

# Índice

<b>1. Setup</b>	<b>2</b>	<b>6. Matemática</b>	<b>4</b>
1.1. Template	2	6.1. Aritmética	4
1.2. Makefile	2	6.1.1. Techo de la división, piso de la raíz cuadrada, piso del log2	4
1.3. Compilar	2	6.1.2. Aritmética en $\mathbb{Z}_p$	5
1.4. Correr	2	6.1.3. Números combinatorios	5
<b>2. STL</b>	<b>2</b>	6.2. Teoría de números	5
2.1. Búsqueda binaria en vector ordenado	2	6.2.1. Test de primalidad	5
2.2. Priority queue custom compare	2	6.3. Geometría	5
2.3. Intervalos consecutivos	2	6.3.1. Template base	5
2.4. Indexed set y multiset	3	6.3.2. Punto/vector/recta	6
<b>3. Range queries</b>	<b>3</b>	6.3.3. Producto escalar y vectorial	6
3.1. Prefix + diff arrays	3	6.3.4. Área triángulo	6
3.1.1. 1D	3	6.3.5. Fórmula de Herón	6
3.1.2. 2D	3	<b>7. Programación Dinámica</b>	<b>6</b>
3.2. Fenwick tree	3	7.1. Ejemplos de DP	6
3.2.1. Range query point update	3	7.1.1. DP en prefijo:	6
3.2.2. Range update point query	3	7.1.2. DP en rango:	6
3.2.3. Range update range query	3	7.1.3. DP en bitmask: traveling salesman	6
3.3. Operaciones sin inverso	3	7.1.4. DP en árbol con toposort	6
3.3.1. Sparse table	3	7.1.5. DP en DAG:	6
3.3.2. Segment tree: range query point set	4	7.1.6. DP en número: knapsack	6
<b>4. Grafos</b>	<b>4</b>	7.1.7. Re-rooting DP	6
4.1. Preprocesamiento	4	7.1.8. DP con Fenwick: número de subsecuencias de largo k	6
4.1.1. Clasificación de aristas	4	7.1.9. Reconstruir solución	6
4.1.2. Puentes y puntos de articulación	4	7.2. Optimizaciones	6
4.1.3. DAG condensado	4	7.2.1. Recuperar un parámetro a partir de los otros	6
4.1.4. Kruskal	4	7.2.2. Reducir complejidad de transición con una flag	6
4.2. Menor camino	4	7.2.3. Optimización knapsack en árbol	6
4.2.1. BFS	4	7.2.4. Optimización de Knuth	6
4.2.2. Dijkstra	4	7.2.5. Optimización D&C	6
4.2.3. Floyd-Warshall	4	<b>8. Algoritmos</b>	<b>6</b>
4.3. Flujo y corte	4	8.1. Búsqueda binaria	6
4.3.1. Dinics	4	8.2. Búsqueda binaria paralela	6
4.3.2. Maximum matching	4	<b>9. Sin categorizar</b>	<b>6</b>
4.4. Árboles	4	9.1. Union find	6
4.4.1. Aplanamiento	4	9.2. Algoritmo de Mo	6
4.4.2. Ancestro común menor	4	9.3. Min dequeue	6
<b>5. Strings</b>	<b>4</b>	9.4. Menor subarray que suma k	6
5.1. Trie: policy based	4	9.5. Subarray con mayor suma	6
5.2. Trie genérico	4	9.6. Mayor subcadena común	6
5.3. Rabin-Karp	4	<b>10. Brainstorming</b>	<b>6</b>

## 1. Setup

### 1.1. Template

```
#include <bits/stdc++.h>
using namespace std;
using i64 = int64_t;
#define endl      '\n'
#define forn(i,N)  for (int i = 0; i < int(N); i++)
#define dbg(x)     cerr << #x << " = " << (x) << endl
#define raya      cerr << "=====" << endl
#define all(v)     begin(v), end(v)
#define pb         push_back
#define mp         make_pair
#define fst        first
#define snd        second
#define forall(it,v) for (auto it = begin(v); it != end(v); it++)
#define printall(v) forall(x,v) { cout << *x << " "; } cout << endl
#define printpair(p) cout << "(" << p.fst << ", " << p.snd << ")" << endl

int main (void) {
    ios::sync_with_stdio(0); cin.tie(0);

    return 0;
}
```

### 1.2. Makefile

```
1 CPPFLAGS = -std=c++17 -O0 -Wall -g
2 CC = g++
```

### 1.3. Compilar

Compilar \$1 y mostrar primeras \$2 líneas de error

```
1 rm -f $1
2 clear
3 make $1 2>&1 | head -$2
```

### 1.4. Correr

Correr \$1 con el input \$2

```
1 rm -f $1
2 clear
3 make $1 && ./ $1 < $2
```

## 2. STL

### 2.1. Búsqueda binaria en vector ordenado

```
template <class T> int primer_igual (vector<T>& arr, T x) {
    auto it = lower_bound(all(arr), x);
    if (it == arr.end() || *it != x) return -1;
    return it - arr.begin();
}

template <class T> int ultimo_igual (vector<T>& arr, T x) {
    if (arr.begin() == arr.end()) return -1;
    auto it = prev(upper_bound(all(arr), x));
    if (*it != x) return -1;
    return it - arr.begin();
}

template <class T> int ultimo_menor (vector<T>& arr, T x) {
    if (arr.begin() == arr.end()) return -1;
    auto it = prev(lower_bound(all(arr), x));
    if (*it >=) return -1;
    return it - arr.begin();
}

template <class T> int primer_mayor (vector<T>& arr, T x) {
    auto it = upper_bound(all(arr), x);
    if (it == arr.end()) return -1;
    return it - arr.begin();
}
```

### 2.2. Priority queue custom compare

```
priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int>>> pq;
template <typename T> using MinHeap = // ...
```

### 2.3. Intervalos consecutivos

```
struct IntervalosConsecutivos {
    set<int> I;
    map<int, int> L;
    IntervalosConsecutivos (int i, int j) {
        I.insert(i);
        I.insert(j);
        L[j - i]++;
    }
    void cortar (int k) {
        int i = *prev(I.lower_bound(k));
```

```

        int j = *(I.lower_bound(k));
        L[j - i]--;
        if (L[j - i] == 0) L.erase(j - i);
        L[k - i]++;
        L[j - k]++;
        I.insert(k);
    }
    i64 max_intervalo () {
        return (*L.rbegin()).fst;
    }
};

```

## 2.4. Indexed set y multiset

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<class T> struct IndexedSet {
    tree<
        T, null_type, less<T>,
        rb_tree_tag, tree_order_statistics_node_update
    > s;
    void add (T x) { ms.insert(x); }
    int idx (T x) { return ms.order_of_key(x); }
    bool has (T x) { return ms.find(x) != ms.end(); }
    T ith (int i) { return *ms.find_by_order(i); }
};

template<class T> struct IndexedMultiset {
    int t = 0; tree<
        pair<T, int>, null_type, less<pair<T, int>>,
        rb_tree_tag, tree_order_statistics_node_update
    > ms;
    void add (T x) { ms.insert(mp(x, t++)); }
    int nle (T x) { return ms.order_of_key(mp(x, -1)); }
    int nleq (T x) { return ms.order_of_key(mp(x, INT_MAX)); }
    int cnt (T x) { return nleq(x) - nle(x); }
    T ith (int i) { return (*ms.find_by_order(i)).fst; }
};

```

## 3. Range queries

### 3.1. Prefix + diff arrays

#### 3.1.1. 1D

Usar array indexado desde 1 con  $A[0] = 0$ .

Usar intervalos cerrado-cerrado (indexados desde 1).

```

vector<int> make_diff_array (vector<int>& A) {
    vector<int> D(A.size() - 1);
    forn(i, A.size() - 1) D[i] = A[i+1] - A[i];
    return D;
}

void diff_array_range_update (vector<int>& D, int i, int j, int v) {
    D[i-1] += v;
    D[j] -= v;
}

vector<int> make_prefix_array (vector<int>& A) {
    vector<int> P(A.size() + 1);
    P[0] = 0;
    forn(i, A.size()) P[i+1] = P[i] + A[i];
    return P;
}

int prefix_array_range_query (vector<int>& P, int i, int j) {
    return P[j+1] - P[i];
}

```

#### 3.1.2. 2D

### 3.2. Fenwick tree

#### 3.2.1. Range query point update

#### 3.2.2. Range update point query

#### 3.2.3. Range update range query

### 3.3. Operaciones sin inverso

#### 3.3.1. Sparse table

Operacion asociativa **idempotente**

```

// Operacion IDEMPOTENTE

#define log2fl(x) (x ? 63 - __builtin_clzll(x) : -1)

```

```
struct STable {
    vector<int>& arr; int N;
    vector<vector<int>>> st;
    int op (int a, int b) { return min(a,b); }
    void make () {
        st.resize(20, vector<int>(N));
        st[0] = arr; scn(w,1,19) scn(i,0,N - (1 << w))
            st[w][i] = op(st[w-1][i], st[w-1][i + (1 << (w-1))]);
    }
    int q (int i, int j) {
        int w = log2fl(j - i + 1);
        return op(st[w][i], st[w][j - (1 << w) + 1]);
    }
};
```

3.3.2. Segment tree: range query point set

Recordatorio: modificar elemento neutro

```
template<class T> struct SegTree {
    vector<T>& arr; int N; T id;
    T op (T a, T b) { return 0; } // !
    vector<T> t;
    void make () {
        t.resize(N << 1); rep(i,N) t[i+N] = arr[i];
        for (int i = N - 1; i; i--) t[i] = op(t[i<<1], t[i<<1|1]);
    }
    void set (int i, T v) {
        for(t[i += N] = v; i > 1; i >>= 1) t[i>>1] = op(t[i], t[i^1]);
    }
    T q (int l, int r) {
        T res = id;
        for (l += N, r += N; l < r; l >>= 1, r >>= 1) {
            if (l&1) res = op(res, t[l++]);
            if (r&1) res = op(res, t[--r]);
        } return res;
    }
};
```

4. Grafos

4.1. Preprocesamiento

4.1.1. Clasificación de aristas

4.1.2. Puentes y puntos de articulación

4.1.3. DAG condensado

4.1.4. Kruskal

4.2. Menor camino

4.2.1. BFS

4.2.2. Dijkstra

4.2.3. Floyd-Warshall

4.3. Flujo y corte

4.3.1. Dinics

4.3.2. Maximum matching

4.4. Árboles

4.4.1. Aplanamiento

4.4.2. Ancestro común menor

5. Strings

5.1. Trie: policy based

5.2. Trie genérico

5.3. Rabin-Karp

5.4. Suffix Automata

6. Matemática

6.1. Aritmética

6.1.1. Techo de la división, piso de la raíz cuadrada, piso del log2

```
#define ceildiv(a,b) ((a+b-1)/b)

i64 isqrt (i64 x) {
    i64 s = 0; for (i64 k = 1 << 30; k; k >>= 1)
        if ((s+k)*(s+k) <= x) s += k;
```

```
    return s;
}

#define log2fl(x) (x ? 63 - __builtin_clzll(x) : -1)
```

### 6.1.2. Aritmética en $\mathbb{Z}_p$

```
const i64 mod = 1e9 + 7;

i64 resta_mod (i64 a, i64 b) { return (a - b + mod) % mod; }

i64 pow_mod (i64 x, i64 n) {
    i64 res = 0;
    while (n) {
        if (n % 2) res = res * x % mod;
        n /= 2;
        x = x * x % mod;
    } return res;
}

i64 div_mod (i64 a, i64 b) { return a * pow_mod(b, mod - 2) % mod; }
```

### 6.1.3. Números combinatorios

## 6.2. Teoría de números

### 6.2.1. Test de primalidad

```
struct primetest {
    bool c[1000001]; vector<int> p;
    primetest () {
        p.reserve(1<<16); scn(i,2,1000000) if (!c[i]) {
            p.pb(i); for (int j = 2; i*j < 1000001; j++) c[i*j] = 1;
        }
    }
    bool isprime (int x) {
        for (int i = 0, d = p[i]; d*d <= x; d = p[++i])
            if (!(x % d)) return false;
        return x >= 2;
    }
};
```

## 6.3. Geometría

### 6.3.1. Template base

```
using flt = long double;
const flt EPS = 1e-9;
bool flt_leq (flt a, flt b) { return a < b + EPS; }
```

```
bool flt_eq (flt a, flt b) { return -EPS <= a - b && a - b <= EPS; }
```

```
struct Vec {
    int x, y;
    Vec operator+(Vec p) { return {x + p.x, y + p.y}; }
    Vec operator-(Vec p) { return {x - p.x, y - p.y}; }
    int operator*(Vec p) { return x * p.x + y * p.y; }
    int operator^(Vec p) { return x * p.y - y * p.x; }
};
```

```
int norma2 (Vec p) { return p.x * p.x + p.y * p.y; }
```

```
// TODO: area triangulo, formula de heron
```

6.3.2. Punto/vector/recta

6.3.3. Producto escalar y vectorial

6.3.4. Área triángulo

6.3.5. Fórmula de Herón

7. Programación Dinámica

7.1. Ejemplos de DP

7.1.1. DP en prefijo:

7.1.2. DP en rango:

7.1.3. DP en bitmask: traveling salesman

7.1.4. DP en árbol con toposort

7.1.5. DP en DAG:

7.1.6. DP en número: knapsack

7.1.7. Re-rooting DP

7.1.8. DP con Fenwick: número de subsecuencias de largo k

7.1.9. Reconstruir solución

7.2. Optimizaciones

7.2.1. Recuperar un parámetro a partir de los otros

7.2.2. Reducir complejidad de transición con una flag

7.2.3. Optimización knapsack en árbol

7.2.4. Optimización de Knuth

7.2.5. Optimización D&C

8. Algoritmos

8.1. Búsqueda binaria

Si existe, idx de primer true Si no, d

```
i64 bsearch (i64 i, i64 j, bool (*pred)(i64), i64 d) {
    while (!(i + 1 == j)) {
        i64 m = i + ((j - i) >> 1);
        pred(m) ? j = m : i = m;
    }
    if (pred(i)) return i;
    if (pred(j)) return j;
```

```
    return d;
}
```

8.2. Búsqueda binaria paralela

9. Sin categorizar

9.1. Union find

9.2. Algoritmo de Mo

9.3. Min dequeue

9.4. Menor subarray que suma k

9.5. Subarray con mayor suma

9.6. Mayor subcadena común

10. Brainstorming

Graficar como puntos/grafos  
Pensarlo al revez  
¿Que propiedades debe cumplir una solución?  
Si existe una solución, ¿existe otra más simple?  
¿Hay elecciones independientes?  
¿El proceso es parecido a un algoritmo conocido?