

Referencia - ICPC

Mathgic

Octubre 2023

Índice

1. Búsqueda completa	4
1.1. Permutaciones	4
1.2. Subconjuntos	4
1.3. Combinaciones de n en m	5
2. Estructuras básicas	5
2.1. Min stack	5
2.2. Min queue	5
3. Ordenamiento	6
3.1. Bucket sort	6
3.2. Merge sort	6
4. Matemáticas	7
4.1. Criba de Eratóstenes	7
4.2. Criba sobre un rango	7
4.3. Criba segmentada	7
4.4. Criba lineal	8
4.5. Algoritmo extendido de Euclides	8
4.6. Solución de ecuaciones diofánticas lineales	9
4.7. Función Phi de Euler	9
4.8. Función sigma	9
4.9. Función de Moebius	10
4.10. Exponenciación binaria	10
5. Sparse table	11
6. Fenwick Tree	11
7. Segment Tree	12
7.1. Actualizaciones puntuales	12
7.2. Actualizaciones sobre rangos	13
8. DSU	14
9. Grafos	14
9.1. Caminos mínimos	15
9.1.1. Dijkstra	15
9.2. Minimum Spanning Tree	15
9.2.1. Prim	15
9.3. Flujo máximo	16
10. Treap	17
11. Strings	18
11.1. KMP	18
11.2. Suffix array	19
11.2.1. Construcción	19
11.2.2. Prefijo común más largo	20
11.3. Suffix tree	20
12. Geometría	23
12.1. Convex hull	23

13.Utilidades	23
13.1. Plantilla tree	23
13.2. Números aleatorios	24
13.3. Bitmask	24

MAXN es el valor máximo de n .

1. Búsqueda completa

`check()` es la función de lo que se desea hacer con cada permutación o subconjunto.

1.1. Permutaciones

Iterativo. Complejidad: Tiempo $O(n \cdot n!)$ - Memoria extra $O(1)$

```
1 void generaPermutaciones(int arr[], int n){
2     do{
3         check(arr, n);
4     } while(next_permutation(arr, arr + n));
5 }
```

Rekursivo. Complejidad: Tiempo $O(n!)$ - Memoria extra $O(1)$

```
1 void generaPermutaciones(int arr[], int n, int actual = 0){
2     if(actual == n){
3         check(arr, n);
4         return;
5     }
6     for(int i = actual; i < n; ++i){
7         swap(arr[i], arr[actual]);
8         generaPermutaciones(arr, n, actual + 1);
9         swap(arr[i], arr[actual]);
10    }
11 }
```

1.2. Subconjuntos

Bitmask. Complejidad: Tiempo $O(2^n)$ - Memoria extra $O(1)$

```
1 void generaSubconjuntos(int arr[], int n){
2     int lim = 1 << n;
3     for(int i = 0; i < lim; ++i){
4         check(arr, n, i);
5     }
6 }
```

Rekursivo. Complejidad: Tiempo $O(2^n)$ - Memoria extra $O(n)$

```
1 void generaSubconjuntos(int arr[], bool used[], int n, int actual = 0){
2     if(actual == n){
3         check(arr, used, n);
4         return;
5     }
6     used[actual] = true;
7     generaSubconjuntos(arr, used, n, actual + 1);
8     used[actual] = false;
9     generaSubconjuntos(arr, used, n, actual + 1);
10 }
```

1.3. Combinaciones de n en m

Pendiente por mejorar implementación. orz

Bitmask. Complejidad: Tiempo $O(2^n)$ - Memoria extra $O(1)$

```
1 void generaCombinacionesEnM(int arr[], int n, int m){
2     if(n < m) return;
3     int lim = 1 << n;
4     for(int i = 0; i < lim; ++i){
5         if(__builtin_popcount(i) == m){
6             check(arr, n, i);
7         }
8     }
9 }
```

Recurso. Complejidad: Tiempo $O(2^n)$ - Memoria extra $O(n)$

```
1 void generaCombinacionesEnM(int arr[], bool used[], int n, int m, int actual = 0){
2     if(n < m) return;
3     if(actual == n || m == 0){
4         if(m != 0) return;
5         check(arr, used, n);
6         return;
7     }
8     used[actual] = true;
9     generaCombinacionesEnM(arr, used, n, m - 1, actual + 1);
10    used[actual] = false;
11    generaCombinacionesEnM(arr, used, n, m, actual + 1);
12 }
```

2. Estructuras básicas

2.1. Min stack

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

2.2. Min queue

```
1 struct min_queue{
2     min_stack p_in;
3     min_stack p_out;
4     void push(int v){p_in.push(v);}
5     int front(){transfer(); return p_out.top();}
6     void pop(){transfer(); p_out.pop();}
7     int size(){return p_in.size() + p_out.size();}
8     int minV(){return min(p_in.minV(), p_out.minV());}
```

```

9      bool empty(){ return size() == 0;}
10     void transfer(){
11         if(p_out.size()) return;
12         while(p_in.size()){
13             p_out.push(p_in.top());
14             p_in.pop();
15         }
16     }
17 };

```

3. Ordenamiento

3.1. Bucket sort

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

```

1  void bucketSort(int arr[], int n){
2      int cub[MAXVAL + 1] = {};
3      for(int i = 0; i < n; ++i)
4          cub[ arr[i] ]++;
5      int idx = 0;
6      for(int i = 0; i <= MAXVAL; ++i){
7          while( cub[i] ){
8              arr[idx] = i;
9              cub[i]--;
10             idx++;
11         }
12     }
13 }

```

3.2. Merge sort

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

```

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

```

4. Matemáticas

4.1. Criba de Eratóstenes

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

Calcula los primos menores o iguales a n .

```
1 void criba(int n, vector<int> &primos){
2     primos.clear();
3     if(n < 2) return;
4     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 }
```

4.2. Criba sobre un rango

Complejidad: Tiempo $O(\sqrt{b} \log \log \sqrt{b} + (b - a) \log \log(b - a))$ - Memoria extra $O(\sqrt{b} + b - a)$.

Calcula los primos en el rango $[a, b]$.

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

4.3. Criba segmentada

Complejidad: Tiempo $O(\sqrt{n} \log \log \sqrt{n} + n \log \log n)$ - Memoria extra $O(\sqrt{n} + S)$.

Cuenta la cantidad de primos menores o iguales a n .

```
1 int cuentaPrimos(int n){
2     if(n < 2) return 0;
3     const int S = sqrt(n);
```

```

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

```

4.4. Criba lineal

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

Calcula los primos menores o iguales a n y el menor primo que divide a cada entero en $[2, n]$.

```

1 void cribaLineal(int n, vector<int> &primos){
2     primos.clear();
3     if(n < 2) return;
4     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 }

```

4.5. Algoritmo extendido de Euclides

Complejidad: Tiempo $O(\log(\max(a, b)))$ - Memoria extra $O(1)$.

Encuentra una solución a la ecuación $ax + by = \gcd(a, b)$.

```

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

```


4.6. Solución de ecuaciones diofánticas lineales

Complejidad: Tiempo $O(\log(\max(a, b)))$ - Memoria extra $O(1)$.

Encuentra una solución a la ecuación $ax + by = c$ o determina si no existe solución.

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

4.7. Función Phi de Euler

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

Cuenta la cantidad de coprimos con n menores a n .

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

4.8. Función sigma

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

Cuenta la cantidad de divisores de n .

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

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

Calcula la suma de los divisores de n .

```

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

```

4.9. Función de Moebius

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

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

```

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

```

4.10. Exponenciación binaria

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

```

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

```

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

```

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

```

5. Sparse table

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

```
1 struct sparse_table{
2     int n, NEUTRO, ST[LOGN][MAXN], lg2[LOGN];
3     int f(int a, int b){
4         return a + b;
5     }
6     sparse_table(int n_size, int nums[]){
7         n = n_size;
8         NEUTRO = 0;
9         lg2[1] = 0;
10        for(int i = 2; i <= n; ++i)
11            lg2[i] = lg2[i / 2] + 1;
12        for(int i = 0; i < n; ++i)
13            ST[0][i] = nums[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        int ans = NEUTRO;
22        for(int k = lg2[n]; 0 <= k; --k){
23            if( r - l + 1 < (1 << k) ) continue;
24            ans = f(ans, ST[k][l]);
25            l += 1 << k;
26        }
27        return ans;
28    }
29    int queryIdem(int l, int r){
30        int lg = lg2[r - l + 1];
31        return f(ST[lg][l], ST[lg][r - (1 << lg) + 1]);
32    }
33 };
```

6. Fenwick Tree

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

```
1 struct FenwickTree{
2     int n, BIT[MAXN];
3     FenwickTree(int n_size){
4         n = n_size;
5         memset(BIT, 0, sizeof(BIT));
6     }
7     void add(int pos, int x){
8         while(pos <= n){
9             BIT[pos] += x;
10            pos += pos & -pos;
11        }
12    }
```

```

13     int sum(int pos){
14         int ret = 0;
15         while(pos){
16             ret += BIT[pos];
17             pos -= pos & -pos;
18         }
19         return ret;
20     }
21 };

```

7. Segment Tree

Nodo del Segment Tree:

```

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

```

7.1. Actualizaciones puntuales

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

```

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

```

```

28         if(r < left || right < l) return node(); /// Devuelve el neutro
29         if(l <= left && right <= r) return nodes[pos];
30         int mid = (left + right) / 2;
31         return query(l, r, left, mid, pos * 2) + query(l, r, mid + 1, right, pos * 2 + 1);
32     }
33 };

```

7.2. Actualizaciones sobre rangos

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

```

1  struct segment_tree{
2      struct node{...}nodes[4 * MAXN + 1];
3      segment_tree(int n, int data[]){
4          build(1, n, data);
5      }
6      void build(int left, int right, int data[], int pos = 1){
7          if(left == right){
8              nodes[pos].val = data[left];
9              return;
10         }
11         int mid = (left + right) / 2;
12         build(left, mid, data, pos * 2);
13         build(mid + 1, right, data, pos * 2 + 1);
14         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
15     }
16     void combineLazy(int lz, int pos){
17         nodes[pos].lazy += lz;
18     }
19     /// IMPORTANTE: ESTE UPDATE ES PARA SUMA/RESTA. CAMBIAR SI ES NECESARIO
20     void applyLazy(int pos, int tam){
21         nodes[pos].val += nodes[pos].lazy * tam;
22         nodes[pos].lazy = 0;
23     }
24     void pushLazy(int pos, int left, int right){
25         int tam = abs(right - left + 1);
26         if(1 < tam){
27             combineLazy(nodes[pos].lazy, pos * 2);
28             combineLazy(nodes[pos].lazy, pos * 2 + 1);
29         }
30         applyLazy(pos, tam);
31     }
32     void update(int x, int l, int r, int left, int right, int pos = 1){
33         pushLazy(pos, left, right);
34         if(r < left || right < l) return;
35         if(l <= left && right <= r){
36             combineLazy(x, pos);
37             pushLazy(pos, left, right);
38             return;
39         }
40         int mid = (left + right) / 2;
41         update(x, l, r, left, mid, pos * 2);
42         update(x, l, r, mid + 1, right, pos * 2 + 1);
43         nodes[pos] = nodes[pos * 2] + nodes[pos * 2 + 1];
44     }
45     node query(int l, int r, int left, int right, int pos = 1){

```

```

46     pushLazy(pos, left, right);
47     if(r < left || right < l) return node(); /// Devuelve el neutro
48     if(l <= left && right <= r) return nodes[pos];
49     int mid = (left + right) / 2;
50     return query(l, r, left, mid, pos * 2) + query(l, r, mid + 1, right, pos * 2 + 1);
51 }
52 };

```

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      int RA[MAXN], P[MAXN];
3      dsu(int n){
4          for(int i = 0; i < n; ++i){
5              RA[i] = 1;
6              P[i] = i;
7          }
8      }
9      int root(int x){
10         if(x == P[x]) return x;
11         return P[x] = root(P[x]);
12     }
13     void join(int x, int y){
14         int Px = root(x);
15         int Py = root(y);
16         if(Px == Py) return;
17         if(RA[Px] < RA[Py]){
18             RA[Py] += RA[Px];
19             P[Px] = Py;
20         } else {
21             RA[Px] += RA[Py];
22             P[Py] = Px;
23         }
24     }
25 };

```

9. Grafos

```

1  struct edge{
2      int to;
3      long long c;
4      const bool operator<(const edge &b) const{
5          return c > b.c;
6      }
7  };

```

9.1. Caminos mínimos

9.1.1. Dijkstra

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

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

9.2. Minimum Spanning Tree

9.2.1. Prim

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

```
1  vector<edge> grafo[MAXN];
2  long long eCost[MAXN];
3  bool visitado[MAXN];
4  long long prim(){
5      memset(visitado, 0, sizeof(visitado));
6      fill(eCost, eCost + MAXN, LLONG_MAX);
7      long long ans = 0;
8      priority_queue<pos> q;
9      q.push(pos{1, 0});
10     while(q.size()){
11         int node = q.top().to;
12         long long c = q.top().c;
13         q.pop();
14         if(visitado[node]) continue;
15         visitado[node] = true;
16         ans += c;
17         for(edge it : grafo[node]){
18             if(visitado[it.to] || eCost[it.to] <= it.c) continue;
19             eCost[it.to] = it.c;
20             q.push(it);
21         }
22     }
```

```

23     return ans;
24 }

```

9.3. Flujo máximo

Complejidad: $O(|E| \cdot \text{maxFlow})$.

```

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

```



```

51     if(!found) return false;
52     // Encontrar cap. minima del camino de aumento
53     long long uFlow = LLONG_MAX;
54     int current = t;
55     while(current != s) {
56         uFlow = min(uFlow, edges[from[current].second].c - edges[from[current].second].f);
57         current = from[current].first;
58     }
59     current = t;
60     // Actualizar flujo
61     while(current != s){
62         edges[from[current].second].f += uFlow;
63         edges[from[current].second^1].f -= uFlow; // Arista inversa
64         current = from[current].first;
65     }
66     flow += uFlow ;
67     return true;
68 }
69 };

```

10. Treap

AGREGAR PEQUEÑA DESCRIPCIÓN.

```

1  struct treap{
2      typedef struct _node{
3          long long x;
4          int freq, cnt;
5          long long p;
6          _node *l, *r;
7          _node(long long _x): x(_x), p(((long long)(rand()) << 32 )^rand()),
8          cnt(1), freq(1), l(nullptr), r(nullptr){}
9          ~_node(){delete l; delete r;}
10         void recalc(){
11             cnt = freq;
12             cnt += ((l) ? (l->cnt) : 0);
13             cnt += ((r) ? (r->cnt) : 0);
14         }
15     }* node;
16     node root;
17     node merge(node l, node r){
18         if(!l || !r) return l ? l : r;
19         if(l->p < r->p){
20             r->l = merge(l, r->l);
21             r->recalc();
22             return r;
23         } else {
24             l->r = merge(l->r, r);
25             l->recalc();
26             return l;
27         }
28     }
29     void split_by_value(node n, long long d, node &l, node &r){
30         l = r = nullptr;
31         if(!n) return;

```

```

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

```

11. Strings

11.1. KMP

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

```

1 vector<int> prefix_function(string s){
2     int n = (int)s.length();

```

```

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

```

11.2. Suffix array

11.2.1. Construcción

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

```

1  int SA[MAXN], mrank[MAXN];
2  int tmpSA[MAXN], tmpMrank[MAXN];
3  void countingSort(int k, int n){
4      int freqs[MAXN] = {};
5      for(int i = 0; i < n; ++i){
6          if(i + k < n) freqs[ mrank[i + k] ]++;
7          else freqs[0]++;
8      }
9      int m = max(100, n);
10     for(int i = 0, sfs = 0; i < m; ++i){
11         int f = freqs[i];
12         freqs[i] = sfs;
13         sfs += f;
14     }
15     for(int i = 0; i < n; ++i){
16         if(SA[i] + k < n) tmpSA[ freqs[mrank[ SA[i] + k ] ]++ ] = SA[i];
17         else tmpSA[ freqs[0]++ ] = SA[i];
18     }
19     for(int i = 0; i < n; ++i) SA[i] = tmpSA[i];
20 }
21
22 void buildSA(string &str){
23     int n = str.size();
24     for(int i = 0; i < n; ++i){
25         mrank[i] = str[i] - '#';
26         SA[i] = i;
27     }
28     for(int k = 1; k < n; k <= 1){
29         countingSort(k, n);
30         countingSort(0, n);
31         int r = 0;
32         tmpMrank[ SA[0] ] = 0;
33         for(int i = 1; i < n; ++i){
34             if(mrank[ SA[i] ] != mrank[ SA[i - 1] ] || mrank[ SA[i] + k ] != mrank[ SA[i - 1] + k ] )
35                 tmpMrank[ SA[i] ] = ++r;
36             else
37                 tmpMrank[ SA[i] ] = r;
38         }
39         for(int i = 0; i < n; ++i) mrank[i] = tmpMrank[i];
40     }

```

```

41 }
42 inline bool suff_compare1(int idx, const string &pattern) {
43     return (s.substr(idx).compare(0, pattern.size(), pattern) < 0);
44 }
45 inline bool suff_compare2(const string &pattern, int idx) {
46     return (s.substr(idx).compare(0, pattern.size(), pattern) > 0);
47 }
48 pair<int, int> match(const string &pattern) {
49     int *low = lower_bound (SA, SA + s.size(), pattern, suff_compare1);
50     int *up = upper_bound (SA, SA + s.size(), pattern, suff_compare2);
51     return make_pair((int)(low - SA), (int)(up - SA));
52 }

```

11.2.2. Prefijo común más largo

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

```

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

```

11.3. Suffix tree

COPIADO Y PEGADO POR

```

1  /// MEJORAR ESTA COSA, SOLO LO COPIE Y PEGUÉ por cuestiones de tiempo
2  const int inf = 1e9;
3  const int maxn = 1e6;
4  int s[maxn];
5  map<int, int> to[maxn];
6  //Root is the vertex 0
7  //f_pos[i] is the initial index with the letter of the edge that goes from the parent of i to i
8  //len[i] is the number of letters in the edge that enters in i
9  //slink[i] is the suffix link
10 int len[maxn], f_pos[maxn], slink[maxn];
11 int node, pos;
12 int sz = 1, n = 0;
13

```

```

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

```

```

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

```

```

126 |         cutGeneralized(finishPoints);
127 |     }

```

12. Geometría

12.1. Convex hull

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

```

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

```

13. Utilidades

13.1. Plantilla tree

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>

```

```

3 using namespace __gnu_pbds;
4 typedef tree<int, null_type, less(int), rb_tree_tag, tree_order_statistics_node_update> ordered_set;
5
6 int main(){
7     ordered_set S;
8     S.insert(1);
9     S.insert(4);
10    S.insert(0);
11    S.insert(-10);
12    cout << S.order_of_key(1) << '\n'; /// 2
13    cout << S.order_of_key(4) << '\n'; /// 3
14    cout << *(S.find_by_order(1)) << '\n'; /// 0
15 }

```

13.2. Números aleatorios

mt19937_64 genera números de 64 bits.

```

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

```

13.3. Bitmask

```

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

```