# Competitive Programming Notes

Ignacio Hermosilla, Nicholas Mc-Donnell, Javier Reyes

2018

## Contents

# 1 Notas Útiles

| $O(f(n))$ | Limite |
|---|---|
| $O(n!)$ | 10...11 |
| $O(2^n n^2)$ | 15...18 |
| $O(2^n n)$ | 18...21 |
| $O(n^4)$ | 100 |
| $O(n^3)$ | 500[1] |
| $O(n^2 \log^2 n)$ | 1000 |
| $O(n^2 \log n)$ | 2000 |
| $O(n^2)$ | 1e4[2] |
| $O(n \log^2 n)$ | 3e5 |
| $O(n \log n)$ | 1e6 |
| $O(n)$ | 1e8[3] |

| Primos hasta | |
|---|---|
| 1e2 | 25 |
| 1e3 | 168 |
| 1e4 | 1229 |
| 1e5 | 9592 |
| 1e6 | 78.498 |
| 1e7 | 664.579 |
| 1e8 | 5.761.455 |
| 1e9 | 50.487.534 |

El mayor HCN[4] menor a 1e9 tiene 1344 divisores.
El "prime gap" hasta 1e9 es a lo más 288

[1]Este caso esta justo en el limite de tiempo, además en 256 MB cabe a los una matriz de $400^3$ ints
[2]En general solo funciona hasta 6e3
[3]En general solo funciona hasta 4e7
[4]Highly Compositve Number, números que tienen más divisores que cualquier número más pequeño

## 2 ./headers/headers/headers.h

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef vector<ii> vii;

typedef vector<vi> graph;
typedef vector<vii> wgraph;

#ifndef declaraciones_h
#define declaraciones_h

#define rep(i, n) for (int i = 0; i < (int)n; i++)
#define repx(i, a, b) for (int i = a; i < (int)b; i++)
#define invrep(i, a, b) for (int i = b; i-- > (int)a;)

#define pb push_back
#define eb emplace_back
#define ppb pop_back

#define lg(x) (31 - __builtin_clz(x))
#define lgg(x) (63 - __buitlin_clzll(x))
#define gcd __gcd

#define INF INT_MAX

#define umap unordered_map
#define uset unordered_set

//ios::sync_with_stdio(0); cin.tie(0);
//cout.setf(ios::fixed); cout.precision(4);

#define debugx(x) cerr << #x << ": " << x << endl
#define debugv(v)         \
    cerr << #v << ":";    \
    for (auto e : v)      \
    {                     \
        cerr << " " << e; \
    }                     \
```

```
    cerr << endl
#define debugm(m)                                           \
    cerr << #m << endl;                                     \
    rep(i, (int)m.size())                                   \
    {                                                       \
        cerr << i << ":";                                   \
        rep(j, (int)m[i].size()) cerr << " " << m[i][j];    \
        cerr << endl;                                       \
    }
#define debugmp(m)                                                              \
    cerr << #m << endl;                                                         \
    rep(i, (int)m.size())                                                       \
    {                                                                           \
        cerr << i << ":";                                                       \
        rep(j, (int)m[i].size())                                                \
        {                                                                       \
            cerr << " {" << m[i][j].first << "," << m[i][j].second << "}";     \
        }                                                                       \
        cerr << endl;                                                           \
    }
#define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, \")), cout << endl

#endif
```

## 3 ./strings/searching/kmp.cpp

```cpp
#include "../../headers/headers/headers.h"

vi prefix(string &S)
{
    vector<int> p(S.size());
    p[0] = 0;
    for (int i = 1; i < S.size(); ++i)
    {
        p[i] = p[i - 1];
        while (p[i] > 0 && S[p[i]] != S[i])
            p[i] = p[p[i] - 1];
        if (S[p[i]] == S[i])
            p[i]++;
    }
    return p;
}

vi KMP(string &P, string &S)
{
    vector<int> pi = prefix(P);
    vi matches;
    int n = S.length(), m = P.length();
    int j = 0, ans = 0;
    for (int i = 0; i < n; ++i)
    {
        while (j > 0 && S[i] != P[j])
            j = pi[j - 1];
        if (S[i] == P[j])
            ++j;

        if (j == P.length())
        {
            /* This is where KMP found a match
             * we can calculate its position on S by using i - m + 1
             * or we can simply count it
             */
            ans += 1; // count the number of matches
            matches.eb(i - m + 1); // store the position of those matches
            // return; we can return on the first match if needed
            // this must stay the same
            j = pi[j - 1];
        }
    }
    return matches; // can be modified to return number of matches or location
```

```
}
```

# 4 ./strings/trie/trie.cpp

```cpp
#include "../../headers/headers/headers.h"

/* Implementation from: https://pastebin.com/fyqsH65k */
struct TrieNode
{
    int leaf; // number of words that end on a TrieNode (allows for duplicate words)
    int height; // height of a TrieNode, root starts at height = 1, can be changed with the default value
    // number of words that pass through this node,
    // ask root node for this count to find the number of entries on the whole Trie
    // all nodes have 1 as they count the words than end on themselves (ie leaf nodes count themselves)
    int count;
    TrieNode *parent; // pointer to parent TrieNode, used on erasing entries
    map<char, TrieNode *> child;
    TrieNode(TrieNode *parent = NULL, int height = 1):
        parent(parent),
        leaf(0),
        height(height),
        count(0), // change to -1 if leaf nodes are to have count 0 insead of 1
        child()
    {}
};

/**
 * Complexity: O(|key| * log(k))
 */
TrieNode *trie_find(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return NULL;
        pNode = pNode->child[*key];
    }
    return (pNode->leaf) ? pNode : NULL; // returns only whole word
    // return pNode; // allows to search for a suffix
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_insert(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
```

```
    root -> count += 1;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
        pNode = pNode->child[*key];
        pNode -> count += 1;
    }
    pNode->leaf += 1;
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_erase(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    string::const_iterator key = str.begin();
    for (; key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return;
        pNode = pNode->child[*key];
    }
    pNode->leaf -= 1;
    pNode->count -= 1;
    while (pNode->parent != NULL)
    {
        if (pNode->child.size() > 0 || pNode->leaf)
            break;
        pNode = pNode->parent, key--;
        pNode->child.erase(*key);
        pNode->count -= 1;
    }
}
```

## 5 ./estructuras/segmentTree/lazySegmentTree.cpp

```cpp
#include "../../headers/headers/headers.h"

struct RSQ // Range sum query
{
    static ll const neutro = 0;
    static ll op(ll x, ll y)
    {
        return x + y;
    }
    static ll
    lazy_op(int i, int j, ll x)
    {
        return (j - i + 1) * x;
    }
};

struct RMinQ // Range minimun query
{
    static ll const neutro = 1e18;
    static ll op(ll x, ll y)
    {
        return min(x, y);
    }
    static ll
    lazy_op(int i, int j, ll x)
    {
        return x;
    }
};

template <class t>
class SegTreeLazy
{
    vector<ll> arr, st, lazy;
    int n;

    void build(int u, int i, int j)
    {
        if (i == j)
        {
            st[u] = arr[i];
            return;
        }
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
```

```
    build(l, i, m);
    build(r, m + 1, j);
    st[u] = t::op(st[l], st[r]);
}

void propagate(int u, int i, int j, ll x)
{
    // nota, las operaciones pueden ser un and, or, ..., etc.
    st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
    // st[u] = t::lazy_op(i, j, x); // setear el valor
    if (i != j)
    {
        // incrementar el valor
        lazy[u * 2 + 1] += x;
        lazy[u * 2 + 2] += x;
        // setear el valor
        //lazy[u * 2 + 1] = x;
        //lazy[u * 2 + 2] = x;
    }
    lazy[u] = 0;
}

ll query(int a, int b, int u, int i, int j)
{
    if (j < a or b < i)
        return t::neutro;
    int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
    if (lazy[u])
        propagate(u, i, j, lazy[u]);
    if (a <= i and j <= b)
        return st[u];
    ll x = query(a, b, l, i, m);
    ll y = query(a, b, r, m + 1, j);
    return t::op(x, y);
}

void update(int a, int b, ll value,
            int u, int i, int j)
{
    int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
    if (lazy[u])
        propagate(u, i, j, lazy[u]);
    if (a <= i and j <= b)
        propagate(u, i, j, value);
    else if (j < a or b < i)
        return;
```

```
        else
        {
            update(a, b, value, l, i, m);
            update(a, b, value, r, m + 1, j);
            st[u] = t::op(st[l], st[r]);
        }
    }

  public:
    SegTreeLazy(vector<ll> &v)
    {
        arr = v;
        n = v.size();
        st.resize(n * 4 + 5);
        lazy.assign(n * 4 + 5, 0);
        build(0, 0, n - 1);
    }

    ll query(int a, int b)
    {
        return query(a, b, 0, 0, n - 1);
    }

    void update(int a, int b, ll value)
    {
        update(a, b, value, 0, 0, n - 1);
    }
};
```

## 6 ./estructuras/segmentTree/segmentTree.cpp

```cpp
#include "../../headers/headers/headers.h"

// Se requiere un struct para el nodo (ej: prodsgn).
// Un nodo debe tener tres constructores:
//     Aridad 0: Construye el neutro de la operación
//     Aridad 1: Construye un nodo hoja a partir del input
//     Aridad 2: Construye un nodo según sus dos hijos
//
// Construcción del segment tree:
//     Hacer un arreglo de nodos (usar ctor de aridad 1).
//     ST<miStructNodo> miSegmentTree(arregloDeNodos);
// Update:
//     miSegmentTree.set_point(indice, miStructNodo(input));
// Query:
//     miSegmentTree.query(l, r) es inclusivo exclusivo y da un nodo. Usar la info del nodo para obtener l


// Logic And Query
struct ANDQ
{
    intt value;
    ANDQ() { value = -1ll; }
    ANDQ(intt x) { value = x; }
    ANDQ(const ANDQ &a,
         const ANDQ &b)
    {
        value = a.value & b.value;
    }
};

// Interval Product (LiveArchive)
struct prodsgn {
    int sgn;
    prodsgn() {sgn = 1;}
    prodsgn(int x) {
        sgn = (x > 0) - (x < 0);
    }
    prodsgn(const prodsgn &a,
            const prodsgn &b) {
        sgn = a.sgn*b.sgn;
    }
};

// Maximum Sum (SPOJ)
```

```
struct maxsum {
    int first, second;
    maxsum() {first = second = -1;}
    maxsum(int x) {
        first = x; second = -1;
    }
    maxsum(const maxsum &a,
           const maxsum &b) {
        if (a.first > b.first) {
            first = a.first;
            second = max(a.second,
                         b.first);
        } else {
            first = b.first;
            second = max(a.first,
                         b.second);
        }
    }
    int answer() {
        return first + second;
    }
};

// Range Minimum Query
struct rminq {
    int value;
    rminq() {value = INT_MAX;}
    rminq(int x) {value = x;}
    rminq(const rminq &a,
          const rminq &b) {
        value = min(a.value,
                    b.value);
    }
};

template <class node>
class ST
{
    vector<node> t;
    int n;

  public:
    ST(vector<node> &arr)
    {
        n = arr.size();
        t.resize(n * 2);
```

```
        copy(arr.begin(), arr.end(), t.begin() + n);
        for (int i = n - 1; i > 0; --i)
            t[i] = node(t[i << 1], t[i << 1 | 1]);
    }

    // 0-indexed
    void set_point(int p, const node &value)
    {
        for (t[p += n] = value; p > 1; p >>= 1)
            t[p >> 1] = node(t[p], t[p ^ 1]);
    }

    // inclusive exclusive, 0-indexed
    node query(int l, int r)
    {
        node ansl, ansr;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1)
        {
            if (l & 1)
                ansl = node(ansl, t[l++]);
            if (r & 1)
                ansr = node(t[--r], ansr);
        }
        return node(ansl, ansr);
    }
};
```

## 7 ./estructuras/fenwickTree/fenwickTree2D.cpp

```cpp
#include "../../headers/headers/headers.h"
//Numeración en [0,n-1] y [0,m-1]
template <class T>
class FenwickTree2D
{
    vector<vector<T>> t;
    int n, m;

  public:
    FenwickTree2D() {}

    FenwickTree2D(int n, int m)
    {
        t.assign(n, vector<T>(m, 0));
        this->n = n;
        this->m = m;
    }

    void add(int r, int c, T value)
    {
        for (int i = r; i < n; i |= i + 1)
            for (int j = c; j < m; j |= j + 1)
                t[i][j] += value;
    }

    // sum[(0, 0), (r, c)]
    T sum(int r, int c)
    {
        T res = 0;
        for (int i = r; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = c; j >= 0; j = (j & (j + 1)) - 1)
                res += t[i][j];
        return res;
    }

    // sum[(r1, c1), (r2, c2)]
    T sum(int r1, int c1, int r2, int c2)
    {
        return sum(r2, c2) - sum(r1 - 1, c2) - sum(r2, c1 - 1) + sum(r1 - 1, c1 - 1);
    }

    T get(int r, int c)
    {
        return sum(r, c, r, c);
```

```
    }

    void set(int r, int c, T value)
    {
        add(r, c, -get(t, r, c) + value);
    }
};
```

## 8 ./estructuras/fenwickTree/fenwickTree.cpp

```cpp
#include "../../headers/headers/headers.h"

struct FenwickTree
{
    vector<int> FT;
    FenwickTree(int N)
    {
        FT.resize(N + 1, 0);
    }

    int query(int i)
    {
        int ans = 0;
        for (; i; i -= i & (-i))
            ans += FT[i];
        return ans;
    }

    int query(int i, int j)
    {
        return query(j) - query(i - 1);
    }

    void update(int i, int v)
    {
        int s = query(i, i);
        for (; i < FT.size(); i += i & (-i))
            FT[i] += v - s;
    }

    //Queries puntuales, Updates por rango
    void update(int i, int j, int v)
    {
        update(i, v);
        update(j + 1, -v);
    }
};
```

## 9 ./math/factorialFactorization/factorialFactorization.cpp

```cpp
#include "../../headers/headers/headers.h"

// O(n)
umap<int, int> factorialFactorization(int n, vi &primes)
{
    umap<int, int> p2e;
    for (auto p : primes)
    {
        if (p > n)
            break;
        int e = 0;
        int tmp = n;
        while ((tmp /= p) > 0)
            e += tmp;
        if (e > 0)
            p2e[p] = e;
    }
    return p2e;
}
```

## 10 ./math/polynomials/polynomials.cpp

```cpp
#include "../../headers/headers/headers.h"

template <class T>
class Pol
{
  private:
    vector<T> cofs;
    int n;

  public:
    Pol(vector<T> cofs) : cofs(cofs)
    {
        this->n = cofs.size() - 1;
    }

    Pol<T> operator+(const Pol<T> &o)
    {
        vector<T> n_cofs;
        if (n > o.n)
        {
            n_cofs = cofs;
            rep(i, o.n + 1)
            {
                n_cofs[i] += o.cofs[i];
            }
        }
        else
        {
            n_cofs = o.cofs;
            rep(i, n + 1)
            {
                n_cofs[i] += cofs[i];
            }
        }
        return Pol(n_cofs);
    }

    Pol<T> operator-(const Pol<T> &o)
    {
        vector<T> n_cofs;
        if (n > o.n)
        {
            n_cofs = cofs;
            rep(i, o.n + 1)
```

```
        {
            n_cofs[i] -= o.cofs[i];
        }
    }
    else
    {
        n_cofs = o.cofs;
        rep(i, n + 1)
        {
            n_cofs[i] *= -1;
            n_cofs[i] += cofs[i];
        }
    }
    return Pol(n_cofs);
}

Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when we implement it
{
    vector<T> n_cofs(n + o.n + 1);
    rep(i, n + 1)
    {
        rep(j, o.n + 1)
        {
            n_cofs[i + j] += cofs[i] * o.cofs[j];
        }
    }
    return Pol(n_cofs);
}

Pol<T> operator*(const T &o)
{
    vector<T> n_cofs = cofs;
    for (auto &cof : n_cofs)
    {
        cof *= o;
    }
    return Pol(n_cofs);
}

double operator()(double x)
{
    double ans = 0;
    double temp = 1;
    for (auto cof : cofs)
    {
        ans += (double)cof * temp;
```

```
            temp *= x;
        }
        return ans;
    }

    Pol<T> integrate()
    {
        vector<T> n_cofs(n + 2);
        repx(i, 1, n_cofs.size())
        {
            n_cofs[i] = cofs[i - 1] / T(i);
        }
        return Pol<T>(n_cofs);
    }

    double integrate(T a, T b)
    {
        Pol<T> temp = integrate();
        return temp(b) - temp(a);
    }

    friend ostream &operator<<(ostream &str, const Pol &a);
};

ostream &operator<<(ostream &strm, const Pol<double> &a)
{
    bool flag = false;
    rep(i, a.n + 1)
    {
        if (a.cofs[i] == 0)
            continue;

        if (flag)
            if (a.cofs[i] > 0)
                strm << " + ";
            else
                strm << " - ";
        else
            flag = true;
        if (i > 1)
        {
            if (abs(a.cofs[i]) != 1)
                strm << abs(a.cofs[i]);
            strm << "x^" << i;
        }
        else if (i == 1)
```

```
        {
            if (abs(a.cofs[i]) != 1)
                strm << abs(a.cofs[i]);
            strm << "x";
        }
        else
        {
            strm << a.cofs[i];
        }
    }
    return strm;
}
```

## 11 ./math/simpsonsMethod/simpsonsMethod.cpp

```cpp
#include "../../headers/headers/headers.h"

//O(Evaluate f)=g(f)
//Numerical Integration of f in interval [a,b]
double simpsons_rule(function<double(double)> f, double a, double b)
{
    double c = (a + b) / 2;
    double h3 = abs(b - a) / 6;
    return h3 * (f(a) + 4 * f(c) + f(b));
}

//O(n g(f))
//Integrate f between a and b, using intervals of length (b-a)/n
double simpsons_rule(function<double(double)> f, double a, double b, int n)
{
    //n sets the precision for the result
    double ans = 0;
    double step = 0, h = (b - a) / n;
    rep(i, n)
    {
        ans += simpsons_rule(f, step, step + h);
        step += h;
    }
    return ans;
}
```

## 12 ./graphs/dinic/dinic.cpp

```cpp
#include "../../headers/headers/headers.h"
class Dinic
{
    struct edge
    {
        int to, rev;
        ll f, cap;
    };

    vector<vector<edge>> g;
    vector<ll> dist;
    vector<int> q, work;
    int n, sink;

    bool bfs(int start, int finish)
    {
        dist.assign(n, -1);
        dist[start] = 0;
        int head = 0, tail = 0;
        q[tail++] = start;
        while (head < tail)
        {
            int u = q[head++];
            for (const edge &e : g[u])
            {
                int v = e.to;
                if (dist[v] == -1 and e.f < e.cap)
                {
                    dist[v] = dist[u] + 1;
                    q[tail++] = v;
                }
            }
        }
        return dist[finish] != -1;
    }

    ll dfs(int u, ll f)
    {
        if (u == sink)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); ++i)
        {
            edge &e = g[u][i];
            int v = e.to;
```

```
            if (e.cap <= e.f or dist[v] != dist[u] + 1)
                continue;
            ll df = dfs(v, min(f, e.cap - e.f));
            if (df > 0)
            {
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
        return 0;
    }

  public:
    Dinic(int n)
    {
        this->n = n;
        g.resize(n);
        dist.resize(n);
        q.resize(n);
    }

    void add_edge(int u, int v, ll cap)
    {
        edge a = {v, (int)g[v].size(), 0, cap};
        edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si la arista es bidireccional
        g[u].pb(a);
        g[v].pb(b);
    }

    ll max_flow(int source, int dest)
    {
        sink = dest;
        ll ans = 0;
        while (bfs(source, dest))
        {
            work.assign(n, 0);
            while (ll delta = dfs(source, LLONG_MAX))
                ans += delta;
        }
        return ans;
    }
};
```

## 13 ./graphs/dijsktra/dijsktra.cpp

```cpp
#include "../../headers/headers/headers.h"
//g has vectors of pairs of the form (w, index)
int dijsktra(wgraph g, int start, int end)
{
    int n = g.size();
    vi cost(n, 1e9); //~INT_MAX/2
    priority_queue<ii, greater<ii>> q;

    q.emplace(0, start);
    cost[start] = 0;
    while (not q.empty())
    {
        int u = q.top().second, w = q.top().first;
        q.pop();

        // we skip all nodes in the q that we have discovered before at a lower cost
        if (cost[u] < w) continue;

        for (auto v : g[u])
        {
            if (cost[v.second] > v.first + w)
            {
                cost[v.second] = v.first + w;
                q.emplace(cost[v.second], v.second);
            }
        }
    }

    return cost[end];
}
```

## 14 ./graphs/dfs/dfsRecursive.cpp

```cpp
#include "../../headers/headers/headers.h"
//Recursive (create visited filled with 1s)
void dfs_r(graph &g, vi &visited, int u)
{
    cout << u << '\n';
    visited[u] = 0;

    for (int v : g[u])
        if (visited[v])
            dfs_r(g, visited, v);
}
```

## 15 ./graphs/dfs/dfsIterative.cpp

```cpp
#include "../../headers/headers/headers.h"
//Iterative
void dfs_i(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    stack<int> s;

    s.emplace(start);
    visited[start] = 0;

    while (not s.empty())
    {
        int u = s.top();
        s.pop();

        cout << u << '\n';

        for (int v : g[u])
            if (visited[v])
            {
                s.emplace(v);
                visited[v] = 0;
            }
    }
}
```

## 16 ./graphs/lca/lca.cpp

```
#include "../../headers/headers/headers.h"
class LcaTree
{
    int n;
    vi parent;
    vi level;
    vi root;
    graph P;
public:
    LcaTree(int n){
        this->n = n;
        parent.assign(n,-1);
        level.assign(n,-1);
        P.assign(n,vi(lg(n)+1,-1));
        root.assign(n,-1);
    }
    void addLeaf(int index, int par){
        parent[index] = par;
        level[index] = level[par] + 1;
        P[index][0] = par;
        root[index] = root[par];
        for(int j=1; (1<<j) < n; ++j){
            if(P[index][j-1] != -1)
                P[index][j] = P[P[index][j-1]][j-1];
        }
    }
    void addRoot(int index){
        parent[index] = index;
        level[index] = 0;
        root[index] = index;
    }
    int lca(int u, int v){
        if(root[u] != root[v] || root[u] == -1)
            return -1;
        if(level[u] < level[v])
            swap(u,v);
        int dist = level[u] - level[v];
        while(dist != 0){
            int raise = lg(dist);
            u = P[u][raise];
            dist -= (1<<raise);
        }
        if(u == v)
            return u;
```

```
        for(int j = lg(n); j>=0; --j){
            if(P[u][j] != -1 && P[u][j] != P[v][j]){
                u=P[u][j];
                v=P[v][j];
            }
        }
        return parent[u];
    }
};
```

## 17 ./graphs/kruskal/kruskal.cpp

```cpp
#include "../../headers/headers/headers.h"
struct edge
{
    int u, v;
    ll w;
    edge(int u, int v, ll w) : u(u), v(v), w(w) {}

    bool operator<(const edge &o) const
    {
        return w < o.w;
    }
};

class Kruskal
{
  private:
    ll sum;
    vi p, rank;

  public:
  //Amount of Nodes n, and unordered vector of Edges E
    Kruskal(int n, vector<edge> E)
    {
        sum = 0;
        p.resize(n);
        rank.assign(n, 0);
        rep(i, n) p[i] = i;
        sort(E.begin(), E.end());
        for (auto &e : E)
            UnionSet(e.u, e.v, e.w);
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j, ll w)
    {
        if (not isSameSet(i, j))
        {
            int x = findSet(i), y = findSet(j);
```

```
            if (rank[x] > rank[y])
                p[y] = x;
            else
                p[x] = y;

            if (rank[x] == rank[y])
                rank[y]++;

            sum += w;
        }
    }
    ll mst_val()
    {
        return sum;
    }
};
```

## 18 ./graphs/unionFind/unionFind.cpp

```cpp
#include "../../headers/headers/headers.h"
class UnionFind
{
  private:
    int numSets;
    vi p, rank, setSize;

  public:
    UnionFind(int n)
    {
        numSets = n;
        rank.assign(n, 0);
        setSize.assign(n, 1);
        p.resize(n);
        rep(i, n) p[i] = i;
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j)
    {
        if (not isSameSet(i, j))
        {
            numSets--;
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
            {
                p[y] = x;
                setSize[x] += setSize[y];
            }
            else
            {
                p[x] = y;
                setSize[y] += setSize[x];
                if (rank[x] == rank[y])
                    rank[y]++;
            }
        }
    }
```

```
    int numSets()
    {
        return numSets;
    }
    int setOfSize(int i)
    {
        return setSize[i];
    }
};
```

## 19 ./graphs/floydWarshall/floydWarshall.cpp

```cpp
#include "../../headers/headers/headers.h"

/*
Floyd Warshall implemenation, note that g is using an adjacency matrix and not an
adjacency list
*/
graph floydWarshall (const graph g)
{
    int n = g.size();
    graph dist(n, vi(n, -1));

    rep(i, n)
        rep(j, n)
            dist[i][j] = g[i][j];

    rep(k, n)
        rep(i, n)
            rep(j, n)
                if (dist[i][k] + dist[k][j] < dist[i][j] &&
                    dist[i][k] != INF                    &&
                    dist[k][j] != INF)
                    dist[i][j] = dist[i][k] + dist[k][j];

   return dist;
}
```

## 20 ./graphs/bfs/bfs.cpp

```cpp
#include "../../headers/headers/headers.h"

void bfs(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    queue<int> q;

    q.emplace(start);
    visited[start] = 0;
    while (not q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v : g[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
            }
        }
    }
}
```

## 21 ./graphs/bellmanFord/bellmanFord.cpp

```cpp
#include "../../headers/headers/headers.h"
bool bellman_ford(wgraph &g, int start)
{
    int n = g.size();
    vector<int> dist(n, 1e9); //~INT_MAX/2
    dist[start] = 0;
    rep(i, n - 1) rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        dist[v] = min(dist[v], dist[u] + w);
    }

    bool hayCicloNegativo = false;
    rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        if (dist[v] > dist[u] + w)
            hayCicloNegativo = true;
    }

    return hayCicloNegativo;
}
```

## 22 ./geometry/vector2D/vector2D.cpp

```cpp
#include "../../headers/headers/headers.h"

const double PI = acos(-1);

struct vector2D
{
    double x, y;

    vector2D &operator+=(const vector2D &o)
    {
        this->x += o.x;
        this->y += o.y;
        return *this;
    }

    vector2D &operator-=(const vector2D &o)
    {
        this->x -= o.x;
        this->y -= o.y;
        return *this;
    }

    vector2D operator+(const vector2D &o)
    {
        return {x + o.x, y + o.y};
    }

    vector2D operator-(const vector2D &o)
    {
        return {x - o.x, y - o.y};
    }

    vector2D operator*(const double &o)
    {
        return {x * o, y * o};
    }

    bool operator==(const vector2D &o)
    {
        return x == o.x and y == o.y;
    }

    double norm2() { return x * x + y * y; }
    double norm() { return sqrt(norm2()); }
```

```
    double dot(const vector2D &o) { return x * o.x + y * o.y; }
    double cross(const vector2D &o) { return x * o.y - y * o.x; }
    double angle()
    {
        double angle = atan2(y, x);
        if (angle < 0)
            angle += 2 * PI;
        return angle;
    }

    vector2D Unit()
    {
        return {x / norm(), y / norm()};
    }
};

/* ========================================================= */
/* Cross Product -> orientation of vector2D with respect to ray */
/* ========================================================= */
// cross product (b - a) x (c - a)
ll cross(vector2D &a, vector2D &b, vector2D &c)
{
    ll dx0 = b.x - a.x, dy0 = b.y - a.y;
    ll dx1 = c.x - a.x, dy1 = c.y - a.y;
    return dx0 * dy1 - dx1 * dy0;
    // return (b - a).cross(c - a); // alternatively, using struct function
}

// calculates the cross product (b - a) x (c - a)
// and returns orientation:
// LEFT (1):      c is to the left of  ray (a -> b)
// RIGHT (-1):    c is to the right of ray (a -> b)
// COLLINEAR (0): c is collinear to    ray (a -> b)
// inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-points/
int orientation(vector2D &a, vector2D &b, vector2D &c)
{
    ll tmp = cross(a, b, c);
    return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
}

/* ========================================================= */
/* Check if a segment is below another segment (wrt a ray) */
/* ========================================================= */
// i.e: check if a segment is intersected by the ray first
// Assumptions:
// 1) for each segment:
```

```cpp
//  p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or COLLINEAR) wrt ray
// 2) segments do not intersect each other
// 3) segments are not collinear to the ray
// 4) the ray intersects all segments
struct Segment
{
    vector2D p1, p2;
};
#define MAXN (int)1e6   //Example
Segment segments[MAXN]; // array of line segments
bool is_si_below_sj(int i, int j)
{ // custom comparator based on cross product
    Segment &si = segments[i];
    Segment &sj = segments[j];
    return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0 : cross(sj.p1, si.p1, si.p2) > 0;
}
// this can be used to keep a set of segments ordered by order of intersection
// by the ray, for example, active segments during a SWEEP LINE
set<int, bool (*)(int, int)> active_segments(is_si_below_sj); // ordered set

/* ======================== */
/* Rectangle Intersection */
/* ======================== */
bool do_rectangles_intersect(vector2D &dl1, vector2D &ur1, vector2D &dl2, vector2D &ur2)
{
    return max(dl1.x, dl2.x) <= min(ur1.x, ur2.x) && max(dl1.y, dl2.y) <= min(ur1.y, ur2.y);
}

/* ========================== */
/* Line Segment Intersection */
/* ========================== */
// returns whether segments p1q1 and p2q2 intersect, inspired by:
// https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
bool do_segments_intersect(vector2D &p1, vector2D &q1, vector2D &p2, vector2D &q2)
{
    int o11 = orientation(p1, q1, p2);
    int o12 = orientation(p1, q1, q2);
    int o21 = orientation(p2, q2, p1);
    int o22 = orientation(p2, q2, q1);
    if (o11 != o12 and o21 != o22) // general case -> non-collinear intersection
        return true;
    if (o11 == o12 and o11 == 0)
    { // particular case -> segments are collinear
        vector2D dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
        vector2D ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
        vector2D dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
```

```
        vector2D ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
        return do_rectangles_intersect(dl1, ur1, dl2, ur2);
    }
    return false;
}

/* ================== */
/* Circle Intersection */
/* ================== */
struct Circle
{
    double x, y, r;
};
bool is_fully_outside(double r1, double r2, double d_sqr)
{
    double tmp = r1 + r2;
    return d_sqr > tmp * tmp;
}
bool is_fully_inside(double r1, double r2, double d_sqr)
{
    if (r1 > r2)
        return false;
    double tmp = r2 - r1;
    return d_sqr < tmp * tmp;
}
bool do_circles_intersect(Circle &c1, Circle &c2)
{
    double dx = c1.x - c2.x;
    double dy = c1.y - c2.y;
    double d_sqr = dx * dx + dy * dy;
    if (is_fully_inside(c1.r, c2.r, d_sqr))
        return false;
    if (is_fully_inside(c2.r, c1.r, d_sqr))
        return false;
    if (is_fully_outside(c1.r, c2.r, d_sqr))
        return false;
    return true;
}

/* ===================== */
/* vector2D - Line distance */
/* ===================== */
// get distance between p and projection of p on line <- a - b ->
double point_line_dist(vector2D &p, vector2D &a, vector2D &b)
{
    vector2D d = b - a;
```

```
    double t = d.dot(p - a) / d.norm2();
    return (a + d * t - p).norm();
}

/* ======================== */
/* vector2D - Segment distance */
/* ======================== */
// get distance between p and truncated projection of p on segment a -> b
double point_segment_dist(vector2D &p, vector2D &a, vector2D &b)
{
    if (a == b)
        return (p - a).norm(); // segment is a single vector2D
    vector2D d = b - a;        // direction
    double t = d.dot(p - a) / d.norm2();
    if (t <= 0)
        return (p - a).norm(); // truncate left
    if (t >= 1)
        return (p - b).norm(); // truncate right
    return (a + d * t - p).norm();
}

/* ==================================== */
/* Straight Line Hashing (integer coords) */
/* ==================================== */
// task: given 2 points p1, p2 with integer coordinates, output a unique
// representation {a,b,c} such that a*x + b*y + c = 0 is the equation
// of the straight line defined by p1, p2. This representation must be
// unique for each straight line, no matter which p1 and p2 are sampled.
struct Line
{
    int a, b, c;
};
int gcd(int a, int b)
{ // greatest common divisor
    a = abs(a);
    b = abs(b);
    while (b)
    {
        int c = a;
        a = b;
        b = c % b;
    }
    return a;
}
Line getLine(vector2D p1, vector2D p2)
{
```

```
    int a = p1.y - p2.y;
    int b = p2.x - p1.x;
    int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
    int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
    int f = gcd(a, gcd(b, c)) * sgn;
    a /= f;
    b /= f;
    c /= f;
    return {a, b, c};
}
```

## 23 ./geometry/greenTheorem/areasGreen.cpp

```cpp
#include "../../headers/headers/headers.h"

// O(1)
// Circle Arc
double arc(double theta, double phi)
{
}

// O(1)
// Line
double line(double x1, double y1, double x2, double y2)
{
}
```