

# Referencia - ICPC

Mathgic

Noviembre 2024

Github

ACTUALIZADO HASTA LA SECCIÓN 10. Falta 11, 12, etc.

# Índice

0.1. OJO . . . . .	4
<b>1. Estructuras básicas</b>	<b>4</b>
1.1. Min stack . . . . .	4
1.2. Min queue . . . . .	4
1.3. Heap actualizable . . . . .	4
<b>2. Ordenamiento</b>	<b>6</b>
2.1. Merge sort . . . . .	6
<b>3. Matemáticas</b>	<b>6</b>
3.1. Criba de Eratóstenes . . . . .	6
3.1.1. Criba . . . . .	6
3.1.2. Criba sobre un rango . . . . .	6
3.1.3. Criba segmentada . . . . .	7
3.1.4. Criba lineal . . . . .	7
3.2. Algoritmo extendido de Euclides . . . . .	8
3.3. Solución de ecuaciones diofánticas lineales . . . . .	8
3.4. Funciones multiplicativas . . . . .	9
3.4.1. Función Phi de Euler . . . . .	9
3.4.2. Función sigma . . . . .	9
3.4.3. Función de Moebius . . . . .	10
3.5. Exponenciación binaria . . . . .	10
3.6. Transformada de Fourier . . . . .	11
3.6.1. FFT . . . . .	11
3.6.2. Multiplicar polinomios . . . . .	11
<b>4. Sparse table</b>	<b>11</b>
<b>5. Fenwick Tree</b>	<b>12</b>
<b>6. Segment Tree</b>	<b>13</b>
6.1. Actualizaciones puntuales . . . . .	13
6.2. Actualizaciones sobre rangos . . . . .	14
<b>7. Sqrt decomposition</b>	<b>14</b>
7.1. Algoritmo de MO . . . . .	14
<b>8. DSU</b>	<b>15</b>
<b>9. Grafos</b>	<b>16</b>
9.1. Caminos mínimos . . . . .	16
9.1.1. Dijkstra . . . . .	16
9.1.2. Bellman-Ford . . . . .	16
9.1.3. Floyd-Warshall . . . . .	17
9.1.4. Johnson's algorithm . . . . .	17
9.2. Árboles . . . . .	17
9.2.1. MST . . . . .	17
9.2.2. LCA . . . . .	18
9.2.3. Sack . . . . .	19
9.3. Máximo flujo . . . . .	20
9.3.1. Algunos problemas de flujos . . . . .	20
9.3.2. Edmonds-Karp . . . . .	20
9.3.3. Dinic . . . . .	23
9.4. SCC . . . . .	24
9.4.1. Kosajaru . . . . .	24
9.5. 2-Sat . . . . .	25

<b>10.Treap</b>	<b>26</b>
<b>11.Strings</b>	<b>27</b>
11.1. KMP . . . . .	27
11.1.1. Autómata de KMP . . . . .	28
11.2. Suffix array . . . . .	28
11.2.1. Construcción . . . . .	28
11.2.2. Prefijo común más largo . . . . .	29
11.3. Aho-Corasick . . . . .	29
11.4. Suffix tree . . . . .	30
<b>12.Geometría</b>	<b>33</b>
12.1. Convex hull . . . . .	33
<b>13.Utilidades</b>	<b>33</b>
13.1. Plantilla tree . . . . .	33
13.2. Números aleatorios . . . . .	34
13.3. Algunas formulas . . . . .	34
<b>14.Bitmask</b>	<b>34</b>
14.1. Útiles . . . . .	34
14.2. Iterar . . . . .	35
14.3. Gospers' Hack . . . . .	35
<b>15.Máximo de funciones</b>	<b>35</b>
15.1. Li-Chao Tree . . . . .	35

## 0.1. OJO

- a) Se usan macros (MAXN, LOGN, etc) con arreglos estáticos para más comodidad, pero puede causar RTE o MLE cuando los valores son grandes. Pensar en usar `vector<>` (STL) cuando sea conveniente.
- b) Temario (no oficial): <https://youkn0wwho.academy/topic-list>.
- c) agréguenle errores/consejos que hay que tener en cuenta sobre las implementaciones y no estemos mucho tiempo tratando de encontrar el error.

## 1. Estructuras básicas

### 1.1. Min stack

```
1  template<typename T> struct min_stack{
2      stack<pair<T, T>> st;
3      min_stack(){}
4      min_stack(const T &MAXVAL){init(MAXVAL);}
5      void init(const T &MAXVAL){st.push(make_pair(MAXVAL, MAXVAL));}
6      void push(const T &v){st.push(make_pair(v, min(v, st.top().second)));}
7      T top(){return st.top().first;}
8      void pop(){if(st.size() > 1)st.pop();}
9      T minV(){return st.top().second;}
10     int size(){return st.size() - 1;}
11     bool empty(){return size() == 0;}
12 };
```

### 1.2. Min queue

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

### 1.3. Heap actualizable

```
1  template<class TPriority, class TKey> class UpdatableHeap{
2  public:
3      UpdatableHeap(){
4          TPriority a;
5          TKey b;
6          nodes.clear();
7          nodes.push_back( make_pair(a, b) );
8      }
9      pair<TPriority, TKey> top() {return nodes[1];}
10     void pop(){
11         if(nodes.size() == 1) return;
```

```

12     TKey k = nodes[1].second;
13     swap_nodes(1, nodes.size() - 1);
14     nodes.pop_back();
15     position.erase(k);
16     heapify(1);
17 }
18 void insert_or_update(const TPriority &p, const TKey &k){
19     int pos;
20     if(is_inserted(k)){
21         pos = position[k];
22         nodes[pos].first += p;
23     } else {
24         position[k] = pos = nodes.size();
25         nodes.push_back( make_pair(p, k) );
26     }
27     heapify(pos);
28 }
29 bool is_inserted(const TKey &k) {
30     return position.count(k);
31 }
32 int get_size() {
33     return (int)nodes.size() - 1;
34 }
35 void erase(const TKey &k){
36     if(!is_inserted(k)) return;
37     int pos = position[k];
38     swap_nodes(pos, nodes.size() - 1);
39     nodes.pop_back();
40     position.erase(k);
41     heapify(pos);
42 }
43 private:
44     vector<pair<TPriority, TKey>> nodes;
45     map<TKey, int> position;
46     void heapify(int pos){
47         if(pos >= nodes.size()) return;
48         while(1 < pos && nodes[pos / 2] <= nodes[pos]){
49             swap_nodes(pos / 2, pos);
50             pos /= 2;
51         }
52         int l = pos * 2, r = pos * 2 + 1, maxi = pos;
53         if(l < nodes.size() && nodes[l] > nodes[maxi]) maxi = l;
54         if(r < nodes.size() && nodes[r] > nodes[maxi]) maxi = r;
55         if(maxi != pos){
56             swap_nodes(pos, maxi);
57             heapify(maxi);
58         }
59     }
60     void swap_nodes(int a, int b){
61         position[ nodes[a].second ] = b;
62         position[ nodes[b].second ] = a;
63         swap(nodes[a], nodes[b]);
64     }
65 };

```

## 2. Ordenamiento

### 2.1. Merge sort

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

```
1 void merge_sort(int arr[], int ini, int fin){
2     if(ini == fin) return;
3     int mitad = (ini + fin) / 2;
4     merge_sort(arr, ini, mitad);
5     merge_sort(arr, mitad + 1, fin);
6
7     int tam1 = mitad - ini + 1, tam2 = fin - mitad;
8     int mitad1[tam1], mitad2[tam2];
9     for(int i = ini, idx = 0; i <= mitad; ++i, idx++){
10         mitad1[idx] = arr[i];
11     }
12     for(int i = mitad + 1, idx = 0; i <= fin; ++i, idx++){
13         mitad2[idx] = arr[i];
14     }
15     for(int i = ini, idx1 = 0, idx2 = 0; i <= fin; ++i){
16         if(idx1 < tam1 && idx2 < tam2){ /// si quedan elementos en ambas mitades
17             arr[i] = mitad1[idx1] < mitad2[idx2] ? mitad1[idx1++] : mitad2[idx2++];
18         } else { /// si solo hay elementos en mitad1
19             arr[i] = idx1 < tam1 ? mitad1[idx1++] : mitad2[idx2++];
20         }
21     }
22 }
```

## 3. Matemáticas

### 3.1. Criba de Eratóstenes

#### 3.1.1. Criba

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

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

#### 3.1.2. Criba sobre un rango

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

```
1 void criba_sobre_rango(long long a, long long b, vector<long long> &primos){
2     a = max(a, 0ll);
3     b = max(b, 0ll);
```

```

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

```

### 3.1.3. Criba segmentada

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

```

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

```

### 3.1.4. Criba lineal

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

```

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

```

```

15     }
16 }

```

### 3.2. Algoritmo extendido de Euclides

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

```

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

```

### 3.3. Solución de ecuaciones diofánticas lineales

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

```

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

```

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

```

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

```

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

```

1  int cuenta_soluciones(int a, int b, int c, int minx, int maxx, int miny, int maxy) {
2      int x, y, g;
3      if(!encuentra_solucion(a, b, c, x, y, g)) return 0;
4      /// ax + by = c ssi (a/g)x + (b/g)y = c/g
5      /// Dividimos entre g para simplificar y no dividir a cada rato
6      a /= g;
7      b /= g;
8      /// Signos de a, b nos sirven para pasar a la
9      /// siguiente (anterior) solucion
10     int sign_a = a > 0 ? +1 : -1;
11     int sign_b = b > 0 ? +1 : -1;
12     /// pasa a la minima solucion tal que minx <= x
13     cambia_solucion(x, y, a, b, (minx - x) / b);
14     /// si x < minx, pasa a la siguiente para que minx <= x
15     if(x < minx) cambia_solucion(x, y, a, b, sign_b);
16     if(x > maxx) return 0; /// si x > maxx, entonces no hay x solucion tal que x in [minx, maxx]
17     int lx1 = x;

```



```

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

```

### 3.4. Funciones multiplicativas

#### 3.4.1. Función Phi de Euler

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

```

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

```

#### 3.4.2. Función sigma

**Sigma 0** ( $\sigma_0$ ). Complejidad: Tiempo  $O(d)$  - Memoria extra  $O(n)$ .  $d$  es la cantidad de factores primos de  $n$ .  $lp[i]$  es el menor primo que divide a  $i$ . Cuenta la cantidad de divisores de  $n$ .

```

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

```

**Sigma 1** ( $\sigma_1$ ). Complejidad: Tiempo  $O(d)$  - Memoria extra  $O(n)$ .  $d$  es la cantidad de factores primos de  $n$ .  $lp[i]$  es el menor primo que divide a  $i$ . Calcula la suma de los divisores de  $n$ .

```

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

```

### 3.4.3. Función de Moebius

Complejidad: Tiempo  $O(d)$  - Memoria extra  $O(n)$ .  $d$  es la cantidad de factores primos de  $n$ .  $lp[i]$  es el menor primo que divide a  $i$ . Devuelve 0 si  $n$  no es divisible por algún cuadrado. Devuelve 1 o  $-1$  si  $n$  es divisible por al menos un cuadrado. Devuelve 1 si  $n$  tiene una cantidad par de factores primos. Devuelve  $-1$  si  $n$  tiene una cantidad impar de factores primos.

```

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

```

## 3.5. Exponenciación binaria

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

```

1  int bin_exp(int a, int b){
2      int ans = 1;
3      while(b){
4          if(b % 2) ans *= a;
5          a *= a;
6          b /= 2;
7      }
8      return ans;
9  }

```

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

```

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

```

## 3.6. Transformada de Fourier

### 3.6.1. FFT

Complejidad: Tiempo  $O(n \log(n))$  - Memoria extra  $O(n \log(n))$ , donde  $n$  es el grado del polinomio  $P$ .

```
1 using comp = complex<double>;
2 const double PI = acos(-1);
3 vector<comp> FFT(vector<comp> &P, bool inversa){
4     int n = P.size();
5     if(n == 1) return P;
6     vector<comp> Pe, Po;
7     for(int i = 0; i < n; ++i)
8         if(i % 2) Po.push_back(P[i]);
9         else Pe.push_back(P[i]);
10    vector<comp> eval_Pe = FFT(Pe, inversa);
11    vector<comp> eval_Po = FFT(Po, inversa);
12    vector<comp> eval(n);
13    double angulo = 2 * PI / n * (inversa ? -1 : 1);
14    comp w(1), w_n(cos(angulo), sin(angulo));
15    for(int i = 0; i < n / 2; ++i){
16        eval[i] = eval_Pe[i] + w * eval_Po[i];
17        eval[i + n / 2] = eval_Pe[i] - w * eval_Po[i];
18        if(inversa){
19            eval[i] /= 2;
20            eval[i + n / 2] /= 2;
21        }
22        w *= w_n;
23    }
24    return eval;
25 }
```

### 3.6.2. Multiplicar polinomios

Complejidad: Tiempo  $O(n \log(n))$  - Memoria extra  $O(n \log(n))$ , donde  $n$  es el grado máximo polinomio  $A$  y  $B$ .

```
1 vector<int> multiplicar(vector<int> A, vector<int> B){
2     vector<comp> cA(A.begin(), A.end()), cB(B.begin(), B.end());
3     int n = 1;
4     while(n < A.size() + B.size()) n *= 2;
5     cA.resize(n);
6     cB.resize(n);
7     vector<comp> val_A = FFT(cA, false);
8     vector<comp> val_B = FFT(cB, false);
9     for(int i = 0; i < n; ++i) val_A[i] *= val_B[i];
10    val_A = FFT(val_A, true);
11    vector<int> res(n);
12    for(int i = 0; i < n; ++i) res[i] = round(val_A[i].real());
13    int carry = 0;
14    for(int i = 0; i < n; i++){
15        res[i] += carry;
16        carry = res[i] / 10;
17        res[i] %= 10;
18    }
19    return res;
20 }
```

## 4. Sparse table

Complejidad: Tiempo de precalculo  $O(n \log n)$  - Tiempo en responder  $O(\log(r - l + 1))$  - Tiempo en responder para operaciones idempotentes  $O(1)$  - Memoria extra  $O(n \log n)$ . LOGN es  $\lceil \log_2(\text{MAXN}) \rceil$ . Indexado en 0.

```

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

```

## 5. Fenwick Tree

Complejidad: Tiempo en responder  $O(\log n)$  - Tiempo de actualización  $O(\log n)$  - Memoria extra  $O(n)$ . Indexado en 1.

```

1 struct fenwick_tree{
2     int n;
3     vector<int> BIT;
4     fenwick_tree(int _n){
5         n = _n;
6         BIT.resize(n + 1);
7     }
8     void add(int pos, int x){
9         while(pos <= n){
10            BIT[pos] += x;
11            pos += lsb(pos);
12        }
13    }
14    int sum(int pos){
15        int res = 0;
16        while(pos){
17            res += BIT[pos];
18            pos -= lsb(pos);

```

```

19     }
20     return res;
21 }
22 };

```

## 6. Segment Tree

Nodo del Segment Tree:

```

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

```

### 6.1. Actualizaciones puntuales

Complejidad: Tiempo de precalculo  $O(n)$  - Tiempo en responder  $O(\log n)$  - Tiempo de actualización  $O(\log n)$  - Memoria extra  $O(n)$ . Indexado en 1.

```

1 struct segment_tree{
2     struct node{...};
3     vector<node> nodes;
4     segment_tree(int n, int data[]){
5         nodes.resize(4 * n + 1);
6         build(1, n, data);
7     }
8     void build(int left, int right, int data[], int pos = 1){
9         if(left == right){
10             nodes[pos].val = data[left];
11             return;
12         }
13         int mid = (left + right) / 2;
14         build(left, mid, data, pos * 2);
15         build(mid + 1, right, data, pos * 2 + 1);
16         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
17     }
18     void update(int x, int idx, int left, int right, int pos = 1){
19         if(idx < left || right < idx) return;
20         if(left == right){
21             nodes[pos].val += x;
22             return;
23         }
24         int mid = (left + right) / 2;
25         update(x, idx, left, mid, pos * 2);
26         update(x, idx, mid + 1, right, pos * 2 + 1);
27         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
28     }
29     node query(int l, int r, int left, int right, int pos = 1){
30         if(r < left || right < l) return node(); /// Devuelve el neutro
31         if(l <= left && right <= r) return nodes[pos];
32         int mid = (left + right) / 2;
33         return query(l, r, left, mid, pos * 2) + query(l, r, mid + 1, right, pos * 2 + 1);
34     }
35 };

```

## 6.2. Actualizaciones sobre rangos

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

```
1 struct segment_tree{
2     struct node{...};
3     vector<node> nodes;
4     segment_tree(int n, int data[]){...}
5     void build(int left, int right, int data[], int pos = 1){...}
6     void combine_lazy(int lz, int pos){nodes[pos].lazy += lz;}
7     void apply_lazy(int pos, int tam){
8         nodes[pos].val += nodes[pos].lazy * tam;
9         nodes[pos].lazy = 0;
10    }
11    void push_lazy(int pos, int left, int right){
12        int tam = abs(right - left + 1);
13        if(1 < tam){
14            combine_lazy(nodes[pos].lazy, pos * 2);
15            combine_lazy(nodes[pos].lazy, pos * 2 + 1);
16        }
17        apply_lazy(pos, tam);
18    }
19    void update(int x, int l, int r, int left, int right, int pos = 1){
20        push_lazy(pos, left, right);
21        if(r < left || right < l) return;
22        if(l <= left && right <= r){
23            combine_lazy(x, pos);
24            push_lazy(pos, left, right);
25            return;
26        }
27        int mid = (left + right) / 2;
28        update(x, l, r, left, mid, pos * 2);
29        update(x, l, r, mid + 1, right, pos * 2 + 1);
30        nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
31    }
32    node query(int l, int r, int left, int right, int pos = 1){
33        push_lazy(pos, left, right);
34        ...
35    }
36};
```

## 7. Sqrt decomposition

### 7.1. Algoritmo de MO

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

```
1 const int block_size = 300; /// Ajustable
2 struct query {
3     int l, r, block, i;
4     bool operator<(const query &b) const {
5         if(block == b.block) return r < b.r;
6         return block < b.block;
7     }
8 };
9 void add(int idx){/**TO-DO*/}
10 void remove(int idx){/**TO-DO*/}
11 int get_answer(){return 0; /**TO-DO*/}
```

```

12 vector<int> solve(vector<query> &queries) {
13     vector<int> answers(queries.size());
14     sort(queries.begin(), queries.end());
15     int l_act = 0;
16     int r_act = -1;
17     for(query q : queries){
18         while(l_act > q.l) add(--l_act);
19         while(r_act < q.r) add(++r_act);
20         while(l_act < q.l) remove(l_act++);
21         while(r_act > q.r) remove(r_act--);
22         answers[q.i] = get_answer();
23     }
24     return answers;
25 }

```

## 8. DSU

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

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

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

```

1 struct dsu{
2     struct action{
3         int x_p, y_p, rank_y;
4     };
5     vector<int> RA, P;
6     vector<action> actions;
7     dsu(int n){
8         RA.resize(n, 1);
9         P.resize(n);
10        iota(P.begin(), P.end(), 0);
11    }
12    int root(int x){
13        return x == P[x] ? x : P[x] = root(P[x]);
14    }
15    bool join(int x, int y, bool recording){
16        x = root(x);
17        y = root(y);
18        if(x == y) return false;
19        if(RA[x] >= RA[y]) swap(x, y);
20        if(recording) actions.push_back({x, y, RA[y]});
21        RA[y] += RA[x];
22        P[x] = y;
23        return true;
24    }
25    void rollback(int cnt){
26        while(cnt-- > 0 && actions.size()){
27            action act = actions.back();
28            actions.pop_back();
29            RA[act.y_p] = act.rank_y;
30            P[act.x_p] = act.x_p;
31        }
32    }
33 };

```

## 9. Grafos

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

### 9.1. Caminos mínimos

#### 9.1.1. Dijkstra

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

```
1 int64_t dijkstra(int a, int b, vector<edge> graph[]){
2     int64_t dist[MAXN];
3     bool vis[MAXN] = {};
4     fill(dist, dist + MAXN, LLONG_MAX);
5     priority_queue<pos> q;
6     q.push(pos{a, 0});
7     dist[a] = 0;
8     while(!q.empty()){
9         pos act = q.top();
10        q.pop();
11        if(vis[act.from]) continue;
12        vis[act.from] = true;
13        for(edge &e : graph[act.from]){
14            if(dist[e.to] <= dist[act.from] + e.w) continue;
15            dist[e.to] = dist[act.from] + e.w;
16            q.push(pos{e.to, dist[e.to]});
17        }
18    }
19    return dist[b];
20 }
```

#### 9.1.2. Bellman-Ford

Complejidad:  $O(|V||E|)$ .

```
1 vector<int> bellman_ford(int s, int n, vector<edge> &edges, bool cycles = false){
2     vector<int> d(n, (cycles ? 0 : INT_MAX));
3     d[s] = 0;
4     vector<int> P(n, -1); /// Predecesor
5     for(int i = 0; i < n - 1; ++i){
6         for(edge &e : edges){
7             if(d[e.from] == INT_MAX) continue;
8             if(d[e.to] > d[e.from] + e.w){
9                 d[e.to] = d[e.from] + e.w;
10                P[e.to] = e.from;
11            }
12        }
13    }
```



```

13     }
14     int last_relax = -1;
15     for(edge &e : edges){
16         if(d[e.from] == INT_MAX) continue;
17         if(d[e.to] > d[e.from] + e.w){
18             d[e.to] = d[e.from] + e.w;
19             P[e.to] = e.from;
20             last_relax = e.to;
21         }
22     }
23     if(last_relax == -1) return d;
24     return {}; /// VACIO
25 }

```

### 9.1.3. Floyd-Warshall

Complejidad:  $O(|V|^3)$ .

```

1 vector<vector<int>> floyd_warshall(int n){
2     vector<vector<int>> d(n, vector<int>(n, INT_MAX));
3     /// aqui inicializa con la lista/matriz de adyacencia
4     /// luego calcula la dp
5     for(int k = 0; k < n; ++k){
6         for(int i = 0; i < n; ++i){
7             for(int j = 0; j < n; ++j){
8                 if(d[i][k] == INT_MAX) continue;
9                 if(d[k][j] == INT_MAX) continue;
10                if(d[i][j] > d[i][k] + d[k][j]) d[i][j] = d[i][k] + d[k][j];
11            }
12        }
13    }
14    return d;
15 }

```

### 9.1.4. Johnson's algorithm

Complejidad:  $O(|V||E|\log|V|)$ . Sea  $p : V \rightarrow \mathbb{R}$  una función potencial del grafo. El algoritmo es como sigue:

1. Hacemos una transformación en el grafo cambiando los pesos  $w$  a  $w'(u, v) = w(u, v) + p(u) - p(v)$ .
2. Calculamos la distancia mínima  $d' : V \times V \rightarrow \mathbb{R}$  desde cada nodo a todos los demás con Dijkstra.
3. Finalmente, la distancia mínima de  $u$  a  $v$  en el grafo original es  $d(u, v) = d'(u, v) - p(u) + p(v)$ .

La función potencial  $p$  puede ser cualquiera. Usando Bellman-Ford se puede calcular el potencial  $p(u)$  como el camino más corto que termina (o empieza) en  $u$ .

## 9.2. Árboles

### 9.2.1. MST

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

```

1 int64_t prim(vector<edge> graph[]){
2     int64_t e_cost[MAXN];
3     bool vis[MAXN] = {};
4     fill(e_cost, e_cost + MAXN, LLONG_MAX);
5     int64_t ans = 0;
6     priority_queue<edge> q;
7     q.push(edge{1, 1, 0});
8     while(q.size()){

```

```

9         int node = q.top().to;
10        int64_t w = q.top().w;
11        q.pop();
12        if(vis[node]) continue;
13        vis[node] = true;
14        ans += w;
15        for(edge &e : graph[node]){
16            if(vis[e.to] || e_cost[e.to] <= e.w) continue;
17            e_cost[e.to] = e.w;
18            q.push(e);
19        }
20    }
21    return ans;
22 }

```

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

```

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

```

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

```

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

```

### 9.2.2. LCA

Complejidad: Tiempo de preproceso  $O(|V| \log |V|)$ . Tiempo de LCA y  $n$ -ésimo ancestro  $O(\log |V|)$ . Indexado en 1.

```

1 void precalc(int node, int p = 0, int d = 1){
2     depth[node] = d;

```

```

3     P[0][node] = p;
4     for(int k = 1; k <= LOGN; ++k)
5         P[k][node] = P[k - 1][P[k - 1][node]];
6     for(int child : tree[node])
7         if(p != child) precalc(child, node, d + 1);
8 }
9 int LCA(int a, int b){
10     if(depth[b] < depth[a]) swap(a, b);
11     int dif = depth[b] - depth[a];
12     for(int k = LOGN; 0 <= k; --k)
13         if(is_on(dif, k)) b = P[k][b];
14     if(a == b) return a;
15     for(int k = LOGN; 0 <= k; --k){
16         if(P[k][a] != P[k][b]){
17             a = P[k][a];
18             b = P[k][b];
19         }
20     }
21     return P[0][a];
22 }
23 int nth_ancestor(int u, int n){
24     for(int k = LOGN; 0 <= k; --k)
25         if(is_on(n, k)) u = P[k][u];
26     return u;
27 }

```

### 9.2.3. Sack

Complejidad: Tiempo  $O(|V| \log |V|)$ . Indexado en 1.

```

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

```

```

30 |     if(!keep) add(node, -1, p);
31 | }

```

## 9.3. Máximo flujo

### 9.3.1. Algunos problemas de flujos

- *Maximum Weight Closure.* Sea  $N_1$  una clausura de  $G$  y  $N_2 = V \setminus N_1$ , tenemos que  $w(N_1) = \sum_{i \in N_1^+} w_i - \sum_{i \in N_1^-} |w_i|$  y  $Cap.Corte = \sum_{i \in N_2^+} w_i + \sum_{i \in N_1^-} |w_i|$ . Entonces

$$Cap.Corte + w(N_1) = \sum_{i \in N_1^+} w_i + \sum_{i \in N_2^+} w_i.$$

- *Mínima cobertura de vértices.* En grafos generales es NP-Completo. En grafos bipartitos el máximo emparejamiento es igual al número de vértices en la mínima cobertura. Para el problema con pesos en los nodos, unimos  $s$  a todos los nodos en  $L$  con capacidad igual al peso de cada nodo, unimos los nodos de  $R$  a  $t$  de la misma manera y unimos los nodos de  $L$  a  $R$  con capacidad infinita. El máximo flujo es el peso mínimo de la mínima cobertura.
- *Máximo conjunto independiente.* Cualquier conjunto independiente es el complemento de alguna cobertura de vértices.
- *Mínimo cubrimiento de caminos independientes.* En grafos generales es NP-hard. En DAG's duplicamos los nodos en un lado IN y un lado OUT. Conectamos  $s$  al lado OUT y el lado IN a  $t$ . Las aristas del DAG las agregamos del lado OUT al lado IN. Sea  $M$  el máximo emparejamiento de la red anterior, entonces el mínimo cubrimiento es  $|V| - M$ .
- *Mínimo cubrimiento de caminos NO necesariamente independientes.* En grafos generales es NP-hard. En DAG's transformamos el DAG a su clausura transitiva y aplicamos el problema anterior.
- *Teorema de Mirsky.* En todo POSET, el tamaño de la cadena de mayor tamaño es igual al número de anticadenas necesarias para cubrir todos los elementos del conjunto.
- *Teorema de Dilworth.* En todo POSET, el tamaño de la anticadena de mayor tamaño es igual al número de cadenas necesarias para cubrir todos los elementos del conjunto.
- *Teorema de Hall.* Un grafo bipartito con subconjuntos  $L$  y  $R$  tiene un emparejamiento de tamaño  $|L|$  si y sólo si para todo subconjunto  $W$  de  $L$ , se cumple que  $|W| \leq |N_G(W)|$ , donde  $N_G(W)$  es el conjunto de vértices vecinos de alguno de los vértices en  $W$ .

### 9.3.2. Edmonds-Karp

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

Ejemplo de uso

```

1 | int main(){
2 |     int n,m,a,b; cin >> n >> m;
3 |     vector< vector<edge> > elGrafo(n);
4 |     int64_t c;
5 |     while( m-- ){
6 |         cin >> a >> b >> c;
7 |         elGrafo[a-1].push_back( {a-1,b-1,c,c,0} );
8 |     }
9 |
10 |    //Flujo maximo
11 |    ford_fulkerson ff(elGrafo);
12 |    cout << ff.get_max_flow( 0, n-1 ) << '\n';
13 |
14 |
15 |
16 |    //corte minimo

```

```

17     vector< vector<edge> > residual = ff.get_residual_graph();
18     vector<bool> visitados(n,false);
19     queue<int> q;
20     q.push(0);
21     visitados[0]=true;
22
23     while( !q.empty() ){
24         int u = q.front(); q.pop();
25         for( edge& e : residual[u] ){
26             if( e.c>0 && !visitados[e.to] ){
27                 visitados[e.to] = true;
28                 q.push(e.to);
29             }
30         }
31     }
32
33     for( int u=0; u<n; u++ ){
34         for( edge& e : grafo[u] ){
35             if( visitados[u] && !visitados[e.to] && e.w > 0){
36                 cout << u+1 << " " << e.to+1 << endl;
37             }
38         }
39     }
40 }

1 struct edge {
2     int from, to;
3     int64_t w; /// weight
4     int64_t c; /// capacity
5     int64_t f; /// flow
6 };

1 class ford_fulkerson {
2 public:
3     ford_fulkerson (vector<vector<edge>> &graph) : graph(graph){}
4     int64_t get_max_flow(int s, int t){
5         init();
6         int64_t f = 0;
7         while(find_and_update(s, t, f)){
8             return f;
9         }
10    vector<int> get_st_cut(const int &s){
11        bool vis[graph.size()] = {};
12        vector<int> S;
13        queue<int> q;
14        q.push(s);
15        S.push_back(s);
16        vis[s] = true;
17        while(q.size()){
18            int u = q.front(); q.pop();
19            for(int eI : edge_indexes[u]){
20                if(edges[eI].c > edges[eI].f && !vis[edges[eI].to]){
21                    q.push(edges[eI].to);
22                    vis[edges[eI].to] = true;
23                    S.push_back(edges[eI].to);
24                }
25            }
26        }
27        return S;
28    }

```

```

29 // Function to get the residual graph after computing the max flow
30 vector<vector<edge>> get_residual_graph() const {
31     vector<vector<edge>> residual(graph.size());
32     for (size_t i = 0; i < edges.size(); i += 2) {
33         const edge& e = edges[i];
34         if (e.c > 0) { // Only consider edges with positive capacity
35             residual[e.from].push_back({e.from, e.to, e.w, e.c - e.f, e.f});
36             residual[e.to].push_back({e.to, e.from, -e.w, e.f, -e.f});
37         }
38     }
39     return residual;
40 }
41 private:
42     vector<vector<edge>> graph; /// graph (to, capacity)
43     vector<edge> edges; /// List of edges (including the inverse ones)
44     vector<vector<int>> edge_indexes; /// indexes of edges going out from each vertex
45     void init(){
46         edges.clear();
47         edge_indexes.clear(); edge_indexes.resize(graph.size());
48         for(int u = 0; u < graph.size(); u++){
49             for(edge &e : graph[u]){
50                 edges.push_back({u, e.to, e.w, e.c, 0});
51                 edges.push_back({e.to, u, -e.w, 0, 0});
52                 edge_indexes[u].push_back(edges.size() - 2);
53                 edge_indexes[e.to].push_back(edges.size() - 1);
54             }
55         }
56     }
57     bool find_and_update(int s, int t, int64_t &flow){
58         // Encontrar camino desat con BFS
59         queue<int> q;
60         // Desde donde llego y con que arista
61         vector<pair<int, int>> from(graph.size(), make_pair(-1, -1));
62         q.push(s);
63         from[s] = make_pair(s, -1);
64         bool found = false;
65         while(q.size() && (!found)){
66             int u = q.front(); q.pop();
67             for(int eI : edge_indexes[u]){
68                 if((edges[eI].c > edges[eI].f) && (from[edges[eI].to].first == -1)){
69                     from[edges[eI].to] = make_pair(u, eI);
70                     q.push(edges[eI].to);
71                     if(edges[eI].to == t) found = true;
72                 }
73             }
74         }
75         if(!found) return false;
76         // Encontrar cap. minima del camino de aumento
77         int64_t u_flow = LLONG_MAX;
78         int current = t;
79         while(current != s) {
80             u_flow = min(u_flow, edges[from[current].second].c - edges[from[current].second].f);
81             current = from[current].first;
82         }
83         current = t;
84         // Actualizar flujo
85         while(current != s){
86             edges[from[current].second].f += u_flow;
87             edges[from[current].second^1].f -= u_flow; // Arista inversa

```

```

88         current = from[current].first;
89     }
90     flow += u_flow ;
91     return true;
92 }
93 };

```

### 9.3.3. Dinic

Complejidad:  $O(|V|^2|E|)$ .

```

1  const int MAXV = 32767; /// 2^15 - 1
2  template<class T = int64_t> struct dinic{
3      dinic(short V){this->V = V; if(V > MAXV){cout << "ERROR"; exit(0);}}
4      const static bool SCALING = true;
5
6      bool sorted = false;
7      short s, t, V;
8      int lim = 1; /// Para escalado
9      const T INF = numeric_limits<T>::max();
10     short level[MAXV]; /// distancia desde s
11     short ptr[MAXV]; /// arista por la que va explorando
12
13     struct edge{
14         short to, rev;
15         T cap, flow, mcap;
16         bool operator<(const edge &b)const{return mcap > b.mcap;}
17     };
18     vector<edge> adj[MAXV];
19     vector<short> adj_current[MAXV]; /// aristas del grafo de nivel
20     void add_edge(short u, short v, T cap, bool is_directed = true){
21         if(u == v) return;
22         T add = (is_directed ? 0 : cap);
23         adj[u].push_back({v, adj[v].size(), cap, 0, cap + add});
24         adj[v].push_back({u, (short)adj[u].size() - 1, add, 0, cap + add});
25     }
26     void mysort(){
27         if(sorted) return;
28         sorted = true;
29         for(int i = 0; i < V; ++i){
30             sort(adj[i].begin(), adj[i].end());
31             for(int j = 0; j < adj[i].size(); ++j){
32                 adj[adj[i][j].to][adj[i][j].rev].rev = j;
33             }
34         }
35     }
36     bool bfs(){ /// Crea grafo de nivel
37         for(int i = 0; i < V; ++i){
38             adj_current[i].clear();
39             adj_current[i].reserve(adj[i].size());
40         }
41         queue<short> q;
42         q.push(s);
43         fill(level, level + V, -1);
44         level[s] = 0;
45         while(q.size()){
46             short u = q.front(); q.pop();
47             if(u == t) return true;
48             for(int i = 0; i < (int)adj[u].size(); ++i){

```

```

49         edge &e = adj[u][i];
50
51         if(e.mcap < lim) break;
52         if(level[e.to] == -1 && e.cap - e.flow >= lim){
53             level[e.to] = level[u] + 1;
54             adj_current[u].push_back(i);
55             q.push(e.to);
56         } else if(level[e.to] == level[u] + 1 && e.cap - e.flow >= lim){
57             adj_current[u].push_back(i);
58         }
59     }
60 }
61
62     return false;
63 }
64 T dfs(short u, T flow, vector<int> &S, bool save = false){ /// Encuentra camino, bloquea aristas
65     if(save) S.push_back(u);
66     if(u == t) return flow;
67     for(; ptr[u] < adj_current[u].size(); ++ptr[u]){
68         edge &e = adj[u][adj_current[u][ptr[u]]];
69         if(T pushed = dfs(e.to, min(flow, e.cap - e.flow), S, save)){
70             e.flow += pushed;
71             adj[e.to][e.rev].flow -= pushed;
72             if(e.cap - e.flow < lim) ptr[u]++;
73             return pushed;
74         }
75     }
76     return 0;
77 }
78 int64_t get_max_flow(short source, short sink){
79     s = source;
80     t = sink;
81     mysort();
82     vector<int> S;
83     int64_t flow = 0;
84     for(lim = SCALING ? (1 << 30) : 1; 0 < lim; lim >>= 1){
85         while(bfs()){
86             memset(ptr, 0, sizeof(ptr));
87             while(T pushed = dfs(s, INF, S)) flow += pushed; /// Bloquear flujo
88         }
89     }
90     return flow;
91 }
92 vector<int> get_st_cut(){
93     vector<int> S;
94     memset(ptr, 0, sizeof(ptr));
95     dfs(s, INF, S, true);
96     return S;
97 }
98 };

```

## 9.4. SCC

### 9.4.1. Kosajaru

Complejidad: Tiempo  $O(n)$ .

```

1 void dfs(int node, vector<int> &topo_ord){
2     if(vis[node]) return;
3     vis[node] = true;

```



```

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

```

## 9.5. 2-Sat

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

```

1 struct two_sat{
2     int n;
3     vector<vector<int>> graph, inv_graph;
4     vector<int> scc, ans;
5     vector<bool> vis;
6     two_sat(){}
7     two_sat(int _n){
8         n = _n;
9         graph.resize(2 * n);
10        inv_graph.resize(2 * n);
11        scc.resize(2 * n);
12        vis.resize(2 * n);
13        ans.resize(n);
14    }
15    void add_edge(int u, int v){
16        graph[u].push_back(v);
17        inv_graph[v].push_back(u);
18    }
19    /// al menos una es verdadera
20    void add_or(int p, bool val_p, int q, bool val_q){
21        add_edge(p + (val_p ? n : 0), q + (val_q ? 0 : n));
22        add_edge(q + (val_q ? n : 0), p + (val_p ? 0 : n));
23    }

```

```

24  /// exactamente una es verdadera
25  void add_xor(int p, bool val_p, int q, bool val_q){
26      add_or(p, val_p, q, val_q);
27      add_or(p, !val_p, q, !val_q);
28  }
29  /// p y q tienen el mismo valor
30  void add_and(int p, bool val_p, int q, bool val_q){
31      add_xor(p, !val_p, q, val_q);
32  }
33  /// Kosajaru
34  void dfs(int node, vector<int> &topo_ord){...}
35  void assign_scc(int node, const int id){...}
36  /// construye respuesta
37  bool build_ans(){
38      fill(vis.begin(), vis.end(), false);
39      vector<int> topo_ord;
40      for(int i = 0; i < 2 * n; ++i) dfs(i, topo_ord);
41      fill(vis.begin(), vis.end(), false);
42      reverse(topo_ord.begin(), topo_ord.end());
43      int id = 0;
44      for(int u : topo_ord) if(!vis[u]) assign_scc(u, id++);
45      for(int i = 0; i < n; ++i){
46          if(scc[i] == scc[i + n]) return false;
47          ans[i] = (scc[i] < scc[i + n] ? 0 : 1);
48      }
49      return true;
50  }
51  };

```

## 10. Treap

Complejidad: Tiempo  $O(\log(n))$  - Memoria  $O(n)$ . Para treap implícito (arreglo dinámico) cambiar en insert/erase a `split_by_pos()`.

```

1  struct treap{
2      typedef struct _node{
3          long long x;
4          int freq, cnt;
5          long long p;
6          _node *l, *r;
7          _node(long long _x): x(_x), p(((long long)(rand()) << 32)^rand()),
8              cnt(1), freq(1), l(nullptr), r(nullptr){}
9          ~_node(){delete l; delete r;}
10         void recalc(){
11             cnt = freq;
12             cnt += ((l) ? (l->cnt) : 0);
13             cnt += ((r) ? (r->cnt) : 0);
14         }
15     }* node;
16     node root;
17     node merge(node l, node r){
18         if(!l || !r) return l ? l : r;
19         if(l->p < r->p){
20             r->l = merge(l, r->l);
21             r->recalc();
22             return r;
23         } else {
24             l->r = merge(l->r, r);
25             l->recalc();

```

```

26         return l;
27     }
28 }
29 void split_by_value(node n, long long d, node &l, node &r){
30     l = r = nullptr;
31     if(!n) return;
32     if(n->x < d){
33         split_by_value(n->r, d, n->r, r);
34         l = n;
35     } else {
36         split_by_value(n->l, d, l, n->l);
37         r = n;
38     }
39     n->recalc();
40 }
41 void split_by_pos(node n, int pos, node &l, Node &r, int l_nodes = 0){
42     l = r = NULL;
43     if(!n) return;
44     int cur_pos = (n->l) ? (l_nodes + n->l->cnt) : l_nodes;
45     if(cur_pos < pos){
46         splitFirstNodes(n->r, pos, n->r, r, cur_pos + 1);
47         l = n;
48     } else {
49         splitFirstNodes(n->l, pos, l, n->l, l_nodes);
50         r = n;
51     }
52     n->recalc();
53 }
54 treap(): root(NULL){}
55 void insert_value(long long x){
56     node l, m, r;
57     split_by_value(root, x, l, m);
58     split_by_value(m, x + 1, m, r);
59     if(m){
60         m->freq++;
61         m->cnt++;
62     } else m = new _node(x);
63     root = merge(merge(l, m), r);
64 }
65 void erase_value(long long x){
66     node l, m, r;
67     split_by_value(root, x, l, m);
68     split_by_value(m, x + 1, m, r);
69     if(!m || m->freq == 1){
70         delete m;
71         m = nullptr;
72     } else {
73         m->freq--;
74         m->cnt--;
75     }
76     root = merge(merge(l, m), r);
77 }
78 };

```

## 11. Strings

### 11.1. KMP

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

```

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

```

### 11.1.1. Autómata de KMP

Complejidad: Tiempo  $O(|s|k)$  - Memoria extra  $O(|s|k)$ , donde  $k$  es el tamaño del alfabeto.

```

1 void compute_automaton(const string &s, vector<vector<int>>& aut){
2     s += '#';
3     int n = s.size();
4     vector<int> pi = prefix_function(s);
5     aut.assign(n, vector<int>(26));
6     for (int i = 0; i < n; ++i) {
7         for (int c = 0; c < 26; ++c) {
8             if (i && 'a' + c != s[i]) aut[i][c] = aut[pi[i - 1]][c];
9             else aut[i][c] = i + ('a' + c == s[i]);
10        }
11    }
12 }

```

## 11.2. Suffix array

### 11.2.1. Construcción

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

```

1 int SA[MAXN], mrank[MAXN];
2 int tmpSA[MAXN], tmp_mrank[MAXN];
3 void counting_sort(int k, int n){
4     int freqs[MAXN] = {};
5     for(int i = 0; i < n; ++i){
6         if(i + k < n) freqs[ mrank[i + k] ]++;
7         else freqs[0]++;
8     }
9     int m = max(100, n);
10    for(int i = 0, sfs = 0; i < m; ++i){
11        int f = freqs[i];
12        freqs[i] = sfs;
13        sfs += f;
14    }
15    for(int i = 0; i < n; ++i){
16        if(SA[i] + k < n) tmpSA[ freqs[mrank[ SA[i] + k ] ]++ ] = SA[i];
17        else tmpSA[ freqs[0]++ ] = SA[i];
18    }
19    for(int i = 0; i < n; ++i) SA[i] = tmpSA[i];
20 }
21 void buildSA(string &str){
22     int n = str.size();
23     for(int i = 0; i < n; ++i){
24         mrank[i] = str[i] - '#';
25         SA[i] = i;
26    }

```

```

27     for(int k = 1; k < n; k <= 1){
28         counting_sort(k, n);
29         counting_sort(0, n);
30         int r = 0;
31         tmp_mrank[ SA[0] ] = 0;
32         for(int i = 1; i < n; ++i){
33             if(mrank[ SA[i] ] != mrnk[ SA[i - 1] ] || mrnk[ SA[i] + k ] != mrnk[ SA[i - 1] + k ])
34                 tmp_mrank[ SA[i] ] = ++r;
35             else tmp_mrank[ SA[i] ] = r;
36         }
37         for(int i = 0; i < n; ++i) mrnk[i] = tmp_mrank[i];
38     }
39 }
40 inline bool suff_compare1(int idx,const string &pattern) {
41     return (s.substr(idx).compare(0, pattern.size(), pattern) < 0);
42 }
43 inline bool suff_compare2(const string &pattern,int idx) {
44     return (s.substr(idx).compare(0, pattern.size(), pattern) > 0);
45 }
46 pair<int,int> match(const string &pattern) {
47     int *low = lower_bound (SA, SA + s.size(), pattern, suff_compare1);
48     int *up = upper_bound (SA, SA + s.size(), pattern, suff_compare2);
49     return make_pair((int)(low - SA),(int)(up - SA));
50 }

```

### 11.2.2. Prefijo común más largo

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

```

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

```

### 11.3. Aho-Corasick

Construcción en  $O(mk)$ , donde  $m$  es el tamaño total de los strings y  $k$  el tamaño del alfabeto.

```

1  struct aho_corasick{
2      const static int K = 26;
3      const char index = 'a';
4      struct vertex{
5          int next[K];
6          bool terminal = false;

```

```

7      int p = -1;
8      char p_edge;
9      int link = -1;
10     int terminal_link = -1;
11     int go[K];
12     vertex(int p = -1, char c = '$') : p(p), p_edge(c){
13         fill(next, next + K, -1);
14         fill(go, go + K, -1);
15     }
16 };
17 vector<vertex> aho;
18 aho_corasick(){ aho.resize(1); }
19 void add_string(const string &s){
20     int u = 0;
21     for(char c : s){
22         int e = c - index;
23         if(aho[u].next[e] == -1){
24             aho[u].next[e] = aho.size();
25             aho.emplace_back(u, c);
26         }
27         u = aho[u].next[e];
28     }
29     aho[u].terminal = true;
30 }
31 int get_link(int u){
32     if(aho[u].link == -1){
33         if(!u || !aho[u].p) aho[u].link = 0;
34         else aho[u].link = go(get_link(aho[u].p), aho[u].p_edge);
35     }
36     return aho[u].link;
37 }
38 int go(int u, char c){
39     int e = c - index;
40     if(aho[u].go[e] == -1){
41         if(aho[u].next[e] != -1) aho[u].go[e] = aho[u].next[e];
42         else aho[u].go[e] = !u ? 0 : go(get_link(u), c);
43     }
44     return aho[u].go[e];
45 }
46 int get_terminal_link(int u){
47     if(aho[u].terminal_link == -1){
48         if(!u || !aho[u].p) aho[u].terminal_link = 0;
49         else aho[u].terminal_link = get_terminal_link(get_link(u));
50     }
51     return aho[u].terminal_link;
52 }
53 };

```

## 11.4. Suffix tree

COPIADO Y PEGADO POR

```

1  /// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ por cuestiones de tiempo
2  const int inf = 1e9;
3  const int maxn = 1e6;
4  int s[maxn];
5  map<int, int> to[maxn];
6  //Root is the vertex 0
7  //f_pos[i] is the initial index with the letter of the edge that goes from the parent of i to i
8  //len[i] is the number of letters in the edge that enters in i

```

```

9 //slink[i] is the suffix link
10 int len[maxn], f_pos[maxn], slink[maxn];
11 int node, pos;
12 int sz = 1, n = 0;
13
14 int make_node(int _pos, int _len){
15     f_pos[sz] = _pos;
16     len [sz] = _len;
17     return sz++;
18 }
19
20 void go_edge(){
21     while(pos > len[to[node][s[n - pos]]){
22         node = to[node][s[n - pos]];
23         pos -= len[node];
24     }
25 }
26
27 void add_letter(int c){
28     s[n++] = c;
29     pos++;
30     int last = 0;
31     while(pos > 0){
32         go_edge();
33         int edge = s[n - pos];
34         int &v = to[node][edge];
35         int t = s[f_pos[v] + pos - 1];
36         if(v == 0){
37             v = make_node(n - pos, inf);
38             //v = make_node(n - pos, 1);
39             slink[last] = node;
40             last = 0;
41         } else if(t == c) {
42             slink[last] = node;
43             return;
44         } else {
45             int u = make_node(f_pos[v], pos - 1);
46             to[u][c] = make_node(n - 1, inf);
47             to[u][t] = v;
48             f_pos[v] += pos - 1;
49             len [v] -= pos - 1;
50             v = u;
51             slink[last] = u;
52             last = u;
53         }
54         if(node == 0) pos--;
55         else node = slink[node];
56     }
57 }
58
59 void correct(int s_size){
60     len[0] = 0;
61     for (int i = 1; i < sz; i++){
62         if (f_pos[i] + len[i] - 1 >= s_size){
63             len[i] = (s_size - f_pos[i]);
64         }
65     }
66 }
67

```

```

68 void print_suffix_tree(int from){
69     cout << "Edge entering in " << from << " has size " << len[from];
70     cout << " and starts in " << f_pos[from] << endl;
71     cout << "Node " << from << " goes to: ";
72     for (auto u : to[from]){
73         cout << u.second << " with " << (char)u.first << " ";
74     }
75     cout << endl;
76     for (auto u : to[from]){
77         print_suffix_tree(u.second);
78     }
79 }
80
81 void build(string &s){
82     for (int i = 0; i < sz; i++){
83         to[i].clear();
84     }
85     sz = 1;
86     node = pos = n = 0;
87     len[0] = inf;
88     for(int i = 0; i < s.size(); i++)
89         add_letter(s[i]);
90     correct(s.size());
91 }
92
93 void cutGeneralized(vector<int> &finishPoints){
94     for (int i = 0; i < sz; i++){
95         int init = f_pos[i];
96         int end = f_pos[i] + len[i] - 1;
97         int idx = lower_bound(finishPoints.begin(), finishPoints.end(), init) - finishPoints.begin();
98         if ((idx != finishPoints.size()) && (finishPoints[idx] <= end)){//Must be cut
99             len[i] = (finishPoints[idx] - f_pos[i] + 1);
100             to[i].clear();
101         }
102     }
103 }
104
105
106 void build_generalized(vector<string> &ss){
107     for (int i = 0; i < sz; i++){
108         to[i].clear();
109     }
110     sz = 1;
111     node = pos = n = 0;
112     len[0] = inf;
113     int sep = 256;
114     vector<int> finishPoints;
115     int next = 0;
116     for (int i = 0; i < ss.size(); i++){
117         for (int j = 0; j < ss[i].size(); j++){
118             add_letter(ss[i][j]);
119         }
120         next += ss[i].size();
121         finishPoints.push_back(next);
122         add_letter(sep++);
123         next++;
124     }
125     correct(next);
126     cutGeneralized(finishPoints);

```



## 12. Geometría

### 12.1. Convex hull

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

```

1  /// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ por cuestiones de tiempo
2  struct pt {
3      double x, y;
4  };
5  int orientation(pt a, pt b, pt c) {
6      double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
7      if (v < 0) return -1; // clockwise
8      if (v > 0) return +1; // counter-clockwise
9      return 0;
10 }
11 bool cw(pt a, pt b, pt c, bool include_collinear) {
12     int o = orientation(a, b, c);
13     return o < 0 || (include_collinear && o == 0);
14 }
15 bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }
16 void convex_hull(vector<pt>& a, bool include_collinear = false) {
17     pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
18         return make_pair(a.y, a.x) < make_pair(b.y, b.x);
19     });
20     sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
21         int o = orientation(p0, a, b);
22         if (o == 0)
23             return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
24                 < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
25         return o < 0;
26     });
27     if (include_collinear) {
28         int i = (int)a.size()-1;
29         while (i >= 0 && collinear(p0, a[i], a.back())) i--;
30         reverse(a.begin()+i+1, a.end());
31     }
32     vector<pt> st;
33     for (int i = 0; i < (int)a.size(); i++) {
34         while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
35             st.pop_back();
36         st.push_back(a[i]);
37     }
38     a = st;
39 }

```

## 13. Utilidades

### 13.1. Plantilla tree

```

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

```

## 13.2. Números aleatorios

mt19937\_64 genera números de 64 bits.

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

## 13.3. Algunas formulas

$\oplus$  es el xor.

- $a|b = a \oplus b + a \& b$
- $a \oplus (a \& b) = (a|b) \oplus b$
- $b \oplus (a \& b) = (a|b) \oplus a$
- $(a \& b) \oplus (a|b) = a \oplus b$
- $a + b = a|b + a \& b = a \oplus b + 2(a \& b)$
- $a - b = (a \oplus (a \& b)) - ((a|b) \oplus a) = ((a|b) \oplus b) - ((a|b) \oplus a)$
- $a - b = (a \oplus (a \& b)) - (b \oplus (a \& b)) = ((a|b) \oplus b) - (b \oplus (a \& b))$

## 14. Bitmask

```
1 #define is_on(S, j) (S & (1ll << (j)))
2 #define set_bit(S, j) (S |= (1ll << (j)))
3 #define clear_bit(S, j) (S &= ~(1ll << (j)))
4 #define toggle_bit(S, j) (S ^= (1ll << (j)))
5 #define lsb(S) ((S) & -(S))
6 #define clear_lsb(S) (S &= (S - 1))
7 #define set_all(S, n) (S = (1ll << (n)) - 1ll)
8 #define clear_trailing_ones(S) (S &= (S + 1))
9 #define set_last_bit_off(S) (S |= (S + 1))
10 #define is_power_of_two(S) (!(S) & ((S) - 1))
11 #define nearest_power_of_two(S) ((int)pow(2, (int)((log((double)(S)) / log(2)) + 0.5)) )
12 #define is_divisible_by_power_of_two(n, k) !((n) & ((1ll << (k)) - 1))
13 #define modulo(S, N) ((S) & ((N) - 1)) // S % N, N potencia de 2
```

### 14.1. Útiles

Hay algunas funciones de gcc que nos pueden ayudar para hacer más eficiente nuestro código y evitar algunos bucles:

```
1 // one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.
2 int __builtin_ffs (int x):
3 // number of leading 0-bits in x, starting at the most significant bit position. If x is 0 is undefined
4 int __builtin_clz (unsigned int x):
5 // number of trailing 0-bits in x, starting at the least significant bit position. If x is 0 undefined.
6 int __builtin_ctz (unsigned int x):
7 // number of 1-bits in x.
8 int __builtin_popcount (unsigned int x):
9 // the parity of x, i.e. the number of 1-bits in x modulo 2.
10 int __builtin_parity (unsigned int x):
```

## 14.2. Iterar

Dada una máscara  $m$ , iterar sobre todos sus subconjuntos

```
1 for(int x=m; x; ){
2     --x &= m;
3     //...
4 }
```

El código anterior itera las máscaras válidas desde la más grande hasta la más pequeña (ojo el código no itera sobre  $x = m$ ) La complejidad de iterar sobre todas las submáscaras de todos los números de 1 a  $2^n$  es  $O(3^n)$ .

## 14.3. Gospers' Hack

Sirve para generar todos las máscaras de  $n$  bits, que tengan exactamente  $k$  bits a 1 (y que sean menores o iguales que  $2^n$ ). Complejidad  $O\left(\binom{n}{k}\right)$ ?

```
1 void GospersHack(int k, int n) {
2     int set = (1 << k) - 1;
3     int limit = (1 << n);
4     while (set < limit){
5         DoStuff(set);
6         // Gosper's hack:
7         int c = set & -set;
8         int r = set + c;
9         set = ((r ^ set) >> 2) / c | r;
10    }
11 }
```

DoStuff() is meant to be replaced with a function that processes each different value that set takes.

```
1 int mask = (1 << k) - 1, r, c;
2 while(mask <= (1 << n) - (1 << (n-k) ) ){
3     //...
4     c = mask & -mask;
5     r = mask + c;
6     mask = r | ( (r ^ mask) >> 2 / c );
7 }
```

## 15. Máximo de funciones

### 15.1. Li-Chao Tree

Dado un conjunto  $A$  con  $M$  valores a evaluar, y  $N$  funciones (tales que cada una de ellas se intersecta con el resto a lo más una vez), te devuelve  $\max_{i \in [N]} (f_i(a))$  en  $\log(M)$  para cualquier  $a \in A$ .

```
1 struct Function {
2     long long m;
3     long long b;
4     long long eval(long long x){
5         if (m == LLONG_MIN) return LLONG_MIN;
6         return m*x+b;
7     }
8     Function(){ m = LLONG_MIN;}
9     Function(long long m_, long long b_): m(m_), b(b_){ }
10 };

1 struct LiChaoTree {
2     vector<long long> values;
3     long long maxV;
4     Function *functions;
```

```

5   LiChaoTree(vector<long long> &values_){
6       values = values_;
7       sort(values.begin(), values.end());
8       functions = new Function[values.size() * 4];
9       maxV = values.size();
10  }
11  //Range from l to r - 1
12  long long get(long long x){
13      return get(x, 1, 0, maxV);
14  }
15  long long get(long long x, int v, int l, int r){
16      int m = (l + r) / 2;
17      long long mv = values[m];
18      if (r - l == 1){
19          return functions[v].eval(x);
20      } else if (x < mv){
21          return max(functions[v].eval(x), get(x, 2 * v, l, m));
22      } else {
23          return max(functions[v].eval(x), get(x, 2 * v + 1, m, r));
24      }
25  }
26  void addFunction(Function f){
27      addFunction(f, 1, 0, maxV);
28  }
29  void addFunction(Function f, int v, int l, int r){
30      int m = (l + r) / 2;
31      long long mv = values[m];
32      long long lv = values[l];
33      bool lef = f.eval(lv) > functions[v].eval(lv);
34      bool mid = f.eval(mv) > functions[v].eval(mv);
35      if (mid){//Si el actual pierde en el medio
36          swap(functions[v], f);
37      }
38      if (r - l == 1){
39          return;
40      } else if (lef != mid){//El cruce esta en el lado izq.
41          addFunction(f, 2 * v, l, m);
42      } else {
43          addFunction(f, 2 * v + 1, m, r);
44      }
45  }
46  ~LiChaoTree(){ delete[] functions; }
47  };

```