_clz(n);
15         for(p, mp)
16             for(x, n - (1 << p))
17                 vec[p + 1][x] = min(vec[p][x], vec[p][x + (1 << p)]);
18     }
19 };

```

### 2.2. RMQ (dynamic)

```

1 // Dado un arreglo y una operacion asociativa con neutro:
2 // get(i, j) opera sobre el rango [i, j).
3 typedef int node; // Tipo de los nodos
4 #define MAXN 100000
5 #define operacion(x, y) min(x, y)
6 const int neutro = INT_MAX;
7 struct RMQ {
8     int sz;
9     node t[4*MAXN];
10     node &operator [](int p){ return t[sz + p]; }
11     void init(int n){ // O(n)
12         sz = 1 << (32 - __builtin_clz(n));
13         for(i, 2*sz) t[i] = neutro;
14     }
15     void updall(){ // O(n)
16         dforsn(i, 0, sz) t[i] = operacion(t[2*i], t[2*i + 1]);
17     }

```

```

18 node get(int i, int j){ return get(i, j, 1, 0, sz); }
19 node get(int i, int j, int n, int a, int b){ // 0(lg n)
20     if(j <= a || i >= b) return neutro;
21     if(i <= a && b <= j) return t[n];
22     int c = (a + b)/2;
23     return operacion(get(i, j, 2*n, a, c), get(i, j, 2*n + 1, c, b));
24 }
25 void set(int p, node val){ // 0(lg n)
26     for(p += sz; p > 0 && t[p] != val;){
27         t[p] = val, p /= 2;
28         val = operacion(t[p*2], t[p*2 + 1]);
29     }
30 }
31 } rmq;
32 // Uso:
33 cin >> n; rmq.init(n); forn(i, n) cin >> rmq[i]; rmq.updall();

```

## 2.3. RMQ (lazy)

```

1 // TODO: Las funciones pueden pasarse a traves de template. Quedara
2 // mejor sacar el struct tipo y reemplazar por todo en template?
3
4 const int N = 1e5, INF = 1e9;
5
6 struct TipoAlt {
7     int val;
8
9     TipoAlt(int _val=0) : val(_val) {}
10
11     static int neutro() { return 0; } // neutro alteracion
12     TipoAlt operator * (const int sz) {
13         return TipoAlt(val*sz);
14     }
15     TipoAlt& operator += (const TipoAlt &o) { val += o.val; return *this; }
16     ; // propaga alteracion, ejemplo suma
17 };
18
19 struct TipoNodo {
20     int val;
21
22     TipoNodo(int _val=0) : val(_val) {}
23
24     static int neutro() { return INF; } // neutro nodo

```

```

23     TipoNodo operator + (const TipoNodo &o) const { return min(val, o.
24         val); } // operacion nodo, ejemplo min
25
26     TipoNodo& operator += (const TipoAlt &o) { val += o.val; return *
27         this; } // aplica alteracion, ejemplo suma
28 };
29
30 // Dado un arreglo y una operacion asociativa con neutro:
31 // get(i, j) opera sobre el rango [i, j].
32 template <int N, class TNodo, class TAlt>
33 struct RMQ {
34     int sz;
35     TNodo t[4*N];
36     TAlt dirty[4*N];
37     TNodo &operator [] (int p){ return t[sz + p]; }
38     void init(int n) { // 0(n lg n)
39         sz = 1 << (32 - __builtin_clz(n));
40         forn(i, 2*sz) {
41             t[i] = TNodo::neutro();
42             dirty[i] = TAlt::neutro();
43         }
44     }
45     void push(int n, int a, int b){ // Propaga el dirty a sus hijos
46         if (dirty[n].val != TAlt::neutro().val){
47             t[n] += dirty[n]*(b - a); // Altera el nodo
48             if (n < sz){
49                 dirty[2*n] += dirty[n];
50                 dirty[2*n + 1] += dirty[n];
51             }
52             dirty[n] = TAlt::neutro();
53         }
54     }
55     TNodo get(int i, int j, int n, int a, int b){ // 0(lg n)
56         if (j <= a || i >= b) return TNodo::neutro();
57         push(n, a, b); // Corrige el valor antes de usarlo
58         if (i <= a && b <= j) return t[n];
59         int c = (a + b)/2;
60         return get(i, j, 2*n, a, c) + get(i, j, 2*n + 1, c, b);
61     }
62     TNodo get(int i, int j){ return get(i, j, 1, 0, sz); }
63     // Altera los valores en [i, j) con una alteracion de val
64     void modify(TAlt val, int i, int j, int n, int a, int b){ // 0(lg n)
65         push(n, a, b);
66         if (j <= a || i >= b) return;

```

```

64     if (i <= a && b <= j) {
65         dirty[n] += val;
66         push(n, a, b);
67         return;
68     }
69     int c = (a + b)/2;
70     modify(val, i, j, 2*n, a, c); modify(val, i, j, 2*n + 1, c, b);
71     t[n] = t[2*n] + t[2*n + 1];
72 }
73 void modify(TAlt val, int i, int j){ modify(val, i, j, 1, 0, sz); }
74 };
75
76 RMQ<N, TipoNodo, TipoAlt> rmq;

```

## 2.4. RMQ (persistente)

```

1  typedef int tipo;
2  tipo oper(const tipo &a, const tipo &b){
3      return a + b;
4  }
5  struct node {
6      tipo v; node *l, *r;
7      node(tipo v):v(v), l(NULL), r(NULL) {}
8      node(node *l, node *r) : l(l), r(r){
9          if(!l) v = r->v;
10         else if(!r) v = l->v;
11         else v = oper(l->v, r->v);
12     }
13 };
14 node *build (tipo *a, int tl, int tr) { // modificar para tomar tipo a
15     if(tl + 1 == tr) return new node(a[tl]);
16     int tm = (tl + tr) >> 1;
17     return new node(build(a, tl, tm), build(a, tm, tr));
18 }
19 node *upd(int pos, int new_val, node *t, int tl, int tr){
20     if(tl + 1 == tr) return new node(new_val);
21     int tm = (tl + tr) >> 1;
22     if(pos < tm) return new node(upd(pos, new_val, t->l, tl, tm), t->r);
23     else return new node(t->l, upd(pos, new_val, t->r, tm, tr));
24 }
25 tipo get(int l, int r, node *t, int tl, int tr){
26     if(l == tl && tr == r) return t->v;
27     int tm = (tl + tr) >> 1;

```

```

28     if(r <= tm) return get(l, r, t->l, tl, tm);
29     else if(l >= tm) return get(l, r, t->r, tm, tr);
30     return oper(get(l, tm, t->l, tl, tm), get(tm, r, t->r, tm, tr));
31 }

```

## 2.5. Sliding window RMQ

```

1  // Para max pasar less y -INF
2  template <class T, class Compare, T INF>
3  struct RMQ {
4      deque<T> d; queue<T> q;
5      void push(T v) {
6          while (!d.empty() && Compare()(d.back(), v)) d.pop_back();
7          d.pb(v);
8          q.push(v);
9      }
10
11     void pop() {
12         if (!d.empty() && d.front() == q.front()) d.pop_front();
13         q.pop();
14     }
15
16     T getMax() {
17         return d.empty() ? INF : d.front();
18     }
19
20     int size() {
21         return si(q);
22     }
23 };
24 RMQ<ll, less<ll>, -INF> rmq;

```

## 2.6. Fenwick Tree

```

1  // Para 2D: tratar cada columna como un Fenwick Tree,
2  // agregando un for anidado en cada operacion.
3  // Trucazo para 2D: si los elementos no se repiten,
4  // se puede usar un ordered set para memoria  $O(n \log^2(n))$ 
5  typedef ll tipo;
6  struct Fenwick {
7      static const int sz = (1 << 18) + 1;
8      tipo t[sz];
9      void adjust(int p, tipo v){ // p en [1, sz),  $O(\lg n)$ 
10         for(int i = p; i < sz; i += (i & -i)) t[i] += v;

```

```

11     }
12     tipo sum(int p){ // Suma acumulada en [1, p], O(lg n)
13         tipo s = 0;
14         for(int i = p; i; i -= (i & -i)) s += t[i];
15         return s;
16     }
17     tipo sum(int a, int b){ return sum(b) - sum(a - 1); }
18     int lower_bound(tipo v) { // Menor x con suma acumulada >= v, O(lg n)
19         )
20         int x = 0, d = sz-1;
21         if(v > t[d]) return sz;
22         for(; d; d >>= 1) if(t[x|d] < v) v -= t[x | = d];
23         return x+1;
24     }
};

```

## 2.7. Union Find

```

1 struct UF { // Operations take O(log*(n))
2     vi p,s;
3     UF(int n){ p.resize(n), iota(all(p), 0), s.assign(n, 1); }
4     int find(int i){
5         while (p[i] != i) p[i] = p[p[i]], i = p[i];
6         return i;
7     }
8     bool connected(int x, int y){ return find(x) == find(y); }
9     bool join(int x, int y){
10         x = find(x), y = find(y);
11         if (connected(x, y)) return false;
12
13         if (s[x] < s[y]) p[x] = y, s[y] += s[x];
14         else p[y] = x, s[x] += s[y];
15         return true;
16     }
17 };

```

## 2.8. Disjoint Intervals

```

1 // Guarda intervalos como [first, second]
2 // En caso de colision, los une en un solo intervalo
3 bool operator <(const pii &a, const pii &b){ return a.first < b.first; }
4 struct disjoint_intervals {
5     set<pii> segs;
6     void insert(pii v){ // O(lg n)

```

```

7         if(v.second - v.first == 0.0) return; // Cuidado!
8         set<pii>::iterator it, at;
9         at = it = segs.lower_bound(v);
10        if(at != segs.begin() && (--at)->second >= v.first){
11            v.first = at->first;
12            --it;
13        }
14        for(; it!=segs.end() && it->first <= v.second; segs.erase(it++))
15            v.second = max(v.second, it->second);
16        segs.insert(v);
17    }
18 };

```

## 2.9. RMQ (2D)

```

1 struct RMQ2D { // n filas, m columnas
2     int sz;
3     RMQ t[4*MAXN]; // t[i][j] = i fila, j columna
4     RMQ &operator [](int p){ return t[sz/2 + p]; }
5     void init(int n, int m){ // O(n*m)
6         sz = 1 << (32 - __builtin_clz(n));
7         forn(i, 2*sz) t[i].init(m);
8     }
9     void set(int i, int j, tipo val){ // O(lg(m)*lg(n))
10        for(i += sz; i > 0;){
11            t[i].set(j, val);
12            i /= 2;
13            val = operacion(t[i*2][j], t[i*2 + 1][j]);
14        }
15    }
16    tipo get(int i1, int j1, int i2, int j2){
17        return get(i1, j1, i2, j2, 1, 0, sz);
18    }
19    // O(lg(m)*lg(n)), rangos cerrado abierto
20    int get(int i1, int j1, int i2, int j2, int n, int a, int b){
21        if(i2 <= a || i1 >= b) return 0;
22        if(i1 <= a && b <= i2) return t[n].get(j1, j2);
23        int c = (a + b)/2;
24        return operacion(get(i1, j1, i2, j2, 2*n, a, c),
25                           get(i1, j1, i2, j2, 2*n + 1, c, b));
26    }
27 } rmq;
28 // Ejemplo para inicializar una matriz de n filas por m columnas

```

```

29 RMQ2D rmq; rmq.init(n, m);
30 forn(i, n) forn(j, m){
31     int v; cin >> v; rmq.set(i, j, v);
32 }

```

## 2.10. Big Int

```

1  #define BASE 10
2  #define LMAX 1000
3  int pad(int x){
4      x--; int c = 0;
5      while(x) x /= 10, c++;
6      return c;
7  }
8  const int PAD = pad(BASE);
9  struct bint {
10     int l;
11     ll n[LMAX];
12     bint(ll x = 0){
13         l = 1;
14         forn(i,LMAX){
15             if(x) l = i+1;
16             n[i] = x % BASE;
17             x /= BASE;
18         }
19     }
20     bint(string x){
21         int sz = si(x);
22         l = (sz-1)/PAD + 1;
23         fill(n, n+LMAX, 0);
24         ll r = 1;
25         forn(i,sz){
26             if(i % PAD == 0) r = 1;
27             n[i/PAD] += r*(x[sz-1-i]-'0');
28             r *= 10;
29         }
30     }
31     void out() const {
32         cout << n[l-1] << setfill('0');
33         dforn(i,l-1) cout << setw(PAD) << n[i];
34     }
35     void invar(){
36         fill(n+1, n+LMAX, 0);

```

```

37         while(l > 1 && !n[l-1]) l--;
38     }
39 };
40 bint operator+(const bint &a, const bint &b){
41     bint c;
42     c.l = max(a.l, b.l);
43     ll q = 0;
44     forn(i,c.l){
45         q += a.n[i] + b.n[i];
46         c.n[i] = q % BASE;
47         q /= BASE;
48     }
49     if(q) c.n[c.l++] = q;
50     c.invar();
51     return c;
52 }
53 pair<bint,bool> lresta(const bint &a, const bint &b){ // c = a - b
54     bint c;
55     c.l = max(a.l, b.l);
56     ll q = 0;
57     forn(i,c.l){
58         q += a.n[i] - b.n[i];
59         c.n[i] = (q + BASE) % BASE;
60         q = (q + BASE)/BASE - 1;
61     }
62     c.invar();
63     return {c,!q};
64 }
65 bint &operator --(bint &a, const bint &b){ return a = lresta(a, b).fst;
66 }
67 bint operator -(const bint &a, const bint &b){ return lresta(a, b).fst;
68 }
69 bool operator <(const bint &a, const bint &b){ return !lresta(a, b).snd;
70 }
71 bool operator <=(const bint &a, const bint &b){ return lresta(b, a).snd;
72 }
73 bool operator ==(const bint &a, const bint &b){ return a <= b && b <= a;
74 }
75 bool operator !=(const bint &a, const bint &b){ return a < b || b < a; }
76 bint operator *(const bint &a, ll b){
77     bint c;
78     ll q = 0;
79     forn(i,a.l){

```

```

75     q += a.n[i]*b;
76     c.n[i] = q % BASE;
77     q /= BASE;
78 }
79 c.l = a.l;
80 while(q){
81     c.n[c.l++] = q % BASE;
82     q /= BASE;
83 }
84 c.invar();
85 return c;
86 }
87 bint operator *(const bint &a, const bint &b){
88     bint c;
89     c.l = a.l+b.l;
90     fill(c.n, c.n+b.l, 0);
91     forn(i,a.l){
92         ll q = 0;
93         forn(j,b.l){
94             q += a.n[i]*b.n[j] + c.n[i+j];
95             c.n[i + j] = q % BASE;
96             q /= BASE;
97         }
98         c.n[i+b.l] = q;
99     }
100    c.invar();
101    return c;
102 }
103 pair<bint,ll> ldiv(const bint &a, ll b){ // c = a / b ; rm = a % b
104     bint c;
105     ll rm = 0;
106     dforn(i,a.l){
107         rm = rm*BASE + a.n[i];
108         c.n[i] = rm/b;
109         rm %= b;
110     }
111     c.l = a.l;
112     c.invar();
113     return {c,rm};
114 }
115 bint operator /(const bint &a, ll b){ return ldiv(a, b).fst; }
116 ll operator %(const bint &a, ll b){ return ldiv(a, b).snd; }
117 pair<bint,bint> ldiv(const bint &a, const bint &b){

```

```

118     bint c, rm = 0;
119     dforn(i,a.l){
120         if(rm.l == 1 && !rm.n[0]) rm.n[0] = a.n[i];
121         else {
122             dforn(j,rm.l) rm.n[j+1] = rm.n[j];
123             rm.n[0] = a.n[i], rm.l++;
124         }
125         ll q = rm.n[b.l]*BASE + rm.n[b.l-1];
126         ll u = q / (b.n[b.l-1] + 1);
127         ll v = q / b.n[b.l-1] + 1;
128         while(u < v-1){
129             ll m = (u + v)/2;
130             if(b*m <= rm) u = m;
131             else v = m;
132         }
133         c.n[i] = u, rm -= b*u;
134     }
135     c.l = a.l;
136     c.invar();
137     return {c,rm};
138 }
139 bint operator /(const bint &a, const bint &b){ return ldiv(a, b).fst; }
140 bint operator %(const bint &a, const bint &b){ return ldiv(a, b).snd; }
141 bint gcd(bint a, bint b){
142     while(b != bint(0)){
143         bint r = a % b;
144         a = b, b = r;
145     }
146     return a;
147 }

```

## 2.11. Modnum

```

1 struct num {
2     int a;
3     num(int _a = 0) : a(_a) {} // o tambien num(ll _a=0) : a((_a%M+M)%M)
4     {}
5     operator int(){ return a; }
6     num operator +(num b){ return a+b.a >= M ? a+b.a-M : a+b.a; }
7     num operator -(num b){ return a-b.a < 0 ? a-b.a+M : a-b.a; }
8     num operator *(num b){ return int((ll)a*b.a % M); }
9     num operator ~(num e){
10         if(!e.a) return 1;

```



```

10     num q = (*this)^num(e.a/2);
11     return e.a & 1 ? q*q*(this) : q*q;
12 }
13 num operator ++(int x){ return a++; }
14 };
15 int norm(int x){ return x < 0 ? int(x % M + M) : int(x % M); }
16 num inv(num x){ return x^num(M-2); } // M must be prime
17 num operator /(num a, num b){ return a*inv(b); }
18 num neg(num x){ return x.a ? -x.a+M : 0; }
19 istream& operator >>(istream &i, num &x){ i >> x.a; return i; }
20 ostream& operator <<(ostream &o, const num &x){ o << x.a; return o; }
21 // Cast integral values to num in arithmetic expressions!

```

## 2.12. Treap para set

```

1  typedef int Key;
2  typedef struct node *pnode;
3  struct node {
4      Key key;
5      int prior, size;
6      pnode l, r;
7      node(Key key = 0): key(key), prior(rand()), size(1), l(0), r(0) {}
8  };
9  static int size(pnode p){ return p ? p->size : 0; }
10 void push(pnode p){
11     // modificar y propagar el dirty a los hijos aca (para lazy)
12 }
13 // Update function and size from children's Value
14 void pull(pnode p){ // recalcular valor del nodo aca (para rmq)
15     p->size = 1 + size(p->l) + size(p->r);
16 }
17 //junta dos arreglos
18 pnode merge(pnode l, pnode r){
19     if(!l || !r) return l ? l : r;
20     push(l), push(r);
21     pnode t;
22     if(l->prior < r->prior) l->r = merge(l->r, r), t = l;
23     else r->l = merge(l, r->l), t = r;
24     pull(t);
25     return t;
26 }
27 //parte el arreglo en dos, l < key <= r
28 void split(pnode t, Key key, pnode &l, pnode &r){

```

```

29     if(!t) return void(l = r = 0);
30     push(t);
31     if(key <= t->key) split(t->l, key, l, t->l), r = t;
32     else split(t->r, key, t->r, r), l = t;
33     pull(t);
34 }
35
36 void erase(pnode &t, Key key){
37     if(!t) return;
38     push(t);
39     if(key == t->key) t = merge(t->l, t->r);
40     else if(key < t->key) erase(t->l, key);
41     else erase(t->r, key);
42     if(t) pull(t);
43 }
44
45 ostream& operator<<(ostream &out, const pnode &t){
46     if(!t) return out;
47     return out << t->l << t->key << ' ' << t->r;
48 }
49 pnode find(pnode t, Key key){
50     if(!t) return 0;
51     if(key == t->key) return t;
52     if(key < t->key) return find(t->l, key);
53     return find(t->r, key);
54 }
55 struct treap {
56     pnode root;
57     treap(pnode root = 0): root(root) {}
58     int size(){ return ::size(root); }
59     void insert(Key key){
60         pnode t1, t2; split(root, key, t1, t2);
61         t1 = ::merge(t1, new node(key));
62         root = ::merge(t1, t2);
63     }
64     void erase(Key key1, Key key2){
65         pnode t1, t2, t3;
66         split(root, key1, t1, t2);
67         split(t2, key2, t2, t3);
68         root = merge(t1, t3);
69     }
70     void erase(Key key){ ::erase(root, key); }
71     pnode find(Key key){ return ::find(root, key); }

```

```

72     Key &operator[](int pos){ return find(pos)->key; }//ojito
73 };
74 treap merge(treap a, treap b){ return treap(merge(a.root, b.root)); }

```

## 2.13. Treap para arreglo

```

1  typedef int Value; // pii(profundidad, nodo)
2  typedef struct node *pnode;
3  struct node {
4      Value val, mini;
5      int dirty;
6      int prior, size;
7      pnode l, r, parent;
8      node(Value val):val(val), mini(val), dirty(0), prior(rand()), size
          (1), l(0), r(0), parent(0) {}
9  };
10 static int size(pnode p){ return p ? p->size : 0; }
11 void push(pnode p){ // propagar dirty a los hijos (aca para lazy)
12     p->val.first += p->dirty;
13     p->mini.first += p->dirty;
14     if(p->l) p->l->dirty += p->dirty;
15     if(p->r) p->r->dirty += p->dirty;
16     p->dirty = 0;
17 }
18 static Value mini(pnode p){ return p ? push(p), p->mini : pii(1e9, -1);
    }
19 // Update function and size from children's Value
20 void pull(pnode p){ // recalcular valor del nodo aca (para rmq)
21     p->size = 1 + size(p->l) + size(p->r);
22     p->mini = min(min(p->val, mini(p->l)), mini(p->r)); //operacion del rmq
    !
23     p->parent = 0;
24     if(p->l) p->l->parent = p;
25     if(p->r) p->r->parent = p;
26 }
27 //junta dos arreglos
28 pnode merge(pnode l, pnode r){
29     if(!l || !r) return l ? l : r;
30     push(l), push(r);
31     pnode t;
32     if(l->prior < r->prior) l->r=merge(l->r, r), t = l;
33     else r->l=merge(l, r->l), t = r;
34     pull(t);

```

```

35     return t;
36 }
37 //parte el arreglo en dos, si(l)==tam
38 void split(pnode t, int tam, pnode &l, pnode &r){
39     if(!t) return void(l = r = 0);
40     push(t);
41     if(tam <= size(t->l)) split(t->l, tam, l, t->l), r = t;
42     else split(t->r, tam - 1 - size(t->l), t->r, r), l = t;
43     pull(t);
44 }
45 pnode at(pnode t, int pos){
46     if(!t) exit(1);
47     push(t);
48     if(pos == size(t->l)) return t;
49     if(pos < size(t->l)) return at(t->l, pos);
50     return at(t->r, pos - 1 - size(t->l));
51 }
52 int getpos(pnode t){ // inversa de at
53     if(!t->parent) return size(t->l);
54     if(t == t->parent->l) return getpos(t->parent) - size(t->r) - 1;
55     return getpos(t->parent) + size(t->l) + 1;
56 }
57 void split(pnode t, int i, int j, pnode &l, pnode &m, pnode &r){
58     split(t, i, l, t), split(t, j-i, m, r);
59 }
60 Value get(pnode &p, int i, int j){ // like rmq
61     pnode l, m, r;
62     split(p, i, j, l, m, r);
63     Value ret = mini(m);
64     p = merge(l, merge(m, r));
65     return ret;
66 }
67 void print(const pnode &t){ // for debugging
68     if(!t) return;
69     push(t);
70     print(t->l);
71     cout << t->val.first << '␣';
72     print(t->r);
73 }

```

## 2.14. Convex Hull Trick

```

1  struct Line{tipo m,h;};

```

```

2  tipo inter(Line a, Line b){
3      tipo x=b.h-a.h, y=a.m-b.m;
4      return x/y+(x%y?!((x>0)^(y>0)):0);//===ceil(x/y)
5  }
6  struct CHT {
7      vector<Line> c;
8      bool mx;
9      int pos;
10     CHT(bool mx=0):mx(mx),pos(0){};//mx=1 si las query devuelven el max
11     inline Line acc(int i){return c[c[0].m>c.back().m? i : si(c)-1-i];}
12     inline bool irre(Line x, Line y, Line z){
13         return c[0].m>z.m? inter(y, z) <= inter(x, y)
14             : inter(y, z) >= inter(x, y);
15     }
16     void add(tipo m, tipo h) {//O(1), los m tienen que entrar ordenados
17         if(mx) m*=-1, h*=-1;
18         Line l=(Line){m, h};
19         if(si(c) && m==c.back().m) { l.h=min(h, c.back().h), c.pop_back
20             (); if(pos) pos--; }
21         while(si(c)>=2 && irre(c[si(c)-2], c[si(c)-1], l)) { c.pop_back
22             (); if(pos) pos--; }
23         c.pb(l);
24     }
25     inline bool fbin(tipo x, int m) {return inter(acc(m), acc(m+1))>x;}
26     tipo eval(tipo x){
27         int n = si(c);
28         //query con x no ordenados O(lgn)
29         int a=-1, b=n-1;
30         while(b-a>1) { int m = (a+b)/2;
31             if(fbin(x, m)) b=m;
32             else a=m;
33         }
34         return (acc(b).m*x+acc(b).h)*(mx?-1:1);
35         //query O(1)
36         while(pos>0 && fbin(x, pos-1)) pos--;
37         while(pos<n-1 && !fbin(x, pos)) pos++;
38         return (acc(pos).m*x+acc(pos).h)*(mx?-1:1);
39     }
40 } ch;
41 struct CHTBruto {
42     vector<Line> c;
43     bool mx;
44     CHTBruto(bool mx=0):mx(mx){};//mx=si las query devuelven el max o el

```

```

min
43 void add(tipo m, tipo h) {
44     Line l=(Line){m, h};
45     c.pb(l);
46 }
47 tipo eval(tipo x){
48     tipo r=c[0].m*x+c[0].h;
49     forn(i, si(c)) if(mx) r=max(r, c[i].m*x+c[i].h);
50     else r=min(r, c[i].m*x+c[i].h);
51     return r;
52 }
53 } chb;

```

## 2.15. Convex Hull Trick (Dynamic)

```

1 struct Line {
2     tint m, b;
3     mutable multiset<Line>::iterator it;
4     const Line *succ(multiset<Line>::iterator it) const;
5     bool operator<(const Line& rhs) const {
6         if (rhs.b != is_query) return m < rhs.m;
7         const Line *s=succ(it);
8         if(!s) return 0;
9         tint x = rhs.m;
10        return b - s->b < (s->m - m) * x;
11    }
12 };
13 struct HullDynamic : public multiset<Line>{ // will maintain upper hull
14     for maximum
15     bool bad(iterator y) {
16         iterator z = next(y);
17         if (y == begin()) {
18             if (z == end()) return 0;
19             return y->m == z->m && y->b <= z->b;
20         }
21         iterator x = prev(y);
22         if (z == end()) return y->m == x->m && y->b <= x->b;
23         return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m);
24     }
25     iterator next(iterator y){return ++y;}
26     iterator prev(iterator y){return --y;}
27     void insert_line(tint m, tint b) {

```

```

27     iterator y = insert((Line) { m, b });
28     y->it=y;
29     if (bad(y)) { erase(y); return; }
30     while (next(y) != end() && bad(next(y))) erase(next(y));
31     while (y != begin() && bad(prev(y))) erase(prev(y));
32 }
33 tint eval(tint x) {
34     Line l = *lower_bound((Line) { x, is_query });
35     return l.m * x + l.b;
36 }
37 }h;
38 const Line *Line::succ(multiset<Line>::iterator it) const{
39     return (++it==h.end())? NULL : &*it;};

```

## 2.16. Gain-Cost Set

```

1 //esta estructura mantiene pairs(beneficio, costo)
2 //de tal manera que en el set quedan ordenados
3 //por beneficio Y COSTO creciente. (va borrando los que no son optimos)
4 struct V{
5     int gain, cost;
6     bool operator<(const V &b)const{return gain<b.gain;}
7 };
8 set<V> s;
9 void add(V x){
10     set<V>::iterator p=s.lower_bound(x);//primer elemento mayor o igual
11     if(p!=s.end() && p->cost <= x.cost) return;//ya hay uno mejor
12     p=s.upper_bound(x);//primer elemento mayor
13     if(p!=s.begin()){//borro todos los peores (<=beneficio y >=costo)
14         --p;//ahora es ultimo elemento menor o igual
15         while(p->cost >= x.cost){
16             if(p==s.begin()){s.erase(p); break;}
17             s.erase(p--);
18         }
19     }
20     s.insert(x);
21 }
22 int get(int gain){//minimo costo de obtener tal ganancia
23     set<V>::iterator p=s.lower_bound((V){gain, 0});
24     return p==s.end()? INF : p->cost;};

```

## 2.17. Set con índices

```

1 #include <cassert>

```

```

2
3 #include <ext/pb_ds/assoc_container.hpp>
4 #include <ext/pb_ds/tree_policy.hpp>
5 using namespace __gnu_pbds;
6 typedef tree<int,null_type,less<int>,//key,mapped type, comparator
7     rb_tree_tag,tree_order_statistics_node_update> Set;
8 //find_by_order(i) devuelve iterador al i-esimo elemento
9 //order_of_key(k): devuelve la pos del lower bound de k
10 //Ej: 12, 100, 505, 1000, 10000.
11 //order_of_key(10) == 0, order_of_key(100) == 1,
12 //order_of_key(707) == 3, order_of_key(9999999) == 5

```

## 3. Algoritmos

### 3.1. Longest Increasing Subsequence

```

1 const int MAXN = 1e5+10, INF = 1e8;
2
3 //Para non-increasing, cambiar comparaciones y revisar busq binaria
4 //Given an array, paint it in the least number of colors so that each
5 //color turns to a non-increasing subsequence.
6 //Solution:Min number of colors=Length of the longest increasing
7 //subsequence
8 int N, a[MAXN]; //secuencia y su longitud
9 pii d[MAXN+1]; //d[i]=ultimo valor de la subsecuencia de tamaño i
10 int p[MAXN]; //padres
11 vector<int> R; //respuesta
12 void rec(int i){
13     if(i==1) return;
14     R.pb(a[i]);
15     rec(p[i]);
16 }
17 int lis(){//O(nlogn)
18     d[0] = pii(-INF, -1); for(i, N) d[i+1]=pii(INF, -1);
19     for(i, N){
20         int j = upper_bound(d, d+N+1, pii(a[i], INF))-d;
21         if (d[j-1].first < a[i]&&a[i] < d[j].first){ // check < por <= en d
22             j-1;
23             p[i]=d[j-1].second;
24             d[j] = pii(a[i], i);
25         }
26     }
27     R.clear();
28 }

```

```

25 dforsn(i, 0, N+1) if(d[i].first!=INF){
26     rec(d[i].second); //reconstruir
27     reverse(R.begin(), R.end());
28     return i; //longitud
29 }
30 return 0;
31 }

```

### 3.2. Alpha-Beta pruning

```

1 ll alphabeta(State &s, bool player = true, int depth = 1e9, ll alpha = -
  INF, ll beta = INF) { //player = true -> Maximiza
2     if(s.isFinal()) return s.score;
3     //~ if (!depth) return s.heuristic();
4     vector<State> children;
5     s.expand(player, children);
6     int n = children.size();
7     forn(i, n) {
8         ll v = alphabeta(children[i], !player, depth-1, alpha, beta);
9         if(!player) alpha = max(alpha, v);
10        else beta = min(beta, v);
11        if(beta <= alpha) break;
12    }
13    return !player ? alpha : beta;

```

### 3.3. Mo's algorithm

```

1 int n,sq;
2 struct Qu{//queries [l, r]
3     //intervalos cerrado abiertos !!! importante!!
4     int l, r, id;
5 }qs[MAXN];
6 int ans[MAXN], curans; //ans[i]=ans to ith query
7 bool bymos(const Qu &a, const Qu &b){
8     if(a.l/sq!=b.l/sq) return a.l<b.l;
9     return (a.l/sq&1? a.r<b.r : a.r>b.r;
10 }
11 void mos(){//(n+q)*sqrt(n)*(0(add()))+0(remove()))
12     forn(i, t) qs[i].id=i;
13     sort(qs, qs+t, bymos);
14     int cl=0, cr=0;
15     sq=sqrt(n);
16     curans=0;
17     forn(i, t){ //intervalos cerrado abiertos !!! importante!!

```

```

18     Qu &q=qs[i];
19     while(cl>q.l) add(--cl);
20     while(cr<q.r) add(cr++);
21     while(cl<q.l) remove(cl++);
22     while(cr>q.r) remove(--cr);
23     ans[q.id]=curans;
24 }
25 }

```

## 4. Strings

### 4.1. Hash

```

1 mt19937 rng;
2 struct basicHashing {
3     int mod, mul;
4
5     bool prime(int n) {
6         for (int d = 2; d*d <= n; d++) if (n%d == 0) return false;
7         return true;
8     }
9
10    void randomize() {
11        rng.seed(chrono::steady_clock::now().time_since_epoch().count())
12        ;
13        mod = uniform_int_distribution<>(0, (int) 5e8)(rng) + 1e9;
14        while (!prime(mod)) mod++;
15        mul = uniform_int_distribution<>(2,mod-2)(rng);
16    }
17
18    basicHashing() { randomize(); }
19
20    vi h, pot;
21    void process(const string &s) {
22        h.resize(si(s)+1);
23        pot.resize(si(s)+1);
24        h[0] = 0; forn(i,si(s)) h[i+1] = (((ll)h[i] * mul) + s[i]) % mod;
25        ;
26        pot[0] = 1; forn(i,si(s)) pot[i+1] = (ll)pot[i] * mul % mod;
27    }
28
29    int hash(int i, int j) {
30        int res = h[j] - (ll)h[i] * pot[j-i] % mod;

```

```

29     if (res < 0) res += mod;
30     return res;
31 }
32
33 int hash(const string &s) {
34     int res = 0;
35     for (char c : s) res = ((ll)res * mul + c) % mod;
36     return res;
37 }
38
39 };
40
41 struct hashing {
42     basicHashing h1, h2;
43
44     void process(const string &s) {
45         h1.process(s);
46         h2.process(s);
47     }
48
49     pii hash(int i, int j) {
50         return {h1.hash(i, j), h2.hash(i, j)};
51     }
52
53     pii hash(const string &s) {
54         return {h1.hash(s), h2.hash(s)};
55     }
56 };
57
58 // uso:
59 hashing h;
60 h.process(s);
61 auto x = h.hash(0, n);

```

## 4.2. Manacher

**Definición:** permite calcular todas las substrings de una string  $s$  que son palíndromos de longitud impar (y par, ver *observación*). Para ello, mantiene un arreglo  $len$  tal que  $len[i]$  almacena la longitud del palíndromo impar maximal con centro en  $i$ .

**Explicación algoritmo:** muy similar al algoritmo para calcular la función  $Z$ . Mantiene el palíndromo que termina más a la derecha entre todos los palíndromos ya detectados. Para calcular  $len[i]$ , utiliza la información ya calculada si  $i$  está dentro de  $[l, r]$ , y luego corre el algoritmo trivial.

**Observación:** para calcular los palíndromos de longitud par, basta con utilizar el

mismo algoritmo con la cadena  $s_0\#s_1\#\dots\#s_{n-1}$ .

```

1 vi pal_array(string s)
2 {
3     int n = si(s);
4     s = "@" + s + "$";
5
6     vi len(n + 1);
7     int l = 1, r = 1;
8
9     forsn(i, 1, n+1) {
10         len[i] = min(r - i, len[l + (r - i)]);
11
12         while (s[i - len[i]] == s[i + len[i]]) len[i]++;
13
14         if (i + len[i] > r) l = i - len[i], r = i + len[i];
15     }
16
17     len.erase(begin(len));
18     return len;
19 }

```

## 4.3. KMP

```

1 // pref[i] = max borde de s[0..i] = failure function al intentar
2 // matchear con s[i+1]
3 vi prefix_function(string &s) {
4     int n = si(s); vi pi(n);
5     forsn(i, 1, n) {
6         int j = pi[i-1];
7         while (j > 0 && s[i] != s[j]) j = pi[j-1];
8         if (s[i] == s[j]) j++;
9         pi[i] = j;
10    }
11    return pi;
12 }
13
14 vi find_occurrences(string &s, string &t) { //apariciones de t en s
15     vi pre = prefix_function(t), res;
16     int n = si(s), m = si(t), j = 0;
17     forn(i, n) {
18         while (j > 0 && s[i] != t[j]) j = pre[j-1];
19         if (s[i] == t[j]) j++;

```

```

19     if (j == m) {
20         res.pb(i-j+1);
21         j = pre[j-1];
22     }
23 }
24 return res;
25 }
26
27 // aut[i][c] = (next o failure function) al intentar matchear s[i] con c
28 void compute_automaton(string s, vector<vi>& aut) {
29     s += '#'; // separador!
30     int n = si(s);
31     vi pi = prefix_function(s);
32     aut.assign(n, vi(26));
33
34     forn(i, n) forn(c, 26)
35         if (i > 0 && 'a' + c != s[i])
36             aut[i][c] = aut[pi[i-1]][c];
37         else
38             aut[i][c] = i + ('a' + c == s[i]);
39 }

```

#### 4.4. Trie

```

1 struct trie {
2     int p = 0, w = 0;
3     map<char, trie*> c;
4     trie(){}
5     void add(const string &s){
6         trie *x = this;
7         forn(i, si(s)){
8             if(!x->c.count(s[i])) x->c[s[i]] = new trie();
9             x = x->c[s[i]];
10            x->p++;
11        }
12        x->w++;
13    }
14    int find(const string &s){
15        trie *x = this;
16        forn(i, si(s)){
17            if(x->c.count(s[i])) x = x->c[s[i]];
18            else return 0;
19        }

```

```

20        return x->w;
21    }
22    void erase(const string &s){
23        trie *x = this, *y;
24        forn(i, si(s)){
25            if(x->c.count(s[i])) y = x->c[s[i]], y->p--;
26            else return;
27            if(!y->p){
28                x->c.erase(s[i]);
29                return;
30            }
31            x = y;
32        }
33        x->w--;
34    }
35    void print(string tab = "") {
36        for(auto &i : c) {
37            cerr << tab << i.fst << endl;
38            i.snd->print(tab + "--");
39        }
40    }
41 };

```

#### 4.5. Suffix Array (largo, nlogn)

```

1 const int MAXN = 1e3+10;
2 #define rBOUND(x) (x<n? r[x] : 0)
3 //sa will hold the suffixes in order.
4 int sa[MAXN], r[MAXN], n;
5 string s; //input string, n=si(s)
6
7 int f[MAXN], tmpsa[MAXN];
8 void countingSort(int k){
9     fill(f, f+MAXN, 0);
10    forn(i, n) f[rBOUND(i+k)]++;
11    int sum=0;
12    forn(i, max(255, n)){
13        int t=f[i]; f[i]=sum; sum+=t;}
14    forn(i, n)
15        tmpsa[f[rBOUND(sa[i]+k)]++] = sa[i];
16    memcpy(sa, tmpsa, sizeof(sa));
17 }
18 void constructsa(){//O(n log n)

```



```

19 n=si(s);
20 forn(i, n) sa[i]=i, r[i]=s[i];
21 for(int k=1; k<n; k<=<=1){
22     countingSort(k), countingSort(0);
23     int rank, tmpr[MAXN];
24     tmpr[sa[0]]=rank=0;
25     forsn(i, 1, n)
26         tmpr[sa[i]]=(r[sa[i]]==r[sa[i-1]] && r[sa[i]+k]==r[sa[i-1]+k])?
            rank : ++rank;
27     memcpy(r, tmpr, sizeof(r));
28     if(r[sa[n-1]]==n-1) break;
29 }
30 }
31 void print(){//for debug
32     forn(i,n){
33         cout << i << ' ';
34         s.substr(sa[i], s.find( '$', sa[i])-sa[i]) << endl;
35     }
36 }
37
38 //returns (lowerbound, upperbound) of the search

```

#### 4.6. String Matching With Suffix Array

```

1 //returns (lowerbound, upperbound) of the search
2 pii stringMatching(string P){ //O(si(P)lgn)
3     int lo=0, hi=n-1, mid=lo;
4     while(lo<hi){
5         mid=(lo+hi)/2;
6         int res=s.compare(sa[mid], si(P), P);
7         if(res>=0) hi=mid;
8         else lo=mid+1;
9     }
10    if(s.compare(sa[lo], si(P), P)!=0) return pii(-1, -1);
11    pii ans; ans.first=lo;
12    lo=0, hi=n-1, mid;
13    while(lo<hi){
14        mid=(lo+hi)/2;
15        int res=s.compare(sa[mid], si(P), P);
16        if(res>0) hi=mid;
17        else lo=mid+1;
18    }
19    if(s.compare(sa[hi], si(P), P)!=0) hi--;

```

```

20 // para verdadero upperbound sumar 1
21 ans.second=hi;
22 return ans;

```

#### 4.7. LCP (Longest Common Prefix)

```

1
2 //Calculates the LCP between consecutives suffixes in the Suffix Array.
3 //LCP[i] is the length of the LCP between sa[i] and sa[i-1]
4 int LCP[MAXN], phi[MAXN], PLCP[MAXN];
5 void computeLCP(){//O(n)
6     phi[sa[0]]=-1;
7     forsn(i,1,n) phi[sa[i]]=sa[i-1];
8     int L=0;
9     forn(i,n){
10         if (phi[i]==-1) {PLCP[i]=0; continue;}
11         while (s[i+L]==s[phi[i]+L]) L++;
12         PLCP[i]=L;
13         L=max(L-1, 0);
14     }
15     forn(i,n) LCP[i]=PLCP[sa[i]];

```

#### 4.8. Corasick

```

1 struct trie{
2     map<char, trie> next;
3     trie* tran[256]; //transiciones del automata
4     int idhoja, sihoja; //id de la hoja o 0 si no lo es
5     //link lleva al sufijo mas largo, nxthoja lleva al mas largo pero que
        es hoja
6     trie *padre, *link, *nxthoja;
7     char pch; //caracter que conecta con padre
8     trie(): tran(), idhoja(), padre(), link() {}
9     void insert(const string &s, int id=1, int p=0){ //id>0!!!
10         if(p<si(s)){
11             trie &ch=next[s[p]];
12             tran[(int)s[p]]=&ch;
13             ch.padre=this, ch.pch=s[p];
14             ch.insert(s, id, p+1);
15         }
16         else idhoja=id, sihoja=si(s);
17     }
18     trie* get_link() {
19         if(!link){

```



```

20     if(!padre) link=this;//es la raiz
21     else if(!padre->padre) link=padre;//hijo de la raiz
22     else link=padre->get_link()->get_tran(pch);
23 }
24 return link; }
25 trie* get_tran(int c) {
26     if(!tran[c]) tran[c] = !padre? this : this->get_link()->get_tran(c);
27     return tran[c]; }
28 trie *get_nxthoja(){
29     if(!nxthoja) nxthoja = get_link()->idhoja? link : link->nxthoja;
30     return nxthoja; }
31 void print(int p){
32     if(idhoja) cout << "found_" << idhoja << "_at_position_" << p-
        sihoja << endl;
33     if(get_nxthoja()) get_nxthoja()->print(p); }
34 void matching(const string &s, int p=0){
35     print(p); if(p<si(s)) get_tran(s[p])->matching(s, p+1); }
36 }tri;

```

## 4.9. Suffix Automaton

**Definición** Un suffix automaton  $A$  es un autómata minimal que reconoce los sufijos de una cadena  $s$ .

### Conceptos importantes

- $A$  reconoce a una cadena  $s$  si comenzando desde el nodo inicial llegamos a un terminal.
- Llamamos *endpoints* de una cadena  $u$  en  $s$  a las posiciones  $i$  tal que  $u = s_{i-|u|, i-1}$
- Dos substrings  $u$  y  $v$  de  $s$  son *equivalentes* si recorrer el autómata con  $u$  y con  $v$  nos lleva al mismo nodo. Esto equivale a que los endpoints de  $u$  y de  $v$  en  $s$  sean los mismos.
- Los nodos del autómata corresponden a las *clases de equivalencia* bajo la relación anterior.
- Si tomamos la *cadena minimal* de una clase y removemos la primera letra, nos movemos al padre en el *suffix tree* de  $s'$ . Esto se señala mediante *suffix links*.
- Si tomamos la *cadena maximal* de una clase y agregamos una letra al principio, nos movemos a otra clase de equivalencia, recorriendo un *suffix link* en sentido inverso, es decir, nos movemos a los hijos en el suffix tree de  $s'$ .

### Problemas clásicos

- Determinar si  $w$  es subcadena de  $s$ : simplemente correr el autómata.

- Determinar si  $w$  es sufijo de  $s$ : correr el autómata y ver si caemos en un terminal.
- Contar cantidad de subcadenas distintas de  $s$ : esto es igual a la cantidad de caminos en el autómata y se calcula mediante una DP.
- Contar cantidad de apariciones de  $w$  en  $s$ : correr autómata con  $w$ . Llamemos  $u$  al nodo en el que terminamos, la cantidad de apariciones es la cantidad de caminos en  $A$  que comienzan en  $u$  y llegan a un terminal.
- Encontrar dónde aparece  $w$  por primera vez en  $s$ : correr autómata con  $w$ . Llamemos  $u$  al nodo en el que terminamos, esto equivale a calcular el camino más largo del autómata a partir del nodo  $u$ .
- Encontrar las posiciones de todas las apariciones de  $w$  en  $s$ : agregar '\$á  $s$ ', encontrar el nodo  $u$  en el que finaliza  $w$ , armar el *suffix tree*, encontrar todas las hojas en el subárbol con raíz en  $u$ , cada hoja corresponde a un prefijo y por lo tanto a una aparición.

```

1 struct state {
2     int len, link;
3     map<char,int> next;
4     state() { }
5 };
6 const int MAXLEN = 1e5+10;
7 state st[MAXLEN*2];
8 int sz, last;
9 void sa_init() {
10     for(i,sz) st[i].next.clear();
11     sz = last = 0;
12     st[0].len = 0;
13     st[0].link = -1;
14     ++sz;
15 }
16 // Es un DAG de una sola fuente y una sola hoja
17 // cantidad de endpos = cantidad de apariciones = cantidad de caminos de
    la clase al nodo terminal
18 // cantidad de miembros de la clase = st[v].len-st[st[v].link].len (v>0)
    = caminos del inicio a la clase
19 // El arbol de los suffix links es el suffix tree de la cadena invertida
    . La string de la arista link(v)->v son los caracteres que difieren
20 void sa_extend (char c) {
21     int cur = sz++;
22     st[cur].len = st[last].len + 1;
23     // en cur agregamos la posicion que estamos extendiendo

```

```

24 //podria agregar tambien un identificador de las cadenas a las cuales
    pertenece (si hay varias)
25 int p;
26 for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link) // modificar
    esta linea para hacer separadores unicos entre varias cadenas (c
    ==','$')
27     st[p].next[c] = cur;
28 if (p == -1)
29     st[cur].link = 0;
30 else {
31     int q = st[p].next[c];
32     if (st[p].len + 1 == st[q].len)
33         st[cur].link = q;
34     else {
35         int clone = sz++;
36         // no le ponemos la posicion actual a clone sino indirectamente
            por el link de cur
37         st[clone].len = st[p].len + 1;
38         st[clone].next = st[q].next;
39         st[clone].link = st[q].link;
40         for (; p!=-1 && st[p].next.count(c) && st[p].next[c]==q; p=st[p].
            link)
41             st[p].next[c] = clone;
42         st[q].link = st[cur].link = clone;
43     }
44 }
45 last = cur;
46 }

```

## 4.10. Z Function

**Definición** La función Z para una string  $s$  de longitud  $n$  es un arreglo  $a$  de la misma longitud tal que  $a[i]$  es la *máxima cantidad de caracteres* comenzando desde la posición  $i$  que coinciden con los primeros caracteres de  $s$ . Es decir, es el *máximo prefijo común*. **Observación**  $z[0]$  no está bien definido, pero se asume igual a 0.

**Algoritmo** La idea es mantener el máximo *match* (es decir, el segmento  $[l, r]$  con máximo  $r$  tal que se sabe que  $s[0..r-l] = s[l..r]$ ).

Siendo  $i$  el índice actual (del que queremos calcular la función Z), el algoritmo se divide en dos casos:

- $i \leq r$ : la posición está dentro del *match actual*, por lo que se puede utilizar como aproximación inicial  $z[i] = \min(r-i+1, z[i-l])$ , y luego correr el *algoritmo trivial*.

## Problemas clásicos

- Buscar una subcadena: concatenamos  $p$  con  $t$  (utilizando un separador). Hay una aparición si la función Z matcheó tantos caracteres como la longitud de  $p$ .

```

1 int z[N]; // z[i] = i==0 ? 0 : max k tq s[0,k] match with s[i,i+k)
2 void z_function(string &s, int z[]) {
3     int n = si(s);
4     forn(i,n) z[i]=0;
5     for (int i = 1, l = 0, r = 0; i < n; ++i) {
6         if (i <= r) z[i] = min (r - i + 1, z[i - l]);
7         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
8         if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
9     }
10 }

```

## 4.11. Palindrome

```

1 bool palindrome(ll x){
2     string s = to_string(x); int n = si(s);
3     forn(i,n/2) if(s[i] != s[n-i-1]) return 0;
4     return 1;
5 }

```

## 5. Geometría

### 5.1. Punto

```

1 struct pto {
2     double x, y;
3     pto(double x=0, double y=0):x(x),y(y){}
4     pto operator+(pto a){return pto(x+a.x, y+a.y);}
5     pto operator-(pto a){return pto(x-a.x, y-a.y);}
6     pto operator+(double a){return pto(x+a, y+a);}
7     pto operator*(double a){return pto(x*a, y*a);}
8     pto operator/(double a){return pto(x/a, y/a);}
9     //dot product, producto interno:
10    double operator*(pto a){return x*a.x+y*a.y;}
11    //module of the cross product or vectorial product:
12    //if a is less than 180 clockwise from b, a^b>0

```

```

13 double operator^(pto a){return x*a.y-y*a.x;}
14 //returns true if this is at the left side of line qr
15 bool left(pto q, pto r){return ((q-*this)^(r-*this))>0;}
16 bool operator<(const pto &a) const{return x<a.x-EPS || (abs(x-a.x)<EPS
    && y<a.y-EPS);}
17 bool operator==(pto a){return abs(x-a.x)<EPS && abs(y-a.y)<EPS;}
18 double norm(){return sqrt(x*x+y*y);}
19 double norm_sq(){return x*x+y*y;}
20 };
21 double dist(pto a, pto b){return (b-a).norm();}
22 double dist_sq(pto a, pto b){return (b-a).norm_sq();}
23 typedef pto vec;
24
25 double angle(pto a, pto o, pto b){
26     pto oa=a-o, ob=b-o;
27     return atan2(oa^ob, oa*ob);
28 }
29
30 //rotate p by theta rads CCW w.r.t. origin (0,0)
31 pto rotate(pto p, double theta){
32     return pto(p.x*cos(theta)-p.y*sin(theta),

```

## 5.2. Orden radial de puntos

```

1 //orden total de puntos alrededor de un punto r
2 // hacer operadores ^ y - constantes
3 struct RadialOrder {
4     pto r;
5     RadialOrder(pto r) : r(r) {}
6     int cuad(const pto &a) const {
7         if(a.x > 0 && a.y >= 0)return 0;
8         if(a.x <= 0 && a.y > 0)return 1;
9         if(a.x < 0 && a.y <= 0)return 2;
10        if(a.x >= 0 && a.y < 0)return 3;
11        return -1;
12    }
13    bool comp(const pto &p1, const pto &p2) const {
14        int c1 = cuad(p1), c2 = cuad(p2);
15        if (c1 == c2) return (p1 ^ p2) > 0;
16        else return c1 < c2;
17    }
18    bool operator()(const pto&p1, const pto&p2) const {
19        return comp(p1 - r, p2 - r);

```

```

20     }
21 };

```

## 5.3. Line

```

1 int sgn(ll x){return x<0? -1 : !!x;}
2 struct line{
3     line() {}
4     double a,b,c;//Ax+By=C
5     //pto MUST store float coordinates!
6     line(double a, double b, double c):a(a),b(b),c(c){}
7     line(pto p, pto q): a(q.y-p.y), b(p.x-q.x), c(a*p.x+b*p.y) {}
8     int side(pto p){return sgn(ll(a) * p.x + ll(b) * p.y - c);}
9 };
10 bool parallels(line l1, line l2){return abs(l1.a*l2.b-l2.a*l1.b)<EPS;}
11 pto inter(line l1, line l2){//intersection
12     double det=l1.a*l2.b-l2.a*l1.b;
13     if(abs(det)<EPS) return pto(INF, INF);//parallels
14     return pto(l2.b*l1.c-l1.b*l2.c, l1.a*l2.c-l2.a*l1.c)/det;
15 }

```

## 5.4. Segment

```

1 struct segm{
2     pto s,f;
3     segm(pto s, pto f) : s(s), f(f) {}
4     pto closest(pto p) { //use for dist to point
5         double l2 = dist_sq(s, f);
6         if(l2==0.) return s;
7         double t=((p-s)*(f-s))/l2;
8         if (t<0.) return s; //dont write if its a line
9         else if(t>1.)return f; //dont write if its a line
10        return s+((f-s)*t);
11    }
12    bool inside(pto p){return abs(dist(s, p)+dist(p, f)-dist(s, f))<EPS;
13        };
14 };
15
16 //NOTA: Si los segmentos son colineales solo devuelve un punto de
17 //interseccion
18 pto inter(segm s1, segm s2){
19     if(s1.inside(s2.s)) return s2.s; //Fix cuando son colineales
20     if(s1.inside(s2.f)) return s2.f; //Fix cuando son colineales
21     pto r=inter(line(s1.s, s1.f), line(s2.s, s2.f));

```

```

20     if(s1.inside(r) && s2.inside(r)) return r;
21     return pto(INF, INF);
22 }

```

## 5.5. Rectangle

```

1 struct rect{
2     //lower-left and upper-right corners
3     pto lw, up;
4 };
5 //returns if there's an intersection and stores it in r
6 bool inter(rect a, rect b, rect &r){
7     r.lw=pto(max(a.lw.x, b.lw.x), max(a.lw.y, b.lw.y));
8     r.up=pto(min(a.up.x, b.up.x), min(a.up.y, b.up.y));
9     //check case when only a edge is common
10    return r.lw.x<r.up.x && r.lw.y<r.up.y;
11 }

```

## 5.6. Polygon Area

```

1 double area(vector<pto> &p){//0(sz(p))
2     double area=0;
3     forn(i, sz(p)) area+=p[i]^p[(i+1)%sz(p)];
4     //if points are in clockwise order then area is negative
5     return abs(area)/2;
6 }
7 //Area ellipse = M_PI*a*b where a and b are the semi axis lengths
8 //Area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s=(a+b+c)/2

```

## 5.7. Circle

```

1 vec perp(vec v){return vec(-v.y, v.x);}
2 line bisector(pto x, pto y){
3     line l=line(x, y); pto m=(x+y)/2;
4     return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
5 }
6 struct Circle{
7     pto o;
8     double r;
9     Circle(pto x, pto y, pto z){
10         o=inter(bisector(x, y), bisector(y, z));
11         r=dist(o, x);
12     }
13     pair<pto, pto> ptosTang(pto p){

```

```

14     pto m=(p+o)/2;
15     tipo d=dist(o, m);
16     tipo a=r*r/(2*d);
17     tipo h=sqrt(r*r-a*a);
18     pto m2=o+(m-o)*a/d;
19     vec per=perp(m-o)/d;
20     return make_pair(m2-per*h, m2+per*h);
21 }
22 };
23 //finds the center of the circle containing p1 and p2 with radius r
24 //as there may be two solutions swap p1, p2 to get the other
25 bool circle2PtsRad(pto p1, pto p2, double r, pto &c){
26     double d2=(p1-p2).norm_sq(), det=r*r/d2-0.25;
27     if(det<0) return false;
28     c=(p1+p2)/2+perp(p2-p1)*sqrt(det);
29     return true;
30 }
31 #define sqr(a) ((a)*(a))
32 #define feq(a,b) (fabs((a)-(b))<EPS)
33 pair<tipo, tipo> ecCuad(tipo a, tipo b, tipo c){//a*x*x+b*x+c=0
34     tipo dx = sqrt(b*b-4.0*a*c);
35     return make_pair((-b + dx)/(2.0*a), (-b - dx)/(2.0*a));
36 }
37 pair<pto, pto> interCL(Circle c, line l){
38     bool sw=false;
39     if((sw=feq(0,l.b))){
40         swap(l.a, l.b);
41         swap(c.o.x, c.o.y);
42     }
43     pair<tipo, tipo> rc = ecCuad(
44         sqr(l.a)+sqr(l.b),
45         2.0*l.a*l.b*c.o.y-2.0*(sqr(l.b)*c.o.x+l.c*l.a),
46         sqr(l.b)*(sqr(c.o.x)+sqr(c.o.y)-sqr(c.r))+sqr(l.c)-2.0*l.c*l.b*c.o.y
47     );
48     pair<pto, pto> p( pto(rc.first, (l.c - l.a * rc.first) / l.b),
49         pto(rc.second, (l.c - l.a * rc.second) / l.b) );
50     if(sw){
51         swap(p.first.x, p.first.y);
52         swap(p.second.x, p.second.y);
53     }
54     return p;
55 }
56 pair<pto, pto> interCC(Circle c1, Circle c2){

```

```

57 line l;
58 l.a = c1.o.x-c2.o.x;
59 l.b = c1.o.y-c2.o.y;
60 l.c = (sqr(c2.r)-sqr(c1.r)+sqr(c1.o.x)-sqr(c2.o.x)+sqr(c1.o.y)
61 -sqr(c2.o.y))/2.0;
62 return interCL(c1, l);
63 }

```

## 5.8. Point in Poly

```

1 //checks if v is inside of P, using ray casting
2 //works with convex and concave.
3 //excludes boundaries, handle it separately using segment.inside()
4 bool inPolygon(pto v, vector<pto>& P) {
5     bool c = 0;
6     forn(i, si(P)){
7         int j = (i+1) % si(P);
8         if((P[j].y > v.y) != (P[i].y > v.y) && (v.x < (P[i].x - P[j].x) * (v
          .y-P[j].y) / (P[i].y - P[j].y) + P[j].x)) c = !c;
9     }
10    return c;
11 }

```

## 5.9. Point in Convex Poly log(n)

```

1 void normalize(vector<pto> &pt){ //delete collinear points first!
2     //this makes it clockwise:
3     if(pt[2].left(pt[0], pt[1])) reverse(pt.begin(), pt.end());
4     int n=si(pt), pi=0;
5     forn(i, n)
6         if(pt[i].x<pt[pi].x || (pt[i].x==pt[pi].x && pt[i].y<pt[pi].y))
7             pi=i;
8     vector<pto> shift(n); //puts pi as first point
9     forn(i, n) shift[i]=pt[(pi+i)%n];
10    pt.swap(shift);
11 }
12 bool inPolygon(pto p, const vector<pto> &pt){
13     //call normalize first!
14     if(p.left(pt[0], pt[1]) || p.left(pt[si(pt)-1], pt[0])) return 0;
15     int a=1, b=si(pt)-1;
16     while(b-a>1){
17         int c=(a+b)/2;
18         if(!p.left(pt[0], pt[c])) a=c;
19         else b=c;

```

```

20     }
21     return !p.left(pt[a], pt[a+1]);
22 }

```

## 5.10. Convex Check CHECK

```

1 bool isConvex(vector<int> &p){//O(N), delete collinear points!
2     int N=sz(p);
3     if(N<3) return false;
4     bool isLeft=p[0].left(p[1], p[2]);
5     forr(i, 1, N)
6         if(p[i].left(p[(i+1)%N], p[(i+2)%N])!=isLeft)
7             return false;
8     return true; }

```

## 5.11. Convex Hull

```

1 //stores convex hull of P in S, CCW order
2 //left must return >=EPS to delete collinear points!
3 void hull(vector<pto>& P, vector<pto> &S){
4     S.clear();
5     sort(P.begin(), P.end()); //first x, then y
6     forn(i, si(P)){//lower hull
7         while(si(S)>= 2 && S[si(S)-1].left(S[si(S)-2], P[i])) S.pop_back();
8         S.pb(P[i]);
9     }
10    S.pop_back();
11    int k=si(S);
12    dforn(i, si(P)){//upper hull
13        while(si(S) >= k+2 && S[si(S)-1].left(S[si(S)-2], P[i])) S.pop_back
          ();
14        S.pb(P[i]);
15    }
16    S.pop_back();
17 }

```

## 5.12. Cut Polygon

```

1 //cuts polygon Q along the line ab
2 //stores the left side (swap a, b for the right one) in P
3 void cutPolygon(pto a, pto b, vector<pto> Q, vector<pto> &P){
4     P.clear();
5     forn(i, sz(Q)){
6         double left1=(b-a)^(Q[i]-a), left2=(b-a)^(Q[(i+1)%sz(Q)]-a);

```

```

7   if(left1>=0) P.pb(Q[i]);
8   if(left1*left2<0)
9       P.pb(inter(line(Q[i], Q[(i+1)%sz(Q)]), line(a, b)));
10  }
11  }

```

### 5.13. Bresenham

```

1  //plot a line approximation in a 2d map
2  void bresenham(pto a, pto b){
3      pto d=b-a; d.x=abs(d.x), d.y=abs(d.y);
4      pto s(a.x<b.x? 1: -1, a.y<b.y? 1: -1);
5      int err=d.x-d.y;
6      while(1){
7          m[a.x][a.y]=1;//plot
8          if(a==b) break;
9          int e2=err;
10         if(e2 >= 0) err-=2*d.y, a.x+=s.x;
11         if(e2 <= 0) err+= 2*d.x, a.y+= s.y;
12     }
13 }

```

### 5.14. Rotate Matrix

```

1  //rotates matrix t 90 degrees clockwise
2  //using auxiliary matrix t2(faster)
3  void rotate(){
4      forn(x, n) forn(y, n)
5          t2[n-y-1][x]=t[x][y];
6      memcpy(t, t2, sizeof(t));
7  }

```

### 5.15. Interseccion de Circulos en $n^3 \log(n)$

```

1  struct event {
2      double x; int t;
3      event(double xx, int tt) : x(xx), t(tt) {}
4      bool operator <(const event &o) const { return x < o.x; }
5  };
6  typedef vector<Circle> VC;
7  typedef vector<event> VE;
8  int n;
9  double cuenta(VE &v, double A, double B) {
10     sort(v.begin(), v.end());

```

```

11     double res = 0.0, lx = ((v.empty())?0.0:v[0].x);
12     int contador = 0;
13     forn(i,sz(v)) {
14         //interseccion de todos (contador == n), union de todos (
15             contador > 0)
16         //conjunto de puntos cubierto por exacta k Circulos (contador ==
17             k)
18         if (contador == n) res += v[i].x - lx;
19         contador += v[i].t, lx = v[i].x;
20     }
21     return res;
22 }
23 // Primitiva de sqrt(r*r - x*x) como funcion double de una variable x.
24 inline double primitiva(double x, double r) {
25     if (x >= r) return r*r*M_PI/4.0;
26     if (x <= -r) return -r*r*M_PI/4.0;
27     double raiz = sqrt(r*r-x*x);
28     return 0.5 * (x * raiz + r*r*atan(x/raiz));
29 }
30 double interCircle(VC &v) {
31     vector<double> p; p.reserve(v.size() * (v.size() + 2));
32     forn(i,sz(v)) p.push_back(v[i].c.x + v[i].r), p.push_back(v[i].c.x
33         - v[i].r);
34     forn(i,sz(v)) forn(j,i) {
35         Circle &a = v[i], b = v[j];
36         double d = (a.c - b.c).norm();
37         if (fabs(a.r - b.r) < d && d < a.r + b.r) {
38             double alfa = acos((sqr(a.r) + sqr(d) - sqr(b.r)) / (2.0 * d
39                 * a.r));
40             pto vec = (b.c - a.c) * (a.r / d);
41             p.pb((a.c + rotate(vec, alfa)).x), p.pb((a.c + rotate(vec, -
42                 alfa)).x);
43         }
44     }
45     sort(p.begin(), p.end());
46     double res = 0.0;
47     forn(i,sz(p)-1) {
48         const double A = p[i], B = p[i+1];
49         VE ve; ve.reserve(2 * v.size());
50         forn(j,sz(v)) {
51             const Circle &c = v[j];
52             double arco = primitiva(B-c.c.x,c.r) - primitiva(A-c.c.x,c.r
53                 );

```



```

48         double base = c.c.y * (B-A);
49         ve.push_back(event(base + arco,-1));
50         ve.push_back(event(base - arco, 1));
51     }
52     res += cuenta(ve,A,B);
53 }
54 return res;
55 }

```

## 5.16. Cayley-Menger

Permite calcular el volumen de un simplex (triángulo en  $k$  dimensiones) mediante el cálculo de una determinante.

```

1 double d[5][5];
2
3 double sqr(double x) { return x*x; }
4
5 double init_cayley_menger() { // en los demas d[i][j] va la longitud del
6     lado del vertice i al vertice j
7     for (int i = 0; i < 4; i++) d[i][4] = d[4][i] = 1;
8 }
9 double cayley_menger(vector<int> idx) { // idx = indices de vertices,
10     por ej {0, 1, 2, 3} en 3d
11     idx.push_back(4);
12     int n = (int) idx.size();
13
14     Mat mat(n, n);
15     forn(i,n) forn(j,n) mat[i][j] = sqr(d[idx[i]][idx[j]]);
16
17     double ans = mat.determinant();
18     forn(i,n-2) ans /= -2*(i+1)*(i+1);
19     return sqrt(-ans);
20 }

```

## 5.17. Heron's formula

It states that the area of a triangle whose sides have lengths  $a$ ,  $b$ , and  $c$  is  $A = \sqrt{s(s-a)(s-b)(s-c)}$ , where  $s$  is the semiperimeter of the triangle; that is,

$$s = \frac{a+b+c}{2}.$$

## 6. DP Opt

Observaciones:

$A[i][j]$  el menor  $k$  que logra la solución óptima. En Knuth y D&C la idea es aprovechar los rangos determinados por este arreglo.

### 6.1. Knuth

**Problema de ejemplo:** dado un palito de longitud  $l$ , con  $n$  puntos en los que se puede cortar, determinar el costo mínimo para partir el palito en  $n + 1$  palitos unitarios (la DP se puede adaptar a  $k$  agregando un parámetro extra), donde hay un costo fijo por partir el rango  $i, j$  que *cumple la condición suficiente*. Una función de costos que cumple es la distancia entre los extremos  $j - i$ . El problema clásico de esta pinta es el del ABB óptimo.

**Recurrencia original:**  $dp[i][j] = \min_{i < k < j} dp[i][k] + dp[k][j] + C[i][j]$

**Condición suficiente:**  $A[i, j - 1] \leq A[i, j] \leq A[i + 1, j]$

Es decir, si saco un elemento a derecha el óptimo se mueve a izquierda o se mantiene, y si saco un elemento a izquierda el óptimo se mueve a derecha o se mantiene.

**Complejidad original:**  $O(n^3)$

**Complejidad optimizada:**  $O(n^2)$

**Solución:** iteramos por el tamaño  $len$  del subarreglo (creciente), y para cada extremo izquierdo  $l$ , determinamos el extremo derecho  $r = l + len$  e iteramos por los  $k$  entre  $A[l][r - 1]$  y  $A[l + 1][r]$ , actualizando la solución del estado actual.

### 6.2. Chull

**Problema de ejemplo:**

**Recurrencia original:**

**Condición suficiente:**

**Complejidad original:**

**Complejidad optimizada:**

**Solución:**

### 6.3. Divide & Conquer

**Problema de ejemplo:** dado un arreglo de  $n$  números con valores  $a_1, a_1, \dots, a_n$ , dividirlo en  $k$  subarreglos, tal que la suma de los cuadrados del peso total de cada subarreglo es mínimo.

**Recurrencia original:**  $dp[i][j] = \min_{k < j} dp[i - 1][k] + C[k][j]$

**Condición suficiente:**  $A[i][j] \leq A[i][j + 1]$  o (normalmente más fácil de probar)  $C[a][d] + C[b][c] \geq C[a][c] + C[b][d]$ , con  $a < b < c < d$ .

La segunda condición suficiente es la intuición de que no conviene que los intervalos se contengan.

**Complejidad original:**  $O(kn^2)$

**Complejidad optimizada:**  $O(kn \log(n))$

**Solución:** la idea es, para un  $i$  determinado, partir el rango  $[j_{left}, j_{right})$  al que pertenecen los  $j$  que queremos calcular a la mitad, determinar el óptimo y utilizarlo como límite para calcular los demás. Para implementar esto de forma sencilla, se suele utilizar la función recursiva  $dp(i, j_{left}, j_{right}, opt_{left}, opt_{right})$  que se encarga de, una vez fijado el punto medio  $m$  del rango  $[j_{left}, j_{right})$  iterar por los  $k$  en  $[j_{left}, j_{right})$  para determinar el óptimo  $opt$  para  $m$ , y continuar calculando  $dp(i, j_{left}, m, opt_{left}, opt)$  y  $dp(i, m, j_{right}, opt, opt_{right})$ .

## 7. Matemática

### 7.1. Teoría de números

#### 7.1.1. Funciones multiplicativas, función de Möbius

Una función  $f(n)$  es **multiplicativa** si para cada par de enteros coprimos  $p$  y  $q$  se cumple que  $f(pq) = f(p)f(q)$ .

Si la función  $f(n)$  es multiplicativa, puede evaluarse en un valor arbitrario conociendo los valores de la función en sus factores primos:  $f(n) = f(p_1^{r_1})f(p_2^{r_2}) \dots f(p_k^{r_k})$ .

La **función de Möbius** se define como:

$$\mu(n) = \begin{cases} 0 & d^2 \mid n, \\ 1 & n = 1, \\ (-1)^k & n = p_1 p_2 \dots p_k. \end{cases}$$

#### 7.1.2. Teorema de Wilson

$(p-1)! \equiv -1 \pmod{p}$  Siendo  $p$  primo.

#### 7.1.3. Pequeño teorema de Fermat

$a^p \equiv a \pmod{p}$  Siendo  $p$  primo.

#### 7.1.4. Teorema de Euler

$a^{\varphi(n)} \equiv 1 \pmod{n}$

## 7.2. Combinatoria

### 7.2.1. Burnside's lemma

Sea  $G$  un grupo que actúa en un conjunto  $X$ . Para cada  $g$  en  $G$ , sea  $X^g$  el conjunto de elementos en  $X$  que son invariantes respecto a  $g$ , entonces el número de órbitas  $|X/G|$

es:

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

Por ejemplo, si el grupo  $G$  consiste de las operaciones de rotación, el conjunto  $X$  son los posibles coloreos de un tablero, entonces el número de órbitas  $|X/G|$  es el número de posibles coloreos de un tablero salvo rotaciones.

### 7.2.2. Combinatorios

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```

1 | int MAXC = 1e3+1, C[MAXC][MAXC];
2 | forn(i,MAXC){ // C[n][k] = C(n, k)
3 |     C[i][0] = C[i][i] = 1;
4 |     forsn(k,1,i) C[i][k] = add(C[i-1][k],C[i-1][k-1]);
5 | }
```

### 7.2.3. Lucas Theorem

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p}$$

where  $m = m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0$ ,

and  $n = n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0$

$\binom{m}{n} = 0$  if  $m < n$ .

```

1 | ll lucas(ll n, ll k, int p){ // Calcula C(n,k) %p teniendo C[p][p]
   |     precalculado, p primo
2 |     ll ans = 1;
3 |     while(n + k){ ans = (ans * C[n%p][k%p]) %p; n/=p; k/=p; }
4 |     return ans;
5 | }
```

### 7.2.4. Stirling

$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$  = cantidad de formas de particionar un conjunto de  $n$  elementos en  $m$  subconjuntos no vacíos.

$$\left\{ \begin{smallmatrix} n+1 \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n \\ k-1 \end{smallmatrix} \right\}$$

for  $k > 0$  with initial conditions

$$\left\{ \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \right\} = 1 \quad \text{and} \quad \left\{ \begin{smallmatrix} n \\ 0 \end{smallmatrix} \right\} = \left\{ \begin{smallmatrix} 0 \\ n \end{smallmatrix} \right\} = 0 \text{ for } n > 0.$$



```

1 int MAXS = 1e3+1, S[MAXS][MAXS];
2 void stirling(){
3     S[0][0] = 1;
4     forsn(i,1,N) S[i][0] = S[0][i] = 0;
5     forsn(i,1,N) forsn(j,1,N)
6         S[i][j] = add(mul(S[i-1][j],j),S[i-1][j-1]);
7 }

```

### 7.2.5. Bell

$B_n$  = cantidad de formas de particionar un conjunto de  $n$  elementos en subconjuntos no vacíos.

$$B_0 = B_1 = 1$$

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k.$$

$$B_n = \sum_{k=0}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\}.$$

```

1 int bell[MAXC+1][MAXC+1];
2 ll bell(int n){
3     bell[0] = 1;
4     forsn(i,1,n+1) forsn(k,0,i)
5         bell[i] = add(bell[i],mul(C[i-1][k],bell[k]));
6 }

```

### 7.2.6. Eulerian

$A_{n,m}$  = cantidad de permutaciones de 1 a  $n$  con  $m$  ascensos ( $m$  elementos mayores que el anterior).

$$A(n, m) = (n - m)A(n - 1, m - 1) + (m + 1)A(n - 1, m).$$

### 7.2.7. Catalan

$C_n$  = cantidad de árboles binarios de  $n+1$  hojas, en los que cada nodo tiene cero o dos hijos.

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} \quad \text{con } n \geq 1.$$

$$C_0 = 1 \quad \text{y} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{con } n \geq 0.$$

## 7.3. Sumatorias conocidas

$$\sum_{i=0}^n \binom{n}{i} = 2^n$$

$$\begin{aligned} \sum_{i=0}^n i \binom{n}{i} &= n * 2^{n-1} \\ \sum_{i=m}^n i &= \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2} \\ \sum_{i=0}^n i &= \sum_{i=1}^n i = \frac{n(n+1)}{2} \\ \sum_{i=0}^n i^2 &= \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \\ \sum_{i=0}^n i(i-1) &= \frac{8}{6} \left(\frac{n}{2}\right) \left(\frac{n}{2} + 1\right) (n+1) \text{ (doubles)} \rightarrow \text{Sino ver caso impar y par} \\ \sum_{i=0}^n i^3 &= \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^n i\right]^2 \\ \sum_{i=0}^n i^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30} \\ \sum_{i=0}^n i^p &= \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^p \frac{B_k}{p-k+1} \binom{p}{k} (n+1)^{p-k+1} \end{aligned}$$

## 7.4. Ec. Característica

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = 0$$

$$p(x) = a_0 x^k + a_1 x^{k-1} + \dots + a_k$$

Sean  $r_1, r_2, \dots, r_q$  las raíces distintas, de mult.  $m_1, m_2, \dots, m_q$

$$T(n) = \sum_{i=1}^q \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

Las constantes  $c_{ij}$  se determinan por los casos base.

## 7.5. Aritmetica Modular

```

1 int normal(int a){ return ((a % M) + M) % M; } // Use for negative
   numbers (the 2nd modulo is avoidable)
2 int add(int a, int b){ return a+b < M ? a+b : a+b-M; }
3 int sub(int a, int b){ return a-b >= 0 ? a-b : a-b+M; }
4 int mul(int a, int b){ return int(ll(a)*b % M); }
5 int div(int a, int b) { return mul(a, inv(b)); }
6 int neg(int a){ return add(-a, M); }

```

## 7.6. Exp. de Numeros Mod.

```

1 ll pot(ll b, ll e){ // O(log e)
2     if(!e) return 1;
3     ll q = pot(b, e/2); q = mul(q, q);
4     return (e & 1 ? mul(b, q) : q);
5 }

```

## 7.7. Exp. de Matrices y Fibonacci en log(n)

```

1 const int S = 2;
2 int temp[S][S];
3 void mul(int a[S][S], int b[S][S]){
4     forn(i, S) forn(j, S) temp[i][j] = 0;

```

```

5   forn(i, S) forn(j, S) forn(k, S) temp[i][j] += a[i][k] * b[k][j];
6   forn(i, S) forn(j, S) a[i][j] = temp[i][j];
7 }
8 void powmat(int a[S][S], ll n, int res[S][S]){
9     forn(i, S) forn(j, S) res[i][j] = (i==j);
10    while(n){
11        if(n&1) mul(res, a), n--;
12        else mul(a, a), n/=2;
13    }
14 }

```

## 7.8. Matrices y determinante $O(n^3)$

```

1 struct Mat {
2     vector<vector<double>> > vec;
3     Mat(int n): vec(n, vector<double>(n)) {}
4     Mat(int n, int m): vec(n, vector<double>(m)) {}
5     vector<double> &operator[](int f){return vec[f];}
6     const vector<double> &operator[](int f) const {return vec[f];}
7     int size() const {return si(vec);}
8     Mat operator+(Mat &b) { ///this de n x m entonces b de n x m
9         Mat m(si(b), si(b[0]));
10        forn(i, si(vec)) forn(j, si(vec[0])) m[i][j] = vec[i][j] + b[i][j];
11        return m;    }
12    Mat operator*(const Mat &b) { ///this de n x m entonces b de m x t
13        int n = si(vec), m = si(vec[0]), t = si(b[0]);
14        Mat mat(n, t);
15        forn(i, n) forn(j, t) forn(k, m) mat[i][j] += vec[i][k] * b[k][j];
16        return mat;    }
17    double determinant(){///sacado de e maxx ru
18        double det = 1;
19        int n = si(vec);
20        Mat m(*this);
21        forn(i, n){///para cada columna
22            int k = i;
23            forsn(j, i+1, n)///busco la fila con mayor val abs
24                if(abs(m[j][i]) > abs(m[k][i])) k = j;
25            if(abs(m[k][i]) < EPS) return 0;
26            m[i].swap(m[k]);///la swapeo
27            if(i!=k) det = -det;
28            det *= m[i][i];
29            forsn(j, i+1, n) m[i][j] /= m[i][i];

```

```

30        ///hago 0 todas las otras filas
31        forn(j, n) if (j!= i && abs(m[j][i]) > EPS)
32            forsn(k, i+1, n) m[j][k] -= m[i][k] * m[j][i];
33        }
34        return det;
35    }
36 };

```

## 7.9. Primes and factorization

```

1 map<ll, int> F;
2 const int N = 1e7;
3 int lp[N+1], P[N+1], sp=0; // prime_density(n) ~= n/ln(n)
4
5 void sieve(){ // O(N)
6     forsn(i, 2, N+1){
7         if(lp[i] == 0) lp[i] = i, P[sp++] = i;
8         for(int j=0; j < sp && P[j] <= lp[i] && i*P[j] <= N; j++) lp[i*P[j]]
9             = P[j];
10    }
11
12    void factorize(int x){ // O(log(x)), x <= N, sieve needed
13        while(x != 1) F[lp[x]]++, x /= lp[x];
14    }
15
16    void factorize(ll x) { // O(sqrt(x)), no sieve needed
17        for(int i = 2; i*i <= x; i++)
18            while(x % i == 0) F[i]++, x /= i;
19        if(x != 1) F[x]++;
20    }

```

## 7.10. Euler's Phi

```

1 const int N = 1e6;
2 int lp[N+1], P[N/5], phi[N+1], sp=0; // prime_density(n) ~= n/ln(n)
3 // lp (least prime) allows fast factorization of numbers <= N
4
5 // Euler's totient function (phi) counts the positive integers up to a
6 // given integer n that are relatively prime to n
7 void init_phi(){ // Primes and Phi <= N in O(N)
8     phi[1] = 1;
9     forsn(i, 2, N+1){
10        if(lp[i] == 0) lp[i] = i, P[sp++] = i, phi[i] = i-1;

```

```

10     else phi[i] = lp[i] == lp[i/lp[i]] ? phi[i/lp[i]]*lp[i] : phi[i/lp[i]]*(lp[i]-1);
11     for(int j = 0; j < sp && P[j] <= lp[i] && i*P[j] <= N; j++) lp[i*P[j]] = P[j];
12 }
13 }
14
15 int eulerPhi(int n){ // 0(sqrt(n)) (single number)
16     int r = n;
17     for(int i = 2; i*i <= n; i++) if(n % i == 0){
18         r -= r/i;
19         while(n % i == 0) n /= i;
20     }
21     if(n > 1) r -= r/n;
22     return r;
23 }

```

## 7.11. Criba

```

1 const int MAXP = 100100; // no inclusive
2 int criba[MAXP];
3 void crearcriba(){
4     int w[] = {4, 2, 4, 2, 4, 6, 2, 6};
5     for(int p = 25; p < MAXP; p += 10) criba[p] = 5;
6     for(int p = 9; p < MAXP; p += 6) criba[p] = 3;
7     for(int p = 4; p < MAXP; p += 2) criba[p] = 2;
8     for(int p = 7, cur = 0; p*p < MAXP; p += w[cur++&7]) if(!criba[p]){
9         for(int j = p*p; j < MAXP; j += (p << 1))
10             if(!criba[j]) criba[j] = p;
11     }
12 }
13 vector<int> primos;
14 void buscarprimos(){
15     crearcriba();
16     forsn(i, 2, MAXP) if(!criba[i]) primos.push_back(i);
17 }

```

## 7.12. Funciones de primos

Sea  $n = \prod p_i^{k_i}$ , fact(n) genera un map donde a cada  $p_i$  le asocia su  $k_i$

```

1 // TODO: actualizar macros. Ver que sean compatibles con criba
2 // INCLUIR CRIBA
3
4 //factoriza bien numeros hasta MAXP^2

```

```

5 map<ll,ll> fact(ll n){ //0 (cant primos)
6     map<ll,ll> ret;
7     for (ll p : primos){
8         while(!(n%p)){
9             ret[p]++; //divisor found
10            n/=p;
11        }
12    }
13    if(n>1) ret[n]++;
14    return ret;
15 }
16 //factoriza bien numeros hasta MAXP
17 map<ll,ll> fact2(ll n){ //0 (lg n)
18     map<ll,ll> ret;
19     while (criba[n]){
20         ret[criba[n]]++;
21         n/=criba[n];
22     }
23     if(n>1) ret[n]++;
24     return ret;
25 }
26 //Usar asi: divisores(fac, divs, fac.begin()); NO ESTA ORDENADO
27 void divisores(const map<ll,ll> &f, vector<ll> &divs, map<ll,ll>::
28     iterator it, ll n=1){
29     if(it==f.begin()) divs.clear();
30     if(it==f.end()) { divs.pb(n); return; }
31     ll p=it->fst, k=it->snd; ++it;
32     forn(_, k+1) divisores(f, divs, it, n), n*=p;
33 }
34 ll sumDiv (ll n){
35     ll rta = 1;
36     map<ll,ll> f=fact(n);
37     forall(it, f) {
38         ll pot = 1, aux = 0;
39         forn(i, it->snd+1) aux += pot, pot *= it->fst;
40         rta*=aux;
41     }
42     return rta;
43 }
44 ll eulerPhi (ll n){ // con criba: 0(lg n)
45     ll rta = n;
46     map<ll,ll> f=fact(n);
47     forall(it, f) rta -= rta / it->first;

```

```

47     return rta;
48 }
49 ll eulerPhi2 (ll n){ // 0 (sqrt n)
50     ll r = n;
51     forr (i,2,n+1){
52         if ((ll)i*i > n) break;
53         if (n % i == 0){
54             while (n%i == 0) n/=i;
55             r -= r/i; }
56     }
57     if (n != 1) r-= r/n;
58     return r;
59 }

```

### 7.13. Phollard's Rho - Miller-Rabin

```

1  ll gcd(ll a, ll b){return b?__gcd(a,b):a;}
2
3  typedef unsigned long long ull;
4  ull mulmod(ull a, ull b, ull m){ // 0 <= a, b < m
5      long double x; ull c; ll r;
6      x = a; c = x * b / m;
7      r = (ll)(a * b - c * m) % (ll)m;
8      return r < 0 ? r + m : r;
9  }
10
11 ll expmod(ll b, ll e, ll m){ // 0(log(b))
12     ll ans = 1;
13     while(e){
14         if(e&1)ans = mulmod(ans, b, m);
15         b = mulmod(b, b, m); e >>= 1;
16     }
17     return ans;
18 }
19
20 bool es_primo_prob (ll n, int a)
21 {
22     if (n == a) return true;
23     ll s = 0, d = n-1;
24     while (d % 2 == 0) s++, d/=2;
25
26     ll x = expmod(a,d,n);

```

```

27     if ((x == 1) || (x+1 == n)) return true;
28
29     forn (i, s-1){
30         x = mulmod(x, x, n);
31         if (x == 1) return false;
32         if (x+1 == n) return true;
33     }
34     return false;
35 }
36
37 bool rabin (ll n){ //devuelve true si n es primo 0(n^0.25)
38     if (n == 1) return false;
39     const int ar[] = {2,3,5,7,11,13,17,19,23};
40     forn (j,9)
41         if (!es_primo_prob(n,ar[j]))
42             return false;
43     return true;
44 }
45
46 ll rho(ll n){
47     if(!(n&1))return 2;
48     ll x = 2, y = 2, d = 1;
49     ll c = rand()%n + 1;
50     while(d == 1){
51         x = (mulmod(x,x, n)+c)%n;
52         y = (mulmod(y,y, n)+c)%n;
53         y = (mulmod(y,y, n)+c)%n;
54         if(x >= y)d = gcd(x-y, n);
55         else d = gcd(y-x, n);
56     }
57     return d == n ? rho(n) : d;
58 }
59 void fact(ll n, map<ll,int>& f){ //0 (lg n)^3
60     if(n == 1)return;
61     if(rabin(n)){ f[n]++; return; }
62     ll q = rho(n); fact(q, f); fact(n/q, f);
63 }

```

### 7.14. GCD

```

1  template<class T> T gcd(T a,T b){return b?__gcd(a,b):a;}
2  //en C++17 gcd(a,b) predefinido

```

## 7.15. LCM

```
1 | template<class T> T lcm(T a,T b){return a*(b/gcd(a,b));}
2 | //en C++17 lcm(a,b) predefinido
```

## 7.16. Euclides extendido

Dados  $a$  y  $b$ , encuentra  $x$  e  $y$  tales que  $a * x + b * y = gcd(a,b)$ .

```
1 | pair<ll,ll> extendedEuclid (ll a, ll b){ //a * x + b * y = gcd(a,b)
2 |     ll x,y;
3 |     if (b==0) return mp(1,0);
4 |     auto p=extendedEuclid(b,a%b);
5 |     x=p.snd;
6 |     y=p.fst-(a/b)*x;
7 |     if (a*x + b*y == -gcd(a,b)) a = -a, b = -b;
8 |     return mp(x,y);
9 | }
```

## 7.17. Inversos

```
1 | const int MAXM = 15485867; // Tiene que ser primo
2 | ll inv[MAXM]; //inv[i]*i=1 M M
3 | void calc(int p){//0(p)
4 |     inv[1]=1;
5 |     forsn(i, 2, p) inv[i]= p-((p/i)*inv[p%i])%p;
6 | }
7 | // Llamar calc(MAXM);
8 |
9 | int inv(int x){//0(log x)
10 |     return pot(x, eulerphi(M)-1);//si M no es primo(sacar a mano)
11 |     return pot(x, M-2);//si M es primo
12 | }
13 |
14 | // Inversos con euclides en O(log(x)) sin precomputo:
15 | // extendedEuclid(a, -m).fst (si coprimos a y m)
```

## 7.18. Ecuaciones diofánticas

Basado en Euclides extendido. Dados  $a$ ,  $b$ , y  $r$  obtiene  $x$  e  $y$  tales que  $a * x + b * y = r$ , suponiendo que  $gcd(a,b)|r$ . Las soluciones son de la forma  $(x,y) = (x_1 - b/gcd(a,b) * k_1, x_2 + a/gcd(a,b) * k_2)$  donde  $x_1$  y  $x_2$  son las soluciones particulares que obtuvo Euclides.

```
1 | pair<pair<ll,ll>,pair<ll,ll> > diophantine(ll a,ll b, ll r) {
```

```
2 |     //a*x+b*y=r where r is multiple of gcd(a,b);
3 |     ll d=gcd(a,b);
4 |     a/=d; b/=d; r/=d;
5 |     auto p = extendedEuclid(a,b);
6 |     p.fst*=r; p.snd*=r;
7 |     assert(a*p.fst+b*p.snd==r);
8 |     return mp(p,mp(-b,a)); // solutions: (p.fst - b*k1, p.snd + a*k2)
9 | }
```

## 7.19. Teorema Chino del Resto

Dadas  $k$  ecuaciones de la forma  $a_i * x \equiv a_i \pmod{n_i}$ , encuentra  $x$  tal que es solución. Existe una única solución módulo  $lcm(n_i)$ .

```
1 | #define mod(a,m) ((a)%(m) < 0 ? (a)%(m)+(m) : (a)%(m)) // evita overflow
2 |     al no sumar si >= 0
3 | typedef tuple<ll,ll,ll> ec;
4 | pair<ll,ll> sol(ec c){ //requires inv, diophantine
5 |     ll a=get<0>(c), x1=get<1>(c), m=get<2>(c), d=gcd(a,m);
6 |     if (d==1) return mp(mod(x1*inv(a,m),m), m);
7 |     else return x1%d ? mp(-1LL,-1LL) : sol({a/d,x1/d,m/d});
8 | }
9 | pair<ll,ll> crt(vector< ec > cond) { // returns: (sol, lcm)
10 |     ll x1=0,m1=1,x2,m2;
11 |     for(auto t:cond){
12 |         tie(x2,m2)=sol(t);
13 |         if((x1-x2)%gcd(m1,m2))return mp(-1,-1);
14 |         if(m1==m2)continue;
15 |         ll k=diophantine(m2,-m1,x1-x2).fst.snd,l=m1*(m2/gcd(m1,m2));
16 |         x1=mod(m1*mod(k, l/m1)+x1,l);m1=l; // evita overflow con prop modulo
17 |     }
18 |     return sol(make_tuple(1,x1,m1));
19 | } //cond[i]={ai,bi,mi} ai*xi=bi (mi); assumes lcm fits in ll
```

## 7.20. Simpson

```
1 | double integral(double a, double b, int n=10000) { //0(n), n=cantdiv
2 |     double area=0, h=(b-a)/n, fa=f(a), fb;
3 |     forn(i, n){
4 |         fb=f(a+h*(i+1));
5 |         area+=fa+ 4*f(a+h*(i+0.5)) +fb, fa=fb;
6 |     }
7 |     return area*h/6.;}
```

## 7.21. Fraction

```

1 template<class T> T gcd(T a,T b){return b==0?a:gcd(b,a%b);}
2
3 struct frac{
4     int p,q;
5     frac(int p=0, int q=1):p(p),q(q) {norm();}
6     void norm(){
7         int a = gcd(p,q);
8         p/=a, q/=a;
9         if(q < 0) q=-q, p=-p;}
10    frac operator+(const frac& o){
11        int a = gcd(q,o.q);
12        return frac(add(mul(p,o.q/a), mul(o.p,q/a)), mul(q,o.q/a));}
13    frac operator-(const frac& o){
14        int a = gcd(q,o.q);
15        return frac(sub(mul(p,o.q/a), mul(o.p,q/a)), mul(q,o.q/a));}
16    frac operator*(frac o){
17        int a = gcd(q,o.p), b = gcd(o.q,p);
18        return frac(mul(p/b,o.p/a), mul(q/a,o.q/b));}
19    frac operator/(frac o){
20        int a = gcd(q,o.q), b = gcd(o.p,p);
21        return frac(mul(p/b,o.q/a), mul(q/a,o.p/b));}
22    bool operator<(const frac &o) const{return ll(p)*o.q < ll(o.p)*q;}
23    bool operator==(frac o){return p==o.p && q==o.q;}
24    bool operator!=(frac o){return p!=o.p || q!=o.q;}
25 };

```

## 7.22. Polinomio, Ruffini e interpolación de Lagrange

**Interpolación de Lagrange:** dados  $n+1$  pares  $(x_i, y_i)$  permite encontrar el polinomio de grado  $n$  tal que  $f(x_i) = y_i$ .

**Explicación:** computa  $P(x) = y_1 * f_1(x) + y_2 * f_2(x) + \dots + y_{n+1} * f_{n+1}(x)$  donde  $f_i(x) = \frac{g_i(x)}{g_i(x_i)}$ ,  $g_i(x) = \frac{h(x)}{x-x_i}$  y  $h(x) = (x-x_1) * (x-x_2) * \dots * (x-x_{n+1})$ . Usa Ruffini para la división de polinomios.

Trucazo para computar en  $O(n)$ :  $x_{i+1} - x_i = x_{j+1} - x_j$  para todo  $i, j < n$ .

**Ejemplo de problema:** tenés que calcular una respuesta que depende de un  $n$  y parece ser polinomial, conseguís un par de puntos e intentás armar el polinomio (usando el algoritmo online u offline).

```

1 using tp = int; // type of polynomial
2 template<class T=tp>
3 struct poly { // poly<> : 1 variable, poly<poly<>>: 2 variables, etc.
4     vector<T> c;

```

```

5     T& operator[](int k){return c[k];}
6     poly(vector<T>& c):c(c){}
7     poly(initializer_list<T> c):c(c){}
8     poly(int k):c(k){}
9     poly(){}
10    poly operator+(poly<T> o){
11        int m=si(c),n=si(o.c);
12        poly res(max(m,n));
13        forn(i,m)res[i]=res[i]+c[i];
14        forn(i,n)res[i]=res[i]+o.c[i];
15        return res;
16    }
17    poly operator*(tp k){
18        poly res(si(c));
19        forn(i,si(c))res[i]=c[i]*k;
20        return res;
21    }
22    poly operator*(poly o){
23        int m=si(c),n=si(o.c);
24        poly res(m+n-1);
25        forn(i,m)forn(j,n)res[i+j]=res[i+j]+c[i]*o.c[j];
26        return res;
27    }
28    poly operator-(poly<T> o){return *this+(o*-1);}
29    T operator()(tp v){
30        T sum(0);
31        dforn(i, si(c)) sum=sum*v+c[i];
32        return sum;
33    }
34 };
35 // example: p(x,y)=2*x^2+3*x*y-y+4
36 // poly<poly<>> p={{4,-1},{0,3},{2}}
37 // printf("%d\n",p(2)(3)) // 27 (p(2,3))
38 set<tp> roots(poly<> p){ // only for integer polynomials
39     set<tp> r;
40     while(!p.c.empty()&&!p.c.back())p.c.pop_back();
41     if(!p(0))r.insert(0);
42     if(p.c.empty())return r;
43     tp a0=0,an=abs(p[si(p.c)-1]);
44     for(int k=0;!a0;a0=abs(p[k++]));
45     vector<tp> ps,qs;
46     forsn(i,1,sqrt(a0)+1)if(a0%i==0)ps.pb(i),ps.pb(a0/i);
47     forsn(i,1,sqrt(an)+1)if(an%i==0)qs.pb(i),qs.pb(an/i);

```

```

48   for(auto pt:ps)for(auto qt:qs)if(pt%qt==0){
49       tp x=pt/qt;
50       if(!p(x).r.insert(x);
51       if(!p(-x).r.insert(-x);
52   }
53   return r;
54 }
55 pair<poly<>,tp> ruffini(poly<> p, tp r){ // returns pair (result,rem)
56     int n=si(p.c)-1;
57     vector<tp> b(n);
58     b[n-1]=p[n];
59     dform(k, n-1) b[k]=p[k+1]+r*b[k+1];
60     return mp(poly<>(b),p[0]+r*b[0]);
61 }
62 // only for double polynomials
63 pair<poly<>,poly<> > polydiv(poly<> p, poly<> q){ // returns pair (
64     result,rem)
65     int n=si(p.c)-si(q.c)+1;
66     vector<tp> b(n);
67     dform(k, n) {
68         b[k]=p.c.back()/q.c.back();
69         forn(i,si(q.c))p[i+k]-=b[k]*q[i];
70         p.c.pop_back();
71     }
72     while(!p.c.empty()&&abs(p.c.back())<EPS)p.c.pop_back();
73     return mp(poly<>(b),p);
74 }
75 // for double polynomials
76 // O(n^2), constante aaaalta
77 poly<> interpolate(vector<tp> x, vector<tp> y){
78     poly<> q={1},S={0};
79     for(tp a:x)q=poly<>({-a,1})*q;
80     forn(i,si(x)){
81         poly<> Li=ruffini(q,x[i]).fst;
82         Li=Li*(1.0/Li(x[i])); // change for int polynomials
83         S=S+Li*y[i];
84     }
85     return S;
86 }
87 // for int polynomials
88 // O(n), rapido, la posta
89 int evalInterpolation(const vector<int> &y, int x) { // {0, y[0]}, ...
    int ans = 0;

```

```

90   int k = 1;
91   forsn(j, 1, si(y)) {
92       if (x == j) return y[j];
93       k = mul(k, normal(x - j));
94       k = div(k, normal(0 - j));
95   }
96   forn(i, si(y)) {
97       ans = add(ans, mul(y[i], k));
98       if (i + 1 >= si(y)) break;
99       k = mul(k, div(normal(x - i), normal(x - (i + 1))));
100      k = mul(k, div(normal(i - (si(y) - 1)), normal(i + 1))); // TODO
101      : terminar de explicar esta linea
102   }
103   return ans;
104 }

```

## 7.23. Ec. Lineales

```

1  bool resolver_ev(Mat a, Vec y, Vec &x, Mat &ev){
2      int n = a.size(), m = n?a[0].size():0, rw = min(n, m);
3      vector<int> p; forn(i,m) p.push_back(i);
4      forn(i, rw) {
5          int uc=i, uf=i;
6          forr(f, i, n) forr(c, i, m) if(fabs(a[f][c])>fabs(a[uf][uc])) {uf=f;
7              uc=c;}
8          if (freq(a[uf][uc], 0)) { rw = i; break; }
9          forn(j, n) swap(a[j][i], a[j][uc]);
10         swap(a[i], a[uf]); swap(y[i], y[uf]); swap(p[i], p[uc]);
11         tipo inv = 1 / a[i][i]; //aca divide
12         forr(j, i+1, n) {
13             tipo v = a[j][i] * inv;
14             forr(k, i, m) a[j][k]-=v * a[i][k];
15             y[j] -= v*y[i];
16         }
17     } // rw = rango(a), aca la matriz esta triangulada
18     forr(i, rw, n) if (!freq(y[i],0)) return false; // chequeo de
19     compatibilidad
20     x = vector<tipo>(m, 0);
21     dform(i, rw){
22         tipo s = y[i];
23         forr(j, i+1, rw) s -= a[i][j]*x[p[j]];
24         x[p[i]] = s / a[i][i]; //aca divide

```



```

23 }
24 ev = Mat(m-rw, Vec(m, 0)); // Esta parte va SOLO si se necesita el ev
25 forn(k, m-rw) {
26     ev[k][p[k+rw]] = 1;
27     dforn(i, rw){
28         tipo s = -a[i][k+rw];
29         forr(j, i+1, rw) s -= a[i][j]*ev[k][p[j]];
30         ev[k][p[i]] = s / a[i][i]; //aca divide
31     }
32 }
33 return true;
34 }

```

## 7.24. FFT y NTT

### Base teórica

Dado el espacio lineal con producto interno (definido como una integral loca)  $E$ , de funciones continuas definidas por partes  $f: [-\pi, \pi] \rightarrow \mathbb{C}$ , un **sistema ortonormal cerrado infinito** es  $\{1/\sqrt{2}, \sin(x), \cos(x), \sin(2x), \cos(2x), \dots\}$ . Por lo tanto, cualquier función  $f \in E$  puede ser representada por  $\sum_{n=1}^{\infty} \langle f, e_n \rangle e_n$ . Esta combinación lineal (utilizando la sumatoria y el sistema ya definidos), es la **serie de Fourier**.

También se puede definir la **serie compleja de Fourier** mediante el sistema  $\{1, e^{ix}, e^{-ix}, e^{i2x}, e^{-i2x}, \dots\}$ .

Una **transformada de Fourier** permite trabajar con funciones que no están restringidas al intervalo  $[-\pi, \pi]$ . La principal diferencia es que el sistema ortonormal pasa de ser discreto a continuo.

Sin embargo, existe una versión discreta de la transformada, la **transformada discreta de Fourier** (DFT).

Una de las propiedades importantes de la transformada es que la **convolución** de funciones sin transformar se traduce en multiplicar las transformadas.

**FFT**, el algoritmo para calcular rápidamente la DFT, se basa en que dado un polinomio  $A(x)$ ,  $A(x) = A_0(x^2) + x * A_1(x^2)$ , donde  $A_0(x)$  y  $A_1(x)$  son los polinomios que se forman al tomar los términos pares e impares respectivamente.

**NTT** es un algoritmo más lento pero más preciso para calcular la DFT, ya que trabaja con enteros módulo un primo  $p$ .

```

1 // MODNTT-1 needs to be a multiple of MAXN !!
2 // big mod and primitive root for NTT:
3 // const ll MODNTT = 2305843009255636993;
4 // const int RT = 5;
5 // struct for FFT, for NTT is simple (ll with mod operations)
6 struct CD { // or typedef complex<double> CD; (but 4x slower)
7     double r,i;
8     CD(double r=0, double i=0):r(r),i(i){}

```

```

9     double real()const{return r;}
10    void operator/=(const int c){r/=c, i/=c;}
11 };
12 CD operator*(const CD& a, const CD& b){
13     return CD(a.r*b.r-a.i*b.i,a.r*b.i+a.i*b.r);}
14 CD operator+(const CD& a, const CD& b){return CD(a.r+b.r,a.i+b.i);}
15 CD operator-(const CD& a, const CD& b){return CD(a.r-b.r,a.i-b.i);}
16
17 const double pi = acos(-1.0); // FFT
18 CD cp1[MAXN+9],cp2[MAXN+9]; // MAXN must be power of 2 !!
19 int R[MAXN+9];
20 //CD root(int n, bool inv){ // NTT
21 // ll r=pot(RT,(MODNTT-1)/n); // pot: modular exponentiation
22 // return CD(inv?pot(r,MODNTT-2):r);
23 //}
24 void dft(CD* a, int n, bool inv){
25     forn(i,n)if(R[i]<i)swap(a[R[i]],a[i]);
26     for (int m=2;m<=n;m*=2){
27         double z = 2*pi/m*(inv?-1:1); // FFT
28         CD wi = CD(cos(z),sin(z)); // FFT
29         // CD wi=root(m,inv); // NTT
30         for (int j=0;j<n;j+=m){
31             CD w(1);
32             for(int k=j,k2=j+m/2;k2<j+m;k++,k2++){
33                 CD u=a[k]; CD v=a[k2]*w; a[k]=u+v; a[k2]=u-v; w=w*wi;
34             }
35         }
36     }
37     if(inv) forn(i,n)a[i]/=n; // FFT
38     //if(inv){ // NTT
39     // CD z(pot(n,MODNTT-2)); // pot: modular exponentiation
40     // forn(i,n)a[i]=a[i]*z;
41     //}
42 }
43 vi multiply(vi& p1, vi& p2){
44     int n=si(p1)+si(p2)+1;
45     int m=1,cnt=0;
46     while(m<=n)m+=m,cnt++;
47     forn(i,m){R[i]=0;forn(j,cnt)R[i]=(R[i]<<1)|((i>>j)&1);}
48     forn(i,m)cp1[i]=0,cp2[i]=0;
49     forn(i,si(p1))cp1[i]=p1[i];
50     forn(i,si(p2))cp2[i]=p2[i];
51     dft(cp1,m,false);dft(cp2,m,false);

```



```

52  forn(i,m)cp1[i]=cp1[i]*cp2[i];
53  dft(cp1,m,true);
54  vi res;
55  n-=2;
56  forn(i,n)res.pb((ll)floor(cp1[i].real()+0.5)); // change for NTT
57  return res;
58 }

```

## 7.25. Programación lineal: Simplex

### Introducción

Permite maximizar cierta función lineal dado un conjunto de restricciones lineales.

### Algoritmo

El algoritmo opera con programas lineales en la siguiente forma canónica: maximizar  $z = c^T x$  sujeta a  $Ax \leq b, x \geq 0$ .

Por ejemplo, si  $c = (2, -1)$ ,  $A = \begin{bmatrix} 1 & 0 \end{bmatrix}$  y  $b = (5)$ , buscamos maximizar  $z = 2x_1 - x_2$  sujeta a  $x_1 \leq 5$  y  $x_i \geq 0$ .

### Detalles implementativos

Canonizar si hace falta.

Para obtener soluciones negativas, realizar el cambio de variable  $x_i = x'_i + \text{INF}$ .

Si la desigualdad no incluye igual, solo menor, **no usar epsilon** al agregarla. Esto ya es considerado por el código.

```

1  const double EPS = 1e-5;
2  // if inequality is strictly less than (< vs <=), do not use EPS! this
   case is covered in the code
3  namespace Simplex {
4      vi X,Y;
5      vector<vector<double>> > A;
6      vector<double> b,c;
7      double z;
8      int n,m;
9      void pivot(int x,int y){
10         swap(X[Y],Y[X]);
11         b[X]/=A[X][Y];
12         forn(i,m)if(i!=y)A[X][i]/=A[X][Y];
13         A[X][Y]=1/A[X][Y];
14         forn(i,n)if(i!=x&&abs(A[i][Y])>EPS){
15             b[i]-=A[i][Y]*b[X];
16             forn(j,m)if(j!=y)A[i][j]-=A[i][Y]*A[X][j];
17             A[i][Y]=-A[i][Y]*A[X][Y];
18         }
19         z+=c[Y]*b[X];

```

```

20     forn(i,m)if(i!=y)c[i]-=c[Y]*A[X][i];
21     c[Y]=-c[Y]*A[X][Y];
22 }
23 pair<double,vector<double>> simplex( // maximize c^T x s.t. Ax<=b,
   x>=0
24     vector<vector<double>> > _A, vector<double> _b, vector<double>
   > _c){
25     // returns pair (maximum value, solution vector)
26     A=_A;b=_b;c=_c;
27     n=si(b);m=si(c);z=0.;
28     X=vi(m);Y=vi(n);
29     forn(i,m)X[i]=i;
30     forn(i,n)Y[i]=i+m;
31     while(1){
32         int x=-1,y=-1;
33         double mn=-EPS;
34         forn(i,n)if(b[i]<mn)mn=b[i],x=i;
35         if(x<0)break;
36         forn(i,m)if(A[X][i]<-EPS){y=i;break;}
37         assert(y>=0); // no solution to Ax<=b
38         pivot(x,y);
39     }
40     while(1){
41         int x=-1,y=-1;
42         double mx=EPS;
43         forn(i,m)if(c[i]>mx)mx=c[i],y=i;
44         if(y<0)break;
45         double mn=1e200;
46         forn(i,n)if(A[i][Y]>EPS&&b[i]/A[i][Y]<mn)mn=b[i]/A[i][Y],x=i;
47         ;
48         assert(x>=0); // c^T x is unbounded
49         pivot(x,y);
50     }
51     vector<double> r(m);
52     forn(i,n)if(Y[i]<m)r[Y[i]]=b[i];
53     return mp(z,r);
54 };

```

## 7.26. Tablas y cotas (Primos, Divisores, Factoriales, etc)

### Factoriales

0! = 1	11! = 39.916.800
1! = 1	12! = 479.001.600 ( $\in \text{int}$ )
2! = 2	13! = 6.227.020.800
3! = 6	14! = 87.178.291.200
4! = 24	15! = 1.307.674.368.000
5! = 120	16! = 20.922.789.888.000
6! = 720	17! = 355.687.428.096.000
7! = 5.040	18! = 6.402.373.705.728.000
8! = 40.320	19! = 121.645.100.408.832.000
9! = 362.880	20! = 2.432.902.008.176.640.000 ( $\in \text{tint}$ )
10! = 3.628.800	21! = 51.090.942.171.709.400.000
max signed tint = 9.223.372.036.854.775.807	
max unsigned tint = 18.446.744.073.709.551.615	

### Primos

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109  
 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229  
 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353  
 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479  
 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617  
 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757  
 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907  
 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033  
 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151  
 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277  
 1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399  
 1409 1423 1427 1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493  
 1499 1511 1523 1531 1543 1549 1553 1559 1567 1571 1579 1583 1597 1601 1607 1609  
 1613 1619 1621 1627 1637 1657 1663 1667 1669 1693 1697 1699 1709 1721 1723 1733  
 1741 1747 1753 1759 1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871  
 1873 1877 1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997  
 1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081

### Primos cercanos a $10^n$

9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079  
 99961 99971 99989 99991 100003 100019 100043 100049 100057 100069  
 999959 999961 999979 999983 1000003 1000033 1000037 1000039  
 9999943 9999971 9999973 9999991 10000019 10000079 10000103 10000121  
 99999941 99999959 99999971 99999989 100000007 100000037 100000039 100000049  
 999999893 999999929 999999937 1000000007 1000000009 1000000021 1000000033

### Cantidad de primos menores que $10^n$

$\pi(10^1) = 4$  ;  $\pi(10^2) = 25$  ;  $\pi(10^3) = 168$  ;  $\pi(10^4) = 1229$  ;  $\pi(10^5) = 9592$   
 $\pi(10^6) = 78.498$  ;  $\pi(10^7) = 664.579$  ;  $\pi(10^8) = 5.761.455$  ;  $\pi(10^9) = 50.847.534$   
 $\pi(10^{10}) = 455.052,511$  ;  $\pi(10^{11}) = 4.118.054.813$  ;  $\pi(10^{12}) = 37.607.912.018$

**Observación:** Una buena aproximación es  $x/\ln(x)$ .

### Divisores

Cantidad de divisores ( $\sigma_0$ ) para *algunos*  $n/\neg\exists n' < n, \sigma_0(n') \geq \sigma_0(n)$

**Referencias:**  $\sigma_0(10^9) = 1344$  y  $\sigma_0(10^{18}) = 103680$

$\sigma_0(60) = 12$  ;  $\sigma_0(120) = 16$  ;  $\sigma_0(180) = 18$  ;  $\sigma_0(240) = 20$  ;  $\sigma_0(360) = 24$   
 $\sigma_0(720) = 30$  ;  $\sigma_0(840) = 32$  ;  $\sigma_0(1260) = 36$  ;  $\sigma_0(1680) = 40$  ;  $\sigma_0(10080) = 72$   
 $\sigma_0(15120) = 80$  ;  $\sigma_0(50400) = 108$  ;  $\sigma_0(83160) = 128$  ;  $\sigma_0(110880) = 144$   
 $\sigma_0(498960) = 200$  ;  $\sigma_0(554400) = 216$  ;  $\sigma_0(1081080) = 256$  ;  $\sigma_0(1441440) = 288$   
 $\sigma_0(4324320) = 384$  ;  $\sigma_0(8648640) = 448$

**Observación:** Una buena aproximación es  $x^{1/3}$ .

Suma de divisores ( $\sigma_1$ ) para *algunos*  $n/\neg\exists n' < n, \sigma_1(n') \geq \sigma_1(n)$

$\sigma_1(96) = 252$  ;  $\sigma_1(108) = 280$  ;  $\sigma_1(120) = 360$  ;  $\sigma_1(144) = 403$  ;  $\sigma_1(168) = 480$   
 $\sigma_1(960) = 3048$  ;  $\sigma_1(1008) = 3224$  ;  $\sigma_1(1080) = 3600$  ;  $\sigma_1(1200) = 3844$   
 $\sigma_1(4620) = 16128$  ;  $\sigma_1(4680) = 16380$  ;  $\sigma_1(5040) = 19344$  ;  $\sigma_1(5760) = 19890$   
 $\sigma_1(8820) = 31122$  ;  $\sigma_1(9240) = 34560$  ;  $\sigma_1(10080) = 39312$  ;  $\sigma_1(10920) = 40320$   
 $\sigma_1(32760) = 131040$  ;  $\sigma_1(35280) = 137826$  ;  $\sigma_1(36960) = 145152$  ;  $\sigma_1(37800) = 148800$   
 $\sigma_1(60480) = 243840$  ;  $\sigma_1(64680) = 246240$  ;  $\sigma_1(65520) = 270816$  ;  $\sigma_1(70560) = 280098$   
 $\sigma_1(95760) = 386880$  ;  $\sigma_1(98280) = 403200$  ;  $\sigma_1(100800) = 409448$   
 $\sigma_1(491400) = 2083200$  ;  $\sigma_1(498960) = 2160576$  ;  $\sigma_1(514080) = 2177280$   
 $\sigma_1(982800) = 4305280$  ;  $\sigma_1(997920) = 4390848$  ;  $\sigma_1(1048320) = 4464096$   
 $\sigma_1(4979520) = 22189440$  ;  $\sigma_1(4989600) = 22686048$  ;  $\sigma_1(5045040) = 23154768$   
 $\sigma_1(9896040) = 44323200$  ;  $\sigma_1(9959040) = 44553600$  ;  $\sigma_1(9979200) = 45732192$

## 8. Grafos

### 8.1. Teoremas y fórmulas

#### 8.1.1. Teorema de Pick

$$A = I + \frac{B}{2} - 1$$

Donde  $A$  es el área,  $I$  es la cantidad de puntos interiores, y  $B$  la cantidad de puntos en el borde.

#### 8.1.2. Formula de Euler

$$v - e + f = k + 1$$

Donde  $v$  es la cantidad de vértices,  $e$  la cantidad de arcos,  $f$  la cantidad de caras y  $k$  la cantidad de componentes conexas.

## 8.2. Dijkstra

```

1 vector<pii> adj[N]; // IMPORTANTE: ver tipo arco
2 //To add an edge (u,v) with cost p use G[u].pb(v,p)
3 ll dist[N];
4 int dad[N];
5 bool seen[N];
6
7 ll dijkstra(int s=0, int t=-1) { //O(|E| log |V|)
8     fill(dist, dist+N, INF);
9     fill(dad, dad+N, -1);
10    fill(seen, seen+N, false);
11
12    priority_queue<pii, vector<pii>, greater<pii>> pq;
13    pq.emplace(0, s); dist[s] = 0;
14
15    while (!pq.empty()){
16        int u = pq.top().snd; pq.pop();
17
18        if (seen[u]) continue;
19        seen[u] = true;
20
21        if (u == t) break;
22
23        for (auto e : adj[u]) {
24            int v, p; tie(v, p) = e;
25            if (dist[u] + p < dist[v]) {
26                dist[v] = dist[u] + p;
27                dad[v] = u;
28                pq.emplace(dist[v], v);
29            }
30        }
31    }
32
33    return t != -1 ? dist[t] : 0;
34 }
35 // path generator
36 if (dist[t] < INF)
37     for (int u = t; u != -1; u = dad[u])
38         cout << u << "▬\n"[u == s];

```

## 8.3. Bellman-Ford

```

1 vector<ii> G[MAX_N]; //ady. list with pairs (weight, dst)
2 int dist[MAX_N];
3 void bford(int src){ //O(VE)
4     dist[src]=0;
5     for(i, N-1) for(j, N) if(dist[j]!=INF) for(auto u: G[j])
6         dist[u.second]=min(dist[u.second], dist[j]+u.first);
7 }
8
9 bool hasNegCycle(){
10    for(j, N) if(dist[j]!=INF) for(auto u: G[j])
11        if(dist[u.second]>dist[j]+u.first) return true;
12    //inside if: all points reachable from u.snd will have -INF distance(
13        do bfs)
14    return false;
15 }

```

## 8.4. Floyd-Warshall

```

1 //G[i][j] contains weight of edge (i, j) or INF
2 //G[i][i]=0
3 int G[MAX_N][MAX_N];
4 void floyd(){ //O(N^3)
5     for(k, N) for(i, N) if(G[i][k]!=INF) for(j, N) if(G[k][j]!=INF)
6         G[i][j]=min(G[i][j], G[i][k]+G[k][j]);
7 }
8 bool inNegCycle(int v){
9     return G[v][v]<0; }
10 //checks if there's a neg. cycle in path from a to b
11 bool hasNegCycle(int a, int b){
12     for(i, N) if(G[a][i]!=INF && G[i][i]<0 && G[i][b]!=INF)
13         return true;
14     return false;
15 }

```

## 8.5. Kruskal

```

1 struct Edge {
2     int u, v, c;
3     Edge(int _u, int _v, int _c) : u(_u), v(_v), c(_c) {}
4     bool operator < (const Edge &o) const { return c < o.c; }
5 };
6

```

```

7 struct Kruskal {
8     vector<Edge> edges;
9     int n;
10
11     Kruskal(int _n) : n(_n) {}
12     void addEdge(int u, int v, int c) { edges.pb(u, v, c); }
13
14     ll build() {
15         sort(all(edges));
16
17         UF uf(n);
18         ll cost = 0;
19         for (Edge &edge : edges) {
20             if (uf.join(edge.u, edge.v)) {
21                 cost += edge.c;
22             }
23         }
24         return cost;
25     }
26 };

```

## 8.6. Prim

```

1 bool taken[MAXN];
2 priority_queue<ii, vector<ii>, greater<ii> > pq; //min heap
3 void process(int v){
4     taken[v]=true;
5     forall(e, G[v])
6         if(!taken[e->second]) pq.push(*e);
7 }
8
9 ll prim(){
10     zero(taken);
11     process(0);
12     ll cost=0;
13     while(sz(pq)){
14         ii e=pq.top(); pq.pop();
15         if(!taken[e.second]) cost+=e.first, process(e.second);
16     }
17     return cost;
18 }

```

## 8.7. 2-SAT + Tarjan SCC

```

1 //We have a vertex representing a var and other for his negation.
2 //Every edge stored in G represents an implication. To add an equation
3 //of the form a||b, use addor(a, b)
4 //N=max cant var, n=cant var
5 struct SAT {
6     const static int N = 1e5;
7
8     vector<int> adj[N*2];
9     //idx[i]=index assigned in the dfs
10    //lw[i]=lowest index(closer from the root) reachable from i
11    int lw[N*2], idx[N*2], qidx;
12    stack<int> q;
13    int qcmp, cmp[N*2];
14    //value[cmp[i]]=valor de la variable i
15    bool value[N*2+1];
16    int n;
17
18    //remember to CALL INIT!!!
19    void init(int _n) {
20        n = _n;
21        forn(u, 2*n) adj[u].clear();
22    }
23
24    int neg(int x) { return x >= n ? x-n : x+n; }
25    void addor(int a, int b) { adj[neg(a)].pb(b), adj[neg(b)].pb(a); }
26
27    void tjn(int v){
28        lw[v]=idx[v]=++qidx;
29        q.push(v), cmp[v]=-2;
30        for (auto u : adj[v]){
31            if (!idx[u] || cmp[u]==-2){
32                if (!idx[u]) tjn(u);
33                lw[v]=min(lw[v], lw[u]);
34            }
35        }
36        if (lw[v]==idx[v]){
37            int x;
38            do { x=q.top(); q.pop(); cmp[x]=qcmp; } while (x!=v);
39            value[qcmp]= (cmp[neg(v)]<0);
40            qcmp++;
41        }
42    }

```

```

43     bool satisf(){ //O(n)
44         memset(idx, 0, sizeof(idx)), qidx=0;
45         memset(cmp, -1, sizeof(cmp)), qcmp=0;
46         forn(i, n){
47             if (!idx[i]) tjn(i);
48             if (!idx[neg(i)]) tjn(neg(i));
49         }
50         forn(i, n) if (cmp[i]==cmp[neg(i)]) return false;
51         return true;
52     }
53 };

```

## 8.8. Kosaraju

```

1 struct Kosaraju {
2     static const int default_sz = 1e5+10;
3     int n;
4     vector<vi> G, revG, C, ady; // ady is the condensed graph
5     vi used, where;
6     Kosaraju(int sz = default_sz){
7         n = sz;
8         G.assign(sz, vi());
9         revG.assign(sz, vi());
10        used.assign(sz, 0);
11        where.assign(sz, -1);
12    }
13    void addEdge(int a, int b){ G[a].pb(b); revG[b].pb(a); }
14    void dfsNormal(vi &F, int u){
15        used[u] = true;
16        for (int v : G[u]) if(!used[v])
17            dfsNormal(F, v);
18        F.pb(u);
19    }
20    void dfsRev(vi &F, int u){
21        used[u] = true;
22        for (int v : revG[u]) if(!used[v])
23            dfsRev(F, v);
24        F.pb(u);
25    }
26    void build(){
27        vi T;
28        fill(all(used), 0);
29        forn(i, n) if(!used[i]) dfsNormal(T, i);

```

```

30        reverse(all(T));
31        fill(all(used), 0);
32        for (int u : T)
33            if(!used[u]){
34                vi F;
35                dfsRev(F, u);
36                for (int v : F) where[v] = si(C);
37                C.pb(F);
38            }
39        ady.resize(si(C)); // Create edges between condensed nodes
40        forn(u, n) for(int v : G[u]){
41            if(where[u] != where[v]){
42                ady[where[u]].pb(where[v]);
43            }
44        }
45        forn(u, si(C)){
46            sort(all(ady[u]));
47            ady[u].erase(unique(all(ady[u])), ady[u].end());
48        }
49    }
50 };

```

## 8.9. Articulation Points

```

1 int N;
2 vector<int> G[1000000];
3 //V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
4 int qV, V[1000000], L[1000000], P[1000000];
5 void dfs(int v, int f){
6     L[v]=V[v]=++qV;
7     for(auto u: G[v])
8         if(!V[u]){
9             dfs(u, v);
10            L[v] = min(L[v], L[u]);
11            P[v] += L[u]>=V[v];
12        }
13        else if(u!=f)
14            L[v]=min(L[v], V[u]);
15    }
16 int cantart(){ //O(n)
17     qV=0;
18     zero(V), zero(P);
19     dfs(1, 0); P[1]--;

```

```

20 int q=0;
21 for(i, N) if(P[i]) q++;
22 return q;
23 }

```

## 8.10. Comp. Biconexas y Puentes

```

1 struct bridge {
2     struct edge {
3         int u,v,comp;
4         bool bridge;
5     };
6
7     int n,t,nbc;
8     vi d,b,comp;
9     stack<int> st;
10    vector<vi> adj;
11    vector<edge> e;
12
13    bridge(int n=0): n(n) {
14        adj = vector<vi>(n);
15        e.clear();
16        initDfs();
17    }
18
19    void initDfs() {
20        d = vi(n), b = vi(n), comp = vi(n);
21        for(i,n) d[i] = -1;
22        nbc = t = 0;
23    }
24
25    void addEdge(int u, int v) {
26        adj[u].pb(si(e)); adj[v].pb(si(e));
27        e.pb((edge){u,v,-1,false});
28    }
29
30    //d[i]=id de la dfs
31    //b[i]=lowest id reachable from i
32    void dfs(int u=0, int pe=-1) {
33        b[u] = d[u] = t++;
34        comp[u] = pe != -1;
35
36        for(int ne : adj[u]) {

```

```

37            if(ne == pe) continue;
38            int v = e[ne].u ^ e[ne].v ^ u;
39            if(d[v] == -1) {
40                st.push(ne);
41                dfs(v,ne);
42                if(b[v] > d[u]) e[ne].bridge = true; // bridge
43                if(b[v] >= d[u]) { // art
44                    int last;
45                    do {
46                        last = st.top(); st.pop();
47                        e[last].comp = nbc;
48                    } while(last != ne);
49                    nbc++, comp[u]++;
50                }
51                b[u] = min(b[u], b[v]);
52            }
53            else if(d[v] < d[u]) { // back edge
54                st.push(ne);
55                b[u] = min(b[u], d[v]);
56            }
57        }
58    }
59 };

```

## 8.11. LCA + Climb

```

1 const int MAXN=100001;
2 const int LOGN=20;
3 //f[v][k] holds the 2^k father of v
4 //L[v] holds the level of v
5 int N, f[MAXN][LOGN], L[MAXN]; //INICIALIZAR N!!!!!!!!!!!!!!
6 //call before build:
7 void dfs(int u, int fa=-1, int lvl=0){ // generate required data
8     f[u][0]=fa, L[u]=lvl;
9     for (auto v : G[u]) if (v != fa) dfs(v, u, lvl+1);
10 }
11 void build(){ //f[i][0] must be filled previously, 0<nlgn
12     for(k, LOGN-1) for(i, N) if (f[i][k] != -1) f[i][k+1]=f[f[i][k]][k];
13 }
14 #define lg(x) (31-__builtin_clz(x))//=floor(log2(x))
15 int climb(int a, int d){//O(lgn)
16     if(!d) return a;
17     dforsn(i, 0, lg(L[a])+1) if(1<<i<=d) a=f[a][i], d-=1<<i;

```

```

18     return a;
19 }
20 int lca(int a, int b){//O(lgn)
21     if (L[a]<L[b]) swap(a, b);
22     a = climb(a, L[a]-L[b]);
23     if (a==b) return a;
24     dforsn(i, 0, lg(L[a])+1) if (f[a][i]!=f[b][i]) a=f[a][i], b=f[b][i];
25     return f[a][0];
26 }
27 int dist(int a, int b) { //returns distance between nodes
28     return L[a]+L[b]-2*L[lca(a, b)];
29 }

```

## 8.12. Heavy Light Decomposition

```

1 // Usa RMQ Dynamic
2 // ATENCION: valores en nodos. Ver comments para valores en arcos.
3 template <int V, class T>
4 class HeavyLight {
5     int parent[V], heavy[V], depth[V];
6     int root[V], treePos[V];
7     RMQ<V, T, T> tree;
8
9     template <class G>
10     int dfs(const G& graph, int v) {
11         int size = 1, maxSubtree = 0;
12         for (int u : graph[v]) if (u != parent[v]) {
13             parent[u] = v;
14             depth[u] = depth[v] + 1;
15             int subtree = dfs(graph, u);
16             if (subtree > maxSubtree) heavy[v] = u, maxSubtree =
                subtree;
17             size += subtree;
18         }
19         return size;
20     }
21
22     template <class BinaryOperation>
23     void processPath(int u, int v, BinaryOperation op) {
24         for (; root[u] != root[v]; v = parent[root[v]]) {
25             if (depth[root[u]] > depth[root[v]]) swap(u, v);
26             op(treePos[root[v]], treePos[v] + 1);
27         }

```

```

28         if (depth[u] > depth[v]) swap(u, v);
29         // ATENCION: para valores almacenados en arcos: cambiar por
                op(treePos[u]+1, treePos[v]+1)
30         op(treePos[u], treePos[v] + 1);
31     }
32
33 public:
34     // ATENCION: grafo como vector<vector<int>>
35     template <class G>
36     void init(const G& graph) {
37         int n = si(graph);
38         fill_n(heavy, n, -1);
39         parent[0] = -1;
40         depth[0] = 0;
41         dfs(graph, 0);
42         for (int i = 0, currentPos = 0; i < n; ++i)
43             if (parent[i] == -1 || heavy[parent[i]] != i)
44                 for (int j = i; j != -1; j = heavy[j]) {
45                     root[j] = i;
46                     treePos[j] = currentPos++;
47                 }
48         tree.init(n);
49     }
50
51     void set(int v, const T& value) {
52         tree.modify(treePos[v], treePos[v]+1, value);
53     }
54
55     void modifyPath(int u, int v, const T& value) {
56         processPath(u, v, [this, &value](int l, int r) { tree.modify(
                value, l, r); });
57     }
58
59     T queryPath(int u, int v) {
60         T res = T();
61         processPath(u, v, [this, &res](int l, int r) { res += tree.get(l
                , r); });
62         return res;
63     }
64 };

```

## 8.13. Centroid Decomposition

```

1 vector<int> adj[N];
2 bool taken[N]; //poner todos en FALSE al principio!!
3 int padre[N]; //padre de cada nodo en el centroid tree
4 int szt[N];
5 void calcsz(int v, int p) {
6     szt[v] = 1;
7     for (int u : adj[v]) if (u!=p && !taken[u])
8         calcsz(u,v), szt[v]+=szt[u];
9 }
10 void centroid(int v=0, int f=-1, int lvl=0, int tam=-1) { //O(nlogn)
11     if(tam==-1) calcsz(v, -1), tam=szt[v];
12     for(int u : adj[v]) if(!taken[u] && szt[u]*2>=tam)
13         {szt[v]=0; centroid(u, f, lvl, tam); return;}
14     taken[v]=true;
15     padre[v]=f;
16     for(int u : adj[v]) if(!taken[u])
17         centroid(u, v, lvl+1, -1);
18 }

```

## 8.14. Euler Cycle

```

1 int n,m,ars[MAXE], eq;
2 vector<int> G[MAXN]; //fill G,n,m,ars,eq
3 list<int> path;
4 int used[MAXN];
5 bool usede[MAXE];
6 queue<list<int>::iterator> q;
7 int get(int v){
8     while(used[v]<sz(G[v]) && usede[ G[v][used[v]] ]) used[v]++;
9     return used[v];
10 }
11 void explore(int v, int r, list<int>::iterator it){
12     int ar=G[v][get(v)]; int u=v^ars[ar];
13     usede[ar]=true;
14     list<int>::iterator it2=path.insert(it, u);
15     if(u!=r) explore(u, r, it2);
16     if(get(v)<sz(G[v])) q.push(it);
17 }
18 void euler(){
19     zero(used), zero(usede);
20     path.clear();
21     q=queue<list<int>::iterator>();
22     path.push_back(0); q.push(path.begin());

```

```

23     while(sz(q)){
24         list<int>::iterator it=q.front(); q.pop();
25         if(used[*it]<sz(G[*it])) explore(*it, *it, it);
26     }
27     reverse(path.begin(), path.end());
28 }
29 void addEdge(int u, int v){
30     G[u].pb(eq), G[v].pb(eq);
31     ars[eq++]=u^v;
32 }

```

## 8.15. Diametro árbol

```

1 int n;
2 vi adj[N];
3
4 pii farthest(int u, int p = -1) {
5     pii ans = {-1, u};
6
7     for (int v : adj[u])
8         if (v != p)
9             ans = max(ans, farthest(v, u));
10
11     ans.fst++;
12     return ans;
13 }
14
15 int diam(int r) {
16     return farthest(farthest(r).snd).fst;
17 }
18
19 bool path(int s, int e, vi &p, int pre = -1) {
20     p.pb(s);
21     if (s == e) return true;
22
23     for (int v : adj[s])
24         if (v != pre && path(v, e, p, s))
25             return true;
26
27     p.pop_back();
28     return false;
29 }
30

```



```

31 int center(int r) {
32     int s = farthest(r).snd, e = farthest(s).snd;
33     vi p; path(s, e, p);
34     return p[si(p)/2];
35 }

```

## 8.16. Chu-liu

```

1 void visit(graph &h, int v, int s, int r,
2     vector<int> &no, vector< vector<int> > &comp,
3     vector<int> &prev, vector< vector<int> > &next, vector<weight> &mcost,
4     vector<int> &mark, weight &cost, bool &found) {
5     if (mark[v]) {
6         vector<int> temp = no;
7         found = true;
8         do {
9             cost += mcost[v];
10            v = prev[v];
11            if (v != s) {
12                while (comp[v].size() > 0) {
13                    no[comp[v].back()] = s;
14                    comp[s].push_back(comp[v].back());
15                    comp[v].pop_back();
16                }
17            }
18        } while (v != s);
19        forall(j, comp[s]) if (*j != r) forall(e, h[*j])
20            if (no[e->src] != s) e->w -= mcost[ temp[*j] ];
21    }
22    mark[v] = true;
23    forall(i, next[v]) if (no[*i] != no[v] && prev[no[*i]] == v)
24        if (!mark[no[*i]] || *i == s)
25            visit(h, *i, s, r, no, comp, prev, next, mcost, mark, cost, found);
26 }
27 weight minimumSpanningArborescence(const graph &g, int r) {
28     const int n=sz(g);
29     graph h(n);
30     forn(u,n) forall(e,g[u]) h[e->dst].pb(*e);
31     vector<int> no(n);
32     vector<vector<int> > comp(n);
33     forn(u, n) comp[u].pb(no[u] = u);
34     for (weight cost = 0; ;) {

```

```

35     vector<int> prev(n, -1);
36     vector<weight> mcost(n, INF);
37     forn(j,n) if (j != r) forall(e,h[j])
38         if (no[e->src] != no[j])
39             if (e->w < mcost[ no[j] ])
40                 mcost[ no[j] ] = e->w, prev[ no[j] ] = no[e->src];
41     vector< vector<int> > next(n);
42     forn(u,n) if (prev[u] >= 0)
43         next[ prev[u] ].push_back(u);
44     bool stop = true;
45     vector<int> mark(n);
46     forn(u,n) if (u != r && !mark[u] && !comp[u].empty()) {
47         bool found = false;
48         visit(h, u, u, r, no, comp, prev, next, mcost, mark, cost, found);
49         if (found) stop = false;
50     }
51     if (stop) {
52         forn(u,n) if (prev[u] >= 0) cost += mcost[u];
53         return cost;
54     }
55 }
56 }

```

## 8.17. Hungarian

```

1 //Dado un grafo bipartito completo con costos no negativos, encuentra el
2   matching perfecto de minimo costo.
3 tipo cost[N][N], lx[N], ly[N], slack[N]; //llenar: cost=matriz de
4   adyacencia
5 int n, max_match, xy[N], yx[N], slackx[N], prev2[N]; //n=cantidad de nodos
6 bool S[N], T[N]; //sets S and T in algorithm
7 void add_to_tree(int x, int prevx) {
8     S[x] = true, prev2[x] = prevx;
9     forn(y, n) if (lx[x] + ly[y] - cost[x][y] < slack[y] - EPS)
10         slack[y] = lx[x] + ly[y] - cost[x][y], slackx[y] = x;
11 }
12 void update_labels(){
13     tipo delta = INF;
14     forn (y, n) if (!T[y]) delta = min(delta, slack[y]);
15     forn (x, n) if (S[x]) lx[x] -= delta;
16     forn (y, n) if (T[y]) ly[y] += delta; else slack[y] -= delta;
17 }
18 void init_labels(){

```

```

17 zero(lx), zero(ly);
18 forn (x,n) forn(y,n) lx[x] = max(lx[x], cost[x][y]);
19 }
20 void augment() {
21     if (max_match == n) return;
22     int x, y, root, q[N], wr = 0, rd = 0;
23     memset(S, false, sizeof(S)), memset(T, false, sizeof(T));
24     memset(prev2, -1, sizeof(prev2));
25     forn (x, n) if (xy[x] == -1){
26         q[wr++] = root = x, prev2[x] = -2;
27         S[x] = true; break; }
28     forn (y, n) slack[y] = lx[root] + ly[y] - cost[root][y], slackx[y] =
        root;
29     while (true){
30         while (rd < wr){
31             x = q[rd++];
32             for (y = 0; y < n; y++) if (cost[x][y] == lx[x] + ly[y] && !T[y]){
33                 if (yx[y] == -1) break; T[y] = true;
34                 q[wr++] = yx[y], add_to_tree(yx[y], x); }
35             if (y < n) break; }
36         if (y < n) break;
37         update_labels(), wr = rd = 0;
38         for (y = 0; y < n; y++) if (!T[y] && slack[y] == 0){
39             if (yx[y] == -1){x = slackx[y]; break;}
40             else{
41                 T[y] = true;
42                 if (!S[yx[y]]) q[wr++] = yx[y], add_to_tree(yx[y], slackx[y]);
43             }
44             if (y < n) break; }
45         if (y < n){
46             max_match++;
47             for (int cx = x, cy = y, ty; cx != -2; cx = prev2[cx], cy = ty)
48                 ty = xy[cx], yx[cy] = cx, xy[cx] = cy;
49             augment(); }
50     }
51     tipo hungarian(){
52         tipo ret = 0; max_match = 0, memset(xy, -1, sizeof(xy));
53         memset(yx, -1, sizeof(yx)), init_labels(), augment(); //steps 1-3
54         forn (x,n) ret += cost[x][xy[x]]; return ret;
55     }

```

## 8.18. Dynamic Connectivity

```

1 struct UnionFind {
2     int n, comp;
3     vector<int> pre, si, c;
4     UnionFind(int n=0):n(n), comp(n), pre(n), si(n, 1) {
5         forn(i,n) pre[i] = i; }
6     int find(int u){return u==pre[u]?u:find(pre[u]);}
7     bool merge(int u, int v) {
8         if((u=find(u))==v) return false;
9         if(si[u]<si[v]) swap(u, v);
10        si[u]+=si[v], pre[v]=u, comp--, c.pb(v);
11        return true;
12    }
13    int snap(){return sz(c);}
14    void rollback(int snap){
15        while(sz(c)>snap){
16            int v = c.back(); c.pop_back();
17            si[pre[v]] -= si[v], pre[v] = v, comp++;
18        }
19    }
20 };
21 enum {ADD,DEL,QUERY};
22 struct Query {int type,u,v;};
23 struct DynCon {
24     vector<Query> q;
25     UnionFind dsu;
26     vector<int> match,res;
27     map<ii,int> last; //se puede no usar cuando hay identificador para
        cada arista (mejora poco)
28     DynCon(int n=0):dsu(n){}
29     void add(int u, int v) {
30         if(u>v) swap(u,v);
31         q.pb((Query){ADD, u, v}), match.pb(-1);
32         last[ii(u,v)] = sz(q)-1;
33     }
34     void remove(int u, int v) {
35         if(u>v) swap(u,v);
36         q.pb((Query){DEL, u, v});
37         int prev = last[ii(u,v)];
38         match[prev] = sz(q)-1;
39         match.pb(prev);
40     }
41     void query() { //podria pasarle un puntero donde guardar la respuesta
42         q.pb((Query){QUERY, -1, -1}), match.pb(-1);}

```

```

43 void process() {
44     forn(i,sz(q)) if (q[i].type == ADD && match[i] == -1) match[i] =
45         sz(q);
46     go(0,sz(q));
47 }
48 void go(int l, int r) {
49     if(l+1==r){
50         if (q[l].type == QUERY)//Aqui responder la query usando el
51             dsu!
52         res.pb(dsu.comp);//aqui query=cantidad de componentes
53             conexas
54         return;
55     }
56     int s=dsu.snap(), m = (l+r) / 2;
57     forr(i,m,r) if(match[i]!=-1 && match[i]<l) dsu.merge(q[i].u, q[i]
58         ].v);
59     go(l,m);
60     dsu.rollback(s);
61     s = dsu.snap();
62     forr(i,l,m) if(match[i]!=-1 && match[i]>=r) dsu.merge(q[i].u, q[
63         i].v);
64     go(m,r);
65     dsu.rollback(s);
66 }
67 }dc;

```

## 9. Flujo

### 9.1. Dinic

```

1 // Corte minimo: vertices con dist[v]>=0 (del lado de src) VS. dist[v]
2 // ==-1 (del lado del dst)
3 // Para el caso de la red de Bipartite Matching (Sean V1 y V2 los
4 // conjuntos mas proximos a src y dst respectivamente):
5 // Reconstruir matching: para todo v1 en V1 ver las aristas a vertices
6 // de V2 con it->f>0, es arista del Matching
7 // Min Vertex Cover: vertices de V1 con dist[v]==-1 + vertices de V2 con
8 // dist[v]>0
9 // MAXN Independent Set: tomar los vertices NO tomados por el Min Vertex
10 // Cover
11 // MAXN Clique: construir la red de G complemento (debe ser bipartito!)
12 // y encontrar un MAXN Independet Set
13 // Min Edge Cover: tomar las aristas del matching + para todo vertices

```

```

8 no cubierto hasta el momento, tomar cualquier arista de el
9 // Tiempos!  $O(V^2 \cdot E)$  en general.  $O(\sqrt{V} \cdot E)$  en matching bipartito.  $O(\min(E^{2/3}, V^{1/2}) \cdot E)$  si capacidad 1.
10 template<int MAXN>
11 struct dinic {
12
13     struct edge {
14         int u,v; ll c,f;
15         ll r() { return c-f; }
16     };
17
18     static const ll INF = 1e18;
19
20     int N,S,T;
21     vector<edge> e;
22     //edge red[MAXN] [MAXN];
23     vi adjG[MAXN];
24
25     void reset() {
26         forn(u,N) for (auto ind : adjG[u]) {
27             auto &ei = e[ind];
28             ei.f = 0;
29         }
30     }
31
32     void initGraph(int n, int s, int t) {
33         N = n; S = s; T = t;
34         e.clear();
35         forn(u,N) adjG[u].clear();
36     }
37
38     void addEdge(int u, int v, ll c) {
39         adjG[u].pb(si(e)); e.pb((edge){u,v,c,0});
40         adjG[v].pb(si(e)); e.pb((edge){v,u,0,0});
41     }
42
43     int dist[MAXN];
44     bool dinic_bfs() {
45         forn(u,N) dist[u] = -1;
46         queue<int> q; q.push(S); dist[S] = 0;
47         while (!q.empty()) {
48             int u = q.front(); q.pop();

```

```

49     for (auto ind : adjG[u]) {
50         auto &ei = e[ind];
51         int v = ei.v;
52         if (dist[v] != -1 || ei.r() == 0) continue;
53         dist[v] = dist[u] + 1;
54         q.push(v);
55     }
56 }
57 return dist[T] != -1;
58 }
59
60 ll dinic_dfs(int u, ll cap) {
61     if (u == T) return cap;
62
63     ll res = 0;
64     for (auto ind : adjG[u]) {
65         auto &ei = e[ind], &ej = e[ind^1];
66         int v = ei.v;
67         if (ei.r() && dist[v] == dist[u] + 1) {
68             ll send = dinic_dfs(v, min(cap, ei.r()));
69             ei.f += send; ej.f -= send;
70             res += send; cap -= send;
71             if (cap == 0) break;
72         }
73     }
74     if (res == 0) dist[u] = -1;
75     return res;
76 }
77
78 ll flow() {
79     ll res = 0;
80     while (dinic_bfs()) res += dinic_dfs(S, INF);
81     return res;
82 }
83
84 vi cut() {
85     dinic_bfs();
86     vi ans;
87     for (auto u : adjG[S]) if (dist[e[u].v] == -1) ans.pb(e[u].v);
88     for (auto u : adjG[T]) if (dist[e[u].v] != -1) ans.pb(e[u].v);
89     return ans;
90 }
91

```

```

92     vi indep() {
93         dinic_bfs();
94         vi ans;
95         for (auto u : adjG[S]) if (dist[e[u].v] != -1) ans.pb(e[u].v);
96         for (auto u : adjG[T]) if (dist[e[u].v] == -1) ans.pb(e[u].v);
97         return ans;
98     }
99 };

```

## 9.2. Konig

```

1  // asume que el dinic YA ESTA tirado
2  // asume que nodes-1 y nodes-2 son la fuente y destino
3  int match[maxnodes]; // match[v]=u si u-v esta en el matching, -1 si v
4                          // no esta matcheado
5  int s[maxnodes]; // numero de la bfs del koning
6  queue<int> kq;
7  // s[e]%2==1 o si e esta en V1 y s[e]==-1-> lo agarras
8  void koning() { // O(n)
9      for(v, nodes-2) s[v] = match[v] = -1;
10     for(v, nodes-2) forall(it, g[v]) if (it->to < nodes-2 && it->f > 0)
11         { match[v] = it->to; match[it->to] = v; }
12     for(v, nodes-2) if (match[v] == -1) { s[v] = 0; kq.push(v); }
13     while(!kq.empty()) {
14         int e = kq.front(); kq.pop();
15         if (s[e] % 2 == 1) {
16             s[match[e]] = s[e] + 1;
17             kq.push(match[e]);
18         } else {
19             forall(it, g[e]) if (it->to < nodes-2 && s[it->to] == -1) {
20                 s[it->to] = s[e] + 1;
21                 kq.push(it->to);
22             }
23         }
24     }
25 }

```

## 9.3. Edmonds Karp's

```

1  #define MAX_V 1000
2  #define INF 1e9
3  //special nodes
4  #define SRC 0

```

```

5 #define SNK 1
6 map<int, int> G[MAX_V]; //limpiar esto
7 //To add an edge use
8 #define add(a, b, w) G[a][b]=w
9 int f, p[MAX_V];
10 void augment(int v, int minE){
11     if(v==SRC) f=minE;
12     else if(p[v]!=-1){
13         augment(p[v], min(minE, G[p[v]][v]));
14         G[p[v]][v]-=f, G[v][p[v]]+=f;
15     }
16 }
17 ll maxflow(){//O(VE^2)
18     ll Mf=0;
19     do{
20         f=0;
21         char used[MAX_V]; queue<int> q; q.push(SRC);
22         zero(used), memset(p, -1, sizeof(p));
23         while(sz(q)){
24             int u=q.front(); q.pop();
25             if(u==SNK) break;
26             forall(it, G[u])
27                 if(it->snd>0 && !used[it->fst])
28                     used[it->fst]=true, q.push(it->fst), p[it->fst]=u;
29         }
30         augment(SNK, INF);
31         Mf+=f;
32     }while(f);
33     return Mf;
34 }

```

#### 9.4. Min-cost Max-flow

```

1 const int MAXN=10000;
2 typedef ll tf;
3 typedef ll tc;
4 const tf INFFLUJO = 1e14;
5 const tc INFCOSTO = 1e14;
6 struct edge {
7     int u, v;
8     tf cap, flow;
9     tc cost;
10    tf rem() { return cap - flow; }

```

```

11 };
12 int nodes; //numero de nodos
13 vector<int> G[MAXN]; // limpiar!
14 vector<edge> e; // limpiar!
15 void addEdge(int u, int v, tf cap, tc cost) {
16     G[u].pb(sz(e)); e.pb((edge){u,v,cap,0,cost});
17     G[v].pb(sz(e)); e.pb((edge){v,u,0,0,-cost});
18 }
19 tc dist[MAXN], mnCost;
20 int pre[MAXN];
21 tf cap[MAXN], mxFlow;
22 bool in_queue[MAXN];
23 void flow(int s, int t) {
24     zero(in_queue);
25     mxFlow=mnCost=0;
26     while(1){
27         fill(dist, dist+nodes, INFCOSTO); dist[s] = 0;
28         memset(pre, -1, sizeof(pre)); pre[s]=0;
29         zero(cap); cap[s] = INFFLUJO;
30         queue<int> q; q.push(s); in_queue[s]=1;
31         while(sz(q)){
32             int u=q.front(); q.pop(); in_queue[u]=0;
33             for(auto it:G[u]) {
34                 edge &E = e[it];
35                 if(E.rem() && dist[E.v] > dist[u] + E.cost + 1e-9){ // ojo EPS
36                     dist[E.v]=dist[u]+E.cost;
37                     pre[E.v] = it;
38                     cap[E.v] = min(cap[u], E.rem());
39                     if(!in_queue[E.v]) q.push(E.v), in_queue[E.v]=1;
40                 }
41             }
42         }
43         if (pre[t] == -1) break;
44         mxFlow +=cap[t];
45         mnCost +=cap[t]*dist[t];
46         for (int v = t; v != s; v = e[pre[v]].u) {
47             e[pre[v]].flow += cap[t];
48             e[pre[v]^1].flow -= cap[t];
49         }
50     }
51 }

```

## 10. Template

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #ifdef LOCAL
5     #define D(a) cerr << #a << " = " << a << endl
6 #else
7     #define D(a)
8     #define cerr false && cerr
9 #endif
10 #define fastio ios_base::sync_with_stdio(0); cin.tie(0)
11 #define dfor(n,i,s) for(int i=int(s);i<int(n);i++)
12 #define for(n,i,s) for(int i=int(s);i<int(n);i++)
13 #define dfor(n,i) for(int i=0;i<int(n);i++)
14 #define for(n,i) for(int i=0;i<int(n);i++)
15 #define all(a) a.begin(),a.end()
16 #define sz(a) int((a).size())
17 #define pb push_back
18 #define mp make_pair
19 #define sn second
20 #define fr first
21 #define endl '\n'
22 using pii = pair<int,int>;
23 using vi = vector<int>;
24 using ll = long long;
25
26 int main() {
27     fastio;
28
29
30     return 0;
31 }

```

## 11. vimrc

```

1 colo desert
2 set number
3 set norelativenumber
4 set autochdir
5 set colorcolumn=80
6 set ignorecase

```

```

7 set showcmd
8 augroup cpp
9     autocmd!
10     autocmd FileType cpp map <f9> :w<enter> :!g++ -std=c++14 -W -Wall -
        Wshadow -Wconversion -DLOCAL -D_GLIBCXX_DEBUG -g3 "%I" -o "a" <
        enter>
11     autocmd FileType cpp map <f5> :!"/a" < a.in <enter>
12     autocmd FileType cpp map <f6> :!"/a" <enter>
13 augroup END
14 set tabstop=4
15 set shiftwidth=4
16 set softtabstop=4
17 set expandtab
18 set smartindent
19 set cindent
20 set clipboard=unnamedplus
21 nmap <c-h> <c-w><c-h>
22 nmap <c-j> <c-w><c-j>
23 nmap <c-k> <c-w><c-k>
24 nmap <c-l> <c-w><c-l>
25 vmap > >gv
26 vmap < <gv
27 map j gj
28 map k gk
29 nnoremap <silent> [b :bp<CR>
30 nnoremap <silent> ]b :bn<CR>
31 nnoremap <silent> [B :bf<CR>
32 nnoremap <silent> ]B :bl<CR>
33 set splitright
34 set nobackup
35 set nowritebackup
36 set noswapfile

```

## 12. misc

```

1 #include <bits/stdc++.h> // Library that includes the most used
    libraries
2 using namespace std; // It avoids the use of std::func(), instead we
    can simply use func()
3
4 ios_base::sync_with_stdio(0); cin.tie(0); // Speeds up considerably the
    read speed, very convenient when the input is large
5

```

```

6 #pragma GCC optimize ("O3") // Asks the compiler to apply more
   optimizations, that way speeding up the program very much!
7
8 Math:
9 max(a,b); // Returns the largest of a and b
10 min(a,b); // Returns the smallest of a and b
11 abs(a,b); // Returns the absolute value of x (integral value)
12 fabs(a,b); // Returns the absolute value of x (double)
13 sqrt(x); // Returns the square root of x.
14 pow(base,exp); // Returns base raised to the power exp
15 ceil(x); // Rounds x upward, returning the smallest integral value that
   is not less than x
16 floor(x); // Rounds x downward, returning the largest integral value
   that is not greater than x
17 exp(x); // Returns the base-e exponential function of x, which is e
   raised to the power x
18 log(x); // Returns the natural logarithm of x
19 log2(x); // Returns the binary (base-2) logarithm of x
20 log10(x); // Returns the common (base-10) logarithm of x
21 modf(double x, double *intpart); /* Breaks x into an integral and a
   fractional part. The integer part is stored in the object
   pointed by intpart, and the fractional part is returned by the function.
   Both parts have the same sign as x. */
23 sin(),cos(),tan(); asin(),acos(),atan(); sinh(),cosh(),tanh(); //
   Trigonometric functions
24 // See http://www.cplusplus.com/reference/cmath/ for more useful math
   functions!
25
26 Strings:
27 s.replace(pos,len,str); // Replaces the portion of the string that
   begins at character pos and spans len characters by str
28 s.replace(start,end,str); // or the part of the string in the range
   between [start,end)
29 s.substr(pos = 0,len = npos); // Returns the substring starting at
   character pos that spans len characters (or until the end of the
   string, whichever comes first).
30 // A value of string::npos indicates all characters until the end of the
   string.
31 s.insert(pos,str); // Inserts str right before the character indicated
   by pos
32 s.erase(pos = 0, len = npos); erase(first,last); erase(iterator p); //
   Erases part of the string
33 s.find(str,pos = 0); // Searches the string for the first occurrence of

```

```

   the sequence specified by its arguments after position pos
34 toupper(char x); // Converts lowercase letter to uppercase. If no such
   conversion is possible, the value returned is x unchanged.
35 tolower(char x); // Converts uppercase letter to lowercase. If no such
   conversion is possible, the value returned is x unchanged.
36
37 Constants:
38 INT_MAX, INT_MIN, LLONG_MIN, LLONG_MAX, ULLONG_MAX
39 const int maxn = 1e5; // 1e5 means 1x10^5, C++ features scientific
   notation. e.g.: 4.56e6 = 4.560.000, 7.67e-5 = 0.0000767.
40 const double pi = acos(-1); // Compute Pi
41
42 Algorithms:
43 swap(a,b); // Exchanges the values of a and b
44 minmax(a,b); // Returns a pair with the smallest of a and b as first
   element, and the largest as second.
45 minmax({1,2,3,4,5}); // Returns a pair with the smallest of all the
   elements in the list as first element and the largest as second
46 next_permutation(a,a+n); // Rearranges the elements in the range [first,
   last) into the next lexicographically greater permutation.
47 reverse(first,last); // Reverses the order of the elements in the range
   [first,last)
48 rotate(first,middle,last) // Rotates the order of the elements in the
   range [first,last), in such a way that the element pointed by middle
   becomes the new first element
49 remove_if(first,last,func) // Returns an iterator to the element that
   follows the last element not removed. The range between first and
   this iterator includes all the elements in the sequence for which
   func does not return true.
50 // See http://www.cplusplus.com/reference/algorithm/ for more useful
   algorithms!
51
52 Binary search:
53 int a[] = {1, 2, 4, 7, 10, 12}, x = 5;
54 int *l = lower_bound(a,a+6,x); // lower_bound: Returns the first element
   that is not less than x
55 cout << (l == a+5 ? -1 : *l) << endl;
56 cout << x << (binary_search(a,a+6,x)? "is\n": "isn't\n"); //
   binary_search: Returns true if any element in the range [first,last)
   is equivalent to x, and false otherwise.
57 vi v(a,a+6);
58 auto i = upper_bound(v.begin(),v.end(),x) // upper_bound: Returns the
   first element that is greater than x

```



```

59 Random numbers:
60 mt19937_64 rng(time(0)); //if TLE use 32 bits: mt19937
61 ll rnd(ll a, ll b) { return a + rng()%(b-a+1); }
62 Unhackable seed (Codeforces):
63 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
64 random_shuffle(a,a+n,rng); // Rearranges the elements in the range [
65     first,last) randomly
66
67 Sorting:
68 sort(a,a+n,comp); /* Sorts the elements in the range [first,last) into
69     ascending order.
70 The third parameter is optional, if greater<Type> is passed then the
71     array is sorted in descending order.
72 comp: Binary function that accepts two elements in the range as
73     arguments, and returns a value convertible to bool. The value
74     returned
75 indicates whether the element passed as first argument is considered to
76     go before the second in the specific strict weak ordering
77 it defines. The function shall not modify any of its arguments. This can
78     either be a function pointer or a function object. */
79 stable_sort(a,a+n); // Sorts the elements in the range [first,last) into
80     ascending order, like sort, but stable_sort preserves the relative
81     order of the elements with equivalent values.
82 sort(a.begin(),a.end()); // Sort using container ranges
83 sort(a,a+n,[](const node &a, const node &b){ // Custom sort with a "
84     lambda expression": an unnamed function object capable of capturing
85     variables in scope.
86     return a.x < b.x || (a.x == b.x && a.y < b.y); // Custom sort
87 }); // see https://en.cppreference.com/w/cpp/language/lambda for more
88     details
89 bool myfunction(const edge &a, const edge &b){ return a.w < b.w; }
90 sort(myvector.begin()+4, myvector.end(), myfunction); // Using a
91     function as a comparator
92 struct comp{ bool operator()(const edge &a, const edge &b){ return a.w <
93     b.w; } };
94 multiset<edge,comp> l; // Using a function object as comparator:
95 bool operator<(const edge &a, const edge &b){ return a.w < b.w; } //
96     Operator definition (it can be inside or outside the class)
97
98 Input/output handling:
99 freopen("input.txt","r",stdin); // Sets the standard input stream (
100     keyboard) to the file input.txt

```

```

86 freopen("output.txt","w",stdout); // Sets the standard output stream (
87     screen) to the file output.txt
88 getline(cin,str); // Reads until an end of line is reached from the
89     input stream into str. If we use cin >> str it would read until it
90     finds a whitespace
91 // Make an extra call if we previously read another thing from the input
92     stream (otherwise it wouldn't work as expected)
93 cout << fixed << setprecision(n); // Sets the decimal precision to be
94     used to format floating-point values on output operations to n
95 cout << setw(n); // Sets the field width to be used on output operations
96     to n
97 cout << setfill('0'); // Sets c as the stream's fill character
98
99 Increment stack size to the maximum (Linux):
100 // #include <sys/resource.h>
101 struct rlimit rl;
102 getrlimit(RLIMIT_STACK, &rl);
103 rl.rlim_cur = rl.rlim_max;
104 setrlimit(RLIMIT_STACK, &rl);
105
106 String to int and vice versa (might be very useful to parse odd things):
107 template <typename T> string to_str(T str) { stringstream s; s << str;
108     return s.str(); }
109 template <typename T> int to_int(T n) { int r; stringstream s; s << n; s
110     >> r; return r; }
111
112 C++11:
113 to_string(num) // returns a string with the representation of num
114 stoi,stoll,stod,stold // string to int,ll,double & long double
115     respectively
116
117 Print structs with cout:
118 ostream& operator << (ostream &o, pto &p) {
119     o << p.x << 'L' << p.y;
120     return o;
121 }

```

## 13. Ayudamemoria

### Cant. decimales

```

1 #include <iomanip>
2 cout << setprecision(2) << fixed;

```

## Rellenar con espacios(para justificar)

```
1 | #include <iomanip>
2 | cout << setfill('␣') << setw(3) << 2 << endl;
```

## Comparación de Doubles

```
1 | const double EPS = 1e-9;
2 | x == y  <=> fabs(x-y) < EPS
3 | x > y   <=> x > y + EPS
4 | x >= y  <=> x > y - EPS
```

## Limites

```
1 | #include <limits>
2 | numeric_limits<T>
3 |     ::max()
4 |     ::min()
5 |     ::epsilon()
```

## Muahaha

```
1 | #include <signal.h>
2 | void divzero(int p){
3 |     while(true);}
4 | void segm(int p){
5 |     exit(0);}
6 | //in main
7 | signal(SIGFPE, divzero);
8 | signal(SIGSEGV, segm);
```

## Mejorar velocidad 2

```
1 | //Solo para enteros positivos
2 | inline void Scanf(int& a){
3 |     char c = 0;
4 |     while(c<33) c = getc(stdin);
5 |     a = 0;
6 |     while(c>33) a = a*10 + c - '0', c = getc(stdin);
7 | }
```

## Leer del teclado

```
1 | freopen("/dev/tty", "a", stdin);
```

## Iterar subconjunto

```
1 | for(int sbm=0; sbm<1<sbm; sbm++)
```

## File setup

```
1 | // tambien se pueden usar comas: {a, x, m, l}
2 | touch {a..l}.in; tee {a..l}.cpp < template.cpp
```

## Releer String

```
1 | string s; int n;
2 | getline(cin, s);
3 | stringstream leer(s);
4 | while(leer >> n){
5 |     // do something ...
6 | }
```