# The AC is a lie - ICPC Library

# Contents


**1 String Algorithms** 1
1.1 String Alignment . . . 1
1.2 KMP . . . 1
1.3 Trie . . . 1
1.4 Aho-Corasick . . . 2

**2 Data Structures** 2
2.1 BIT - Binary Indexed Tree . . . 2
2.2 Iterative Segment Tree . . . 2
2.3 Iterative Segment Tree with Interval Updates . . . 2
2.4 Recursive Segment Tree . . . 2
2.5 Segment Tree with Lazy Propagation . . . 3
2.6 Color Updates Structure . . . 3
2.7 Policy Based Structures . . . 3
2.8 Heavy Light Decomposition . . . 3
2.9 Centroid Decomposition . . . 4
2.10 Sparse Table . . . 5

**3 Graph Algorithms** 5
3.1 Dinic Max Flow . . . 5
3.2 Articulation Points/Bridges/Biconnected Components . . . 5
3.3 SCC - Strongly Connected Components / 2SAT . . . 6
3.4 LCA - Lowest Common Ancestor . . . 6
3.5 Sack . . . 6
3.6 Min Cost Max Flow . . . 7

**4 Math** 7
4.1 Discrete Logarithm . . . 7
4.2 GCD - Greatest Common Divisor . . . 7
4.3 Extended Euclides . . . 8
4.4 Fast Exponentiation . . . 8
4.5 Matrix Fast Exponentiation . . . 8
4.6 FFT - Fast Fourier Transform . . . 8
4.7 NTT - Number Theoretic Transform . . . 8
4.8 Miller and Rho . . . 9

**5 Geometry** 10
5.1 Geometry . . . 10
5.2 Convex Hull . . . 11
5.3 ClosestPair . . . 11
5.4 Intersection Points . . . 11

**6 Miscellaneous** 11
6.1 LIS - Longest Increasing Subsequence . . . 11
6.2 Binary Search . . . 12
6.3 Ternary Search . . . 12

**7 Teoremas e formulas uteis** 12
7.1 Grafos . . . 12
7.2 Math . . . 12
7.3 Geometry . . . 12


# 1 String Algorithms

## 1.1 String Alignment

```
int pd[ms][ms];

int edit_distance(string &a, string &b) {
    int n = a.size(), m = b.size();
    for(int i = 0; i <= n; i++) pd[i][0] = i;
    for(int j = 0; j <= m; j++) pd[0][j] = j;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            int del = pd[i][j-1] + 1;
            int ins = pd[i-1][j] + 1;
            int mod = pd[i-1][j-1] + (a[i-1] != b[j-1]);
            pd[i][j] = min(del, min(ins, mod));
        }
    }
    return pd[n][m];
}
```

## 1.2 KMP

```
string p, t;
int b[ms], n, m;

void kmpPreprocess() {
    int i = 0, j = -1;
    b[0] = -1;
    while(i < m) {
        while(j >= 0 && p[i] != p[j]) j = b[j];
        b[++i] = ++j;
    }
}

void kmpSearch() {
    int i = 0, j = 0, ans = 0;
    while(i < n) {
        while(j >= 0 && t[i] != p[j]) j = b[j];
        i++; j++;
        if(j == m) {
            //ocorrencia aqui comecando em i - j
            ans++;
            j = borda[j];
        }
    }
}
```

## 1.3 Trie

```
int trie[ms][sigma], terminal[ms], z;

void init() {
    memset(trie[0], -1, sizeof trie[0]);
    z = 1;
}

int get_id(char c) {
    return c - 'a';
}

void insert(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            memset(trie[z], -1, sizeof trie[z]);
            trie[cur][id] = z++;
        }
        cur = trie[cur][id];
    }
    terminal[cur]++;
}

int count(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
```

```
            return false;
        }
        cur = trie[cur][id];
    }
    return terminal[cur];
}
```

## 1.4 Aho-Corasick

```
// Construa a Trie do seu dicionario com o codigo acima

int fail[ms], q[ms], front, rear;

void buildFailure() {
    front = 0; rear = 0; q[rear++] = 0;
    while(front < rear) {
        int node = q[front++];
        for(int pos = 0; pos < sigma; pos++) {
            int &v = trie[node][pos];
            int f = node == 0 ? 0 : trie[fail[node]][pos];
            if(v == -1) {
                v = f;
            } else {
                fail[v] = f;
                q.push(v);
                // juntar as informacoes da borda para o V ja q um match em V implica um
                      match na borda
                terminal[v] += terminal[f];
            }
        }
    }
}

int search(string &txt) {
    int node = 0;
    int ans = 0;
    for(int i = 0; i < txt.length(); i++) {
        int pos = get_id(txt[i]);
        node = trie[node][pos];
        // processar informacoes no no atual
       ans += terminal[node];
    }
    return ans;
}
```

# 2 Data Structures

## 2.1 BIT - Binary Indexed Tree

```
int arr[ms], bit[ms], n;

void update(int v, int idx) {
    while(idx <= n) {
        bit[idx] += v;
        idx += idx & -idx;
    }
}

int query(int idx) {
    int r = 0;
    while(idx > 0) {
        r += bit[idx];
        idx -= idx & -idx;
    }
    return r;
}
```

## 2.2 Iterative Segment Tree

```
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int p, int value) { // set value at position p
    for(t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) {
    int res = 0;
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) res += t[l++];
        if(r&1) res += t[--r];
    }
    return res;
}

// If is non-commutative
S query(int l, int r) {
  S resl, resr;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) resl = combine(resl, t[l++]);
    if (r&1) resr = combine(t[--r], resr);
  }
  return combine(resl, resr);
}
```

## 2.3 Iterative Segment Tree with Interval Updates

```
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int l, int r, int value) {
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) t[l++] += value;
        if(r&1) t[--r] += value;
    }
}

int query(int p) {
    int res = 0;
    for(p += n; p > 0; p >>= 1) res += t[p];
    return res;
}

void push() { // push modifications to leafs
    for(int i = 1; i < n; i++) {
        t[i<<1] += t[i];
        t[i<<1|1] += t[i];
        t[i] = 0;
    }
}
```

## 2.4 Recursive Segment Tree

```
int arr[4 * ms], seg[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
```

```
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}

void update(int V, int I, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l > I || r < I) return;
    if(l == r) {
        arr[I] = V;
        seg[idx] = V;
        return;
    }
    update(V, I, left, l, mid); update(V, I, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

## 2.5 Segment Tree with Lazy Propagation

```
int arr[4 * ms],  seg[4 * ms], lazy[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

void propagate(int idx, int l, int r, int left, int right) {
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1);
        if(l < r) {
            lazy[left] += lazy[idx];
            lazy[right] += lazy[idx];
        }
        lazy[idx] = 0;
    }
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    propagate(idx, l, r, left, right);
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}

void update(int V, int L, int R, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    propagate(idx, l, r, left, right);
    if(l > R || r < L) return;
    if(L <= l && r <= R) {
        lazy[idx] += V;
        propagate(idx, l, r, left, right);
        return;
    }
    update(V, L, R, left, l, mid); update(V, L, R, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

## 2.6 Color Updates Structure

```
struct range {
        int l, r;
        int v;

        range(int l = 0, int r = 0, int v = 0) : l(l), r(r), v(v) {}

        bool operator < (const range &a) const {
                return l < a.l;
        }
};

set<range> ranges;

vector<range> update(int l, int r, int v) { // [l, r)
        vector<range> ans;
        if(l >= r) return ans;
        auto it = ranges.lower_bound(l);
        if(it != ranges.begin()) {
                it--;
                if(it->r > l) {
                        auto cur = *it;
                        ranges.erase(it);
                        ranges.insert(range(cur.l, l, cur.v));
                        ranges.insert(range(l, cur.r, cur.v));
                }
        }
        it = ranges.lower_bound(r);
        if(it != ranges.begin()) {
                it--;
                if(it->r > r) {
                        auto cur = *it;
                        ranges.erase(it);
                        ranges.insert(range(cur.l, r, cur.v));
                        ranges.insert(range(r, cur.r, cur.v));
                }
        }
        for(it = ranges.lower_bound(l); it != ranges.end() && it->l < r; it++) {
                ans.push_back(*it);
        }
        ranges.erase(ranges.lower_bound(l), ranges.lower_bound(r));
        ranges.insert(range(l, r, v));
        return ans;
}

int query(int v) { // Substituir -1 por flag para quando nao houver resposta
        auto it = ranges.upper_bound(v);
        if(it == ranges.begin()) {
                return -1;
        }
        it--;
        return it->r >= v ? it->v : -1;
}
```

## 2.7 Policy Based Structures

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including tree_order_statistics_node_update

using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

ordered_set X;
X.insert(1);
X.find_by_order(0);
X.order_of_key(-5);
end(X), begin(X);
```

## 2.8 Heavy Light Decomposition

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

```
#define all(foo) foo.begin(), foo.end()
using namespace std;
const int N = 112345, inf = 0x3f3f3f3f;

int k, adj[N], ant[2*N], to[2*N];

void add(int a, int b, bool f = 1){
	ant[k] = adj[a], adj[a] = k, to[k] = b;
	k++;
	if(f) add(b, a, 0);
}

int par[N], h[N], big[N], sz[N];
void dfs(int v, int p, int hght){
	sz[v] = 1, par[v] = p, h[v] = hght, big[v] = -1;
	for(int i = adj[v]; ~i; i = ant[i]){
		if(to[i] != p){
			dfs(to[i], v, hght+1);
			sz[v] += sz[to[i]];
			if(big[v] == -1 || sz[big[v]] < sz[to[i]]) big[v] = to[i];
		}
	}
}

int chainNo, chain[N], ind[N], chainSz[N], head[N];
vector<int> tree[N];
vector<int> st[N];

void upd(int p, int value, vector<int> &tree){
	int n = tree.size()>>1;
	for(tree[p += n] = value; p > 1; p >>= 1) tree[p>>1] = min(tree[p], tree[p^1]);
}

int rmq(int l, int r, vector<int> &tree){
	int res = inf;
	int n = tree.size()>>1;
	for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
		if(l&1) res = min(res, tree[l++]);
		if(r&1) res = min(res, tree[--r]);
	}
	return res;
}

void HLD(int v, int p, int cIn){
	if(cIn == 0) head[chainNo] = v;
	ind[v] = cIn;
	chain[v] = chainNo;
	st[chainNo].push_back(v);
	if(~big[v]){
		HLD(big[v], v, cIn+1);
	}else{
		int n = chainSz[chainNo] = st[chainNo].size();
		tree[chainNo].resize(2*n);
		fill(all(tree[chainNo]), inf);
		chainNo++;
	}
	for(int i = adj[v]; ~i; i = ant[i]){
		if(to[i] != p && to[i] != big[v]){
			HLD(to[i], v, 0);
		}
	}
}

int up(int v){
	return (head[chain[v]] != v) ? head[chain[v]] : (par[v] != -1 ? par[v] : v);
}

int LCA(int a, int b){
	while(chain[a] != chain[b]){
		if(par[a] == -1 || h[up(a)] < h[up(b)]) swap(a, b);
		a = up(a);
	}
	return h[a] < h[b] ? a : b;
}

int queryUp(int a, int b){
	int ans = -1, curr;
	while(chain[a] != chain[b]){
		curr = rmq(0, ind[a]+1, tree[chain[a]]);
		if(curr != inf) ans = st[chain[a]][curr];
		a = par[head[chain[a]]];
	}
	curr = rmq(ind[b], ind[a]+1, tree[chain[a]]);
	if(curr != inf) ans = st[chain[a]][curr];
	return ans;
}

int main(){
	int n, q;
	scanf("%d %d", &n, &q);
	memset(adj, -1, sizeof adj);
	for(int i = 0; i < n-1; i++){
		int a, b;
		scanf("%d %d", &a, &b);
		add(a, b);
	}
	dfs(1, -1, 0);
	HLD(1, -1, 0);
	for(int i = 0; i < q; i++){
		int o, v;
		scanf("%d %d", &o, &v);
		if(o){
			printf("%d\n", queryUp(v, 1));
		}else{
			int ans = rmq(ind[v], ind[v]+1, tree[chain[v]]);
			upd(ind[v], (ans == inf) ? ind[v] : inf, tree[chain[v]]);
		}
	}
}
```

## 2.9 Centroid Decomposition

```
//Centroid decomposition1

void dfsSize(int v, int pa) {
	sz[v] = 1;
	for(int u : adj[v]) {
		if (u == pa || rem[u]) continue;
		dfsSize(u, v);
		sz[v] += sz[u];
	}
}

int getCentroid(int v, int pa, int tam) {
	for(int u : adj[v]) {
		if (u == pa || rem[u]) continue;
		if (2 * sz[u] > tam) return getCentroid(u, v, tam);
	}
	return v;
}

void decompose(int v, int pa = -1) {
	//cout << v << ' ' << pa << '\n';
	dfsSize(v, pa);
	int c = getCentroid(v, pa, sz[v]);
	//cout << c << '\n';
	par[c] = pa;
	rem[c] = 1;
	for(int u : adj[c]) {
		if (!rem[u] && u != pa) decompose(u, c);
	}
	adj[c].clear();
}

//Centroid decomposition2

void dfsSize(int v, int par) {
	sz[v] = 1;
	for(int u : adj[v]) {
		if (u == par || removed[u]) continue;
		dfsSize(u, v);
		sz[v] += sz[u];
	}
}
```

```
int getCentroid(int v, int par, int tam) {
	for(int u : adj[v]) {
		if (u == par || removed[u]) continue;
		if (2 * sz[u] > tam) return getCentroid(u, v, tam);
	}
	return v;
}

void setDis(int v, int par, int nv, int d) {
	dis[v][nv] = d;
	for(int u : adj[v]) {
		if (u == par || removed[u]) continue;
		setDis(u, v, nv, d + 1);
	}
}

void decompose(int v, int par, int nv) {
	dfsSize(v, par);
	int c = getCentroid(v, par, sz[v]);
	ct[c] = par;
	removed[c] = 1;
	setDis(c, par, nv, 0);
	for(int u : adj[c]) {
		if (!removed[u]) {
			decompose(u, c, nv + 1);
		}
	}
}
```

## 2.10 Sparse Table

```
template<class Info_t>
class SparseTable {
private:
    vector<int> log2;
    vector<vector<Info_t>> table;

    Info_t merge(Info_t &a, Info_t &b) {

    }

public:
    SparseTable(int n, vector<Info_t> v) {
        log2.resize(n + 1);
        log2[1] = 0;
        for (int i = 2; i <= n; i++) {
            log2[i] = log2[i >> 1] + 1;
        }
        table.resize(n + 1);
        for (int i = 0; i < n; i++) {
            table[i].resize(log2[n] + 1);
        }
        for (int i = 0; i < n; i++) {
            table[i][0] = v[i];
        }
        for (int i = 0; i < log2[n]; i++) {
            for (int j = 0; j < n; j++) {
                if (j + (1 << i) >= n) break;
                table[j][i + 1] = merge(table[j][i], table[j + (1 << i)][i]);
            }
        }
    }

    int get(int l, int r) {
        int k = log2[r - l + 1];
        return merge(table[l][k], table[r - (1 << k) + 1][k]);
    }
};
```

# 3 Graph Algorithms

## 3.1 Dinic Max Flow

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], wt[me], z, n;
int copy_adj[ms], fila[ms], level[ms];

void clear() {
    memset(adj, -1, sizeof adj);
    z = 0;
}

int add(int u, int v, int k) {
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = k;
    adj[u] = z++;
    swap(u, v);
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = 0;
    adj[u] = z++;
}

int bfs(int source, int sink) {
	memset(level, -1, sizeof level);
	level[source] = 0;
	int front = 0, size = 0, v;
	fila[size++] = source;
	while(front < size) {
		v = fila[front++];
		for(int i = adj[v]; i != -1; i = ant[i]) {
			if(wt[i] && level[to[i]] == -1) {
				level[to[i]] = level[v] + 1;
				fila[size++] = to[i];
			}
		}
	}
	return level[sink] != -1;
}

int dfs(int v, int sink, int flow) {
	if(v == sink) return flow;
	int f;
	for(int &i = copy_adj[v]; i != -1; i = ant[i]) {
		if(wt[i] && level[to[i]] == level[v] + 1 &&
			(f = dfs(to[i], sink, min(flow, wt[i])))) {
				wt[i] -= f;
				wt[i ^ 1] += f;
				return f;
		}
	}
	return 0;
}

int maxflow(int source, int sink) {
	int ret = 0, flow;
	while(bfs(source, sink)) {
		memcpy(copy_adj, adj, sizeof adj);
		while((flow = dfs(source, sink, 1 << 30))) {
			ret += flow;
		}
	}
	return ret;
}
```

## 3.2 Articulation Points/Bridges/Biconnected Components

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], z, n;
int idx[ms], bc[me], ind, nbc, child, st[me], top;

void generateBc(int edge) {
    while(st[--top] != edge) {
        bc[st[top]] = nbc;
    }
    bc[edge] = nbc++;
}

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            st[top++] = i;
            int temp = dfs(to[i], v);
            if(par == -1 && child > 1 || ~par && temp >= idx[v]) generateBc(i);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par && idx[to[i]] < low) {
            low = idx[to[i]];
            st[top++] = i;
        }
    }
    return low;
}

void biconnected() {
    ind = 0;
    nbc = 0;
    top = -1;
    memset(idx, -1, sizeof idx);
    memset(art, 0, sizeof art);
    memset(bridge, 0, sizeof bridge);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
    }
}
```

## 3.3 SCC - Strongly Connected Components / 2SAT

```
vector<int> g[ms];
int idx[ms], low[ms], z, comp[ms], ncomp, st[ms], top;

int dfs(int u) {
    if(~idx[u]) return idx[u] ? idx[u] : z;
    low[u] = idx[u] = z++;
    st.push(u);
    for(int v : g[u]) {
        low[u] = min(low[u], dfs(v));
    }
    if(low[u] == idx[u]) {
        idx[st.top()] = 0;
        st.pop();
        while(st.top() != u) {
            int v = st.top();
            st.pop();
            idx[v] = 0;
            low[v] = low[u];
            comp[v] = ncomp;
        }
        comp[u] = ncomp++;
    }
    return low[u];
}

bool solveSat() {
    memset(idx, -1, sizeof idx);
    ind = 1; top = -1;
    for(int i = 0; i < n; i++) dfs(i);
    for(int i = 0; i < n; i++) if(comp[i] == comp[i^1]) return false;
    return true;
}

// Operacoes comuns de 2-sat
// ~v = "nao v"
#define trad(v) (v<0?((~v)*2)^1:v*2)
void addImp(int a, int b) { g[trad(a)].push(trad(b)); }
void addOr(int a, int b) { addImp(~a, b); addImp(~b, a); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// valoracao: value[v] = comp[trad(v)] < comp[trad(~v)]
```

## 3.4 LCA - Lowest Common Ancestor

```
int par[ms][mlg], lvl[ms];

void dfs(int v, int p, int l = 0) {
    lvl[v] = l;
    par[v][0] = p;
    for(int i = adj[v]; i > - 1; i = ant[i]) {
        if(to[i] != p) dfs(to[i], v, l + 1);
    }
}

void processAncestors(int root = 0) {
    dfs(root, root);
    for(int k = 1; k <= mlg; k++) {
        for(int i = 0; i < n; i++) {
            par[i][k] = par[par[i][k-1]][k-1];
        }
    }
}

int lca(int a, int b) {
    if(lvl[b] > lvl[a]) swap(a, b);
    for(int i = mlg; i >= 0; i--) {
        if(lvl[a] - (1 << i) >= lvl[b]) a = par[a][i];
    }
    if(a == b) return a;
    for(int i = mlg; i >= 0; i--) {
        if(par[a][i] != par[b][i]) a = par[a][i], b = par[b][i];
    }
    return par[a][0];
}
```

## 3.5 Sack

```
void solve(int a, int p, bool f){
        int big = -1;
        for(auto &b : adj[a]){
                if(b != p && (big == -1 || en[b]-st[b] > en[big]-st[big])){
                        big = b;
                }
        }
        for(auto &b : adj[a]){
                if(b == p || b == big) continue;
                solve(b, a, 0);
        }
        if(~big) solve(big, a, 1);
        add(cnt[v[a]], -1);
        cnt[v[a]]++;
        add(cnt[v[a]], +1);
        for(auto &b : adj[a]){
                if(b == p || b == big) continue;
                for(int i = st[b]; i < en[b]; i++){
                        add(cnt[ett[i]], -1);
                        cnt[ett[i]]++;
                        add(cnt[ett[i]], +1);
                }
        }
```

```
        for(auto &q : Q[a]){
                ans[q.first] = query(mx-1)-query(q.second-1);
        }
        if(!f){
                for(int i = st[a]; i < en[a]; i++){
                        add(cnt[ett[i]], -1);
                        cnt[ett[i]]--;
                        add(cnt[ett[i]], +1);
                }
        }
}
```

## 3.6 Min Cost Max Flow

```
template <class flow_t, class cost_t>
class MinCostMaxFlow {
private:
    typedef pair<cost_t, int> ii;

    struct Edge {
        int to;
        flow_t cap;
        cost_t cost;
        Edge(int to, flow_t cap, cost_t cost) : to(to), cap(cap), cost(cost) {}
    };

    int n;
    vector<vector<int>> adj;
    vector<Edge> edges;
    vector<cost_t> dis;
    vector<int> prev, id_prev;
        vector<int> q;
        vector<bool> inq;

    pair<flow_t, cost_t> spfa(int src, int sink) {
        fill(dis.begin(), dis.end(), int(1e9)); //cost_t inf
        fill(prev.begin(), prev.end(), -1);
        fill(inq.begin(), inq.end(), false);
        q.clear();
        q.push_back(src);
        inq[src] = true;
        dis[src] = 0;
        for(int on = 0; on < (int) q.size(); on++) {
                int cur = q[on];
                inq[cur] = false;
                for(auto id : adj[cur]) {
                        if (edges[id].cap == 0) continue;
                        int to = edges[id].to;
                        if (dis[to] > dis[cur] + edges[id].cost) {
                                prev[to] = cur;
                                id_prev[to] = id;
                                dis[to] = dis[cur] + edges[id].cost;
                                if (!inq[to]) {
                                        q.push_back(to);
                                        inq[to] = true;
                                }
                        }
                }
        }
        flow_t mn = flow_t(1e9);
        for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
            int id = id_prev[cur];
            mn = min(mn, edges[id].cap);
        }
        if (mn == flow_t(1e9) || mn == 0) return make_pair(0, 0);
        pair<flow_t, cost_t> ans(mn, 0);
        for(int cur = sink; prev[cur] != -1; cur = prev[cur]) {
            int id = id_prev[cur];
            edges[id].cap -= mn;
            edges[id ^ 1].cap += mn;
            ans.second += mn * edges[id].cost;
        }
        return ans;
    }
public:
    MinCostMaxFlow(int a = 0) {
        n = a;
        adj.resize(n + 2);
        edges.clear();
        dis.resize(n + 2);
        prev.resize(n + 2);
        id_prev.resize(n + 2);
        inq.resize(n + 2);
    }
    void init(int a) {
        n = a;
        adj.resize(n + 2);
        edges.clear();
        dis.resize(n + 2);
        prev.resize(n + 2);
        id_prev.resize(n + 2);
        inq.resize(n + 2);
    }
    void add(int from, int to, flow_t cap, cost_t cost) {
        adj[from].push_back(int(edges.size()));
                edges.push_back(Edge(to, cap, cost));
                adj[to].push_back(int(edges.size()));
                edges.push_back(Edge(from, 0, -cost));
    }
    pair<flow_t, cost_t> maxflow(int src, int sink) {
        pair<flow_t, cost_t> ans(0, 0), got;
        while((got = spfa(src, sink)).first > 0) {
            ans.first += got.first;
            ans.second += got.second;
        }
        return ans;
    }
};
```

# 4 Math

## 4.1 Discrete Logarithm

```
ll discreteLog(ll a, ll b, ll m) {
        // a^ans == b mod m
        // ou -1 se nao existir
        ll cur = a, on = 1;
        for(int i = 0; i < 100; i++) {
                cur = cur * a % m;
        }
        while(on * on <= m) {
                cur = cur * a % m;
                on++;
        }
        map<ll, ll> position;
        for(ll i = 0, x = 1; i * i <= m; i++) {
                position[x] = i * on;
                x = x * cur % m;
        }
        for(ll i = 0; i <= on + 20; i++) {
                if(position.count(b)) {
                        return position[b] - i;
                }
                b = b * a % m;
        }
        return -1;
}
```

## 4.2 GCD - Greatest Common Divisor

```
ll gcd(ll a, ll b) {
        while(b) a %= b, swap(a, b);
        return a;
}
```

## 4.3 Extended Euclides

```
// euclides estendido: acha u e v da equacao:
// u * x + v * y = gcd(x, y);
// u eh inverso modular de x no modulo y
// v eh inverso modular de y no modulo x

pair<ll, ll> euclides(ll a, ll b) {
        ll u = 0, oldu = 1, v = 1, oldv = 0;
        while(b) {
                ll q = a / b;
                oldv = oldv - v * q;
                oldu = oldu - u * q;
                a = a - b * q;
                swap(a, b);
                swap(u, oldu);
                swap(v, oldv);
        }
        return make_pair(oldu, oldv);
}
```

## 4.4 Fast Exponentiation

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

ll fExp(ll a, ll b) {
        ll ans = 1;
        while(b) {
                if(b & 1) ans = ans * a % mod;
                a = a * a % mod;
                b >>= 1;
        }
        return ans;
}
```

## 4.5 Matrix Fast Exponentiation

```
const ll mod = 1e9+7;
const int m = 2; // size of matrix

struct Matrix {
        ll mat[m][m];
        Matrix operator * (const Matrix &p) {
                Matrix ans;
                for(int i = 0; i < m; i++)
                        for(int j = 0; j < m; j++)
                                for(int k = ans.mat[i][j] = 0; k < m; k++)
                                        ans.mat[i][j] = (ans.mat[i][j] + mat[i][k] * p.mat[k
                                                ][j]) % mod;
                return ans;
        }
};

Matrix fExp(Matrix a, ll b) {
        Matrix ans;
        for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
                ans.mat[i][j] = i == j;
        while(b) {
                if(b & 1) ans = ans * a;
                a = a * a;
                b >>= 1;
        }
        return ans;
}
```

## 4.6 FFT - Fast Fourier Transform

```
typedef complex<double> Complex;
typedef long double ld;
typedef long long ll;

const int ms = maiortamanhoderesposta * 2;
const ld pi = acosl(-1.0);

int rbit[1 << 23];
Complex a[ms], b[ms];

void calcReversedBits(int n) {
    int lg = 0;
    while(1 << (lg + 1) < n) {
        lg++;
    }
    for(int i = 1; i < n; i++) {
        rbit[i] = (rbit[i >> 1] >> 1) | ((i & 1) << lg);
    }
}

void fft(Complex a[], int n, bool inv = false) {
    for(int i = 0; i < n; i++) {
        if(rbit[i] > i) swap(a[i], a[rbit[i]]);
    }
    double ang = inv ? -pi : pi;
    for(int m = 1; m < n; m += m) {
        Complex d(cosl(ang/m), sinl(ang/m));
        for(int i = 0; i < n; i += m+m) {
            Complex cur = 1;
            for(int j = 0; j < m; j++) {
                Complex u = a[i + j], v = a[i + j + m] * cur;
                a[i + j] = u + v;
                a[i + j + m] = u - v;
                cur *= d;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; i++) a[i] /= n;
    }
}

void multiply(ll x[], ll y[], ll ans[], int nx, int ny, int &n) {
    n = 1;
    while(n < nx+ny) n <<= 1;
    calcReversedBits(n);
    for(int i = 0; i < n; i++) {
        a[i] = Complex(x[i]);
        b[i] = Complex(y[i]);
    }
    fft(a, n); fft(b, n);
    for(int i = 0; i < n; i++) {
        a[i] = a[i] * b[i];
    }
    fft(a, n, true);
    for(int i = 0; i < n; i++) {
        ans[i] = ll(a[i].real() + 0.5);
    }
    n = nx + ny;
}
```

## 4.7 NTT - Number Theoretic Transform

```
long long int mod = (119ll << 23) + 1, c_root = 3;

namespace NTT {
        typedef long long int ll;

        ll fexp(ll base, ll e) {
                ll ans = 1;
                while(e > 0) {
                        if (e & 1) ans = ans * base % mod;
                        base = base * base % mod;
                        e >>= 1;
                }
```

```
        return ans;
    }

    ll inv_mod(ll base) {
        return fexp(base, mod - 2);
    }

    void ntt(vector<ll>& a, bool inv) {
        int n = (int) a.size();
        if (n == 1) return;

        for(int i = 0, j = 0; i < n; i++) {
            if (i > j) {
                swap(a[i], a[j]);
            }
            for(int l = n / 2; (j ^= l) < l; l >>= 1);
        }

        for(int sz = 1; sz < n; sz <<= 1) {
            ll delta = fexp(c_root, (mod - 1) / (2 * sz)); //delta = w_2sz
            if (inv) {
                delta = inv_mod(delta);
            }
            for(int i = 0; i < n; i += 2 * sz) {
                ll w = 1;
                for(int j = 0; j < sz; j++) {
                    ll u = a[i + j], v = w * a[i + j + sz] % mod;
                    a[i + j] = (u + v + mod) % mod;
                    a[i + j] = (a[i + j] + mod) % mod;
                    a[i + j + sz] = (u - v + mod) % mod;
                    a[i + j + sz] = (a[i + j + sz] + mod) % mod;
                    w = w * delta % mod;
                }
            }
        }
        if (inv) {
            ll inv_n = inv_mod(n);
            for(int i = 0; i < n; i++) {
                a[i] = a[i] * inv_n % mod;
            }
        }
        for(int i = 0; i < n; i++) {
            a[i] %= mod;
            a[i] = (a[i] + mod) % mod;
        }
    }

    void multiply(vector<ll> &a, vector<ll> &b, vector<ll> &ans) {
        int lim = (int) max(a.size(), b.size());
        int n = 1;
        while(n < lim) n <<= 1;
        n <<= 1;
        a.resize(n);
        b.resize(n);
        ans.resize(n);
        ntt(a, false);
        ntt(b, false);
        for(int i = 0; i < n; i++) {
            ans[i] = a[i] * b[i] % mod;
        }
        ntt(ans, true);
    }
};
```

## 4.8 Miller and Rho

```
typedef long long int ll;

bool overflow(ll a, ll b) {
    return b && (a >= (1ll << 62) / b);
}

ll add(ll a, ll b, ll md) {
    return (a + b) % md;
}

ll mul(ll a, ll b, ll md) {
    if (!overflow(a, b)) return (a * b) % md;
    ll ans = 0;
    while(b) {
        if (b & 1) ans = add(ans, a, md);
        a = add(a, a, md);
        b >>= 1;
    }
    return ans;
}

ll fexp(ll a, ll e, ll md) {
    ll ans = 1;
    while(e) {
        if (e & 1) ans = mul(ans, a, md);
        a = mul(a, a, md);
        e >>= 1;
    }
    return ans;
}

ll my_rand() {
    ll ans = rand();
    ans = (ans << 31) | rand();
    return ans;
}

ll gcd(ll a, ll b) {
    while(b) {
        ll t = a % b;
        a = b;
        b = t;
    }
    return a;
}

bool miller(ll p, int iteracao) {
    if(p < 2) return 0;
    if(p % 2 == 0) return (p == 2);
    ll s = p - 1;
    while(s % 2 == 0) s >>= 1;
    for(int i = 0; i < iteracao; i++) {
        ll a = rand() % (p - 1) + 1, temp = s;
        ll mod = fexp(a, temp, p);
        while(temp != p - 1 && mod != 1 && mod != p - 1) {
            mod = mul(mod, mod, p);
            temp <<= 1;
        }
        if(mod != p - 1 && temp % 2 == 0) return 0;
    }
    return 1;
}

ll rho(ll n) {
    if (n == 1 || miller(n, 10)) return n;
    if (n % 2 == 0) return 2;
    while(1) {
        ll x = my_rand() % (n - 2) + 2, y = x;
        ll c = 0, cur = 1;
        while(c == 0) {
            c = my_rand() % (n - 2) + 1;
        }
        while(cur == 1) {
            x = add(mul(x, x, n), c, n);
            y = add(mul(y, y, n), c, n);
            y = add(mul(y, y, n), c, n);
            cur = gcd((x >= y ? x - y : y - x), n);
        }
        if (cur != n) return cur;
    }
}
```

# 5 Geometry

## 5.1 Geometry

```
const double inf = 1e100, eps = 1e-9;

struct PT {
        double x, y;
        PT(double x = 0, double y = 0) : x(x), y(y) {}
        PT operator + (const PT &p) { return PT(x + p.x, y + p.y); }
        PT operator - (const PT &p) { return PT(x - p.x, y - p.y); }
        PT operator * (double c) { return PT(x * c, y * c); }
        PT operator / (double c) { return PT(x / c, y / c); }
        bool operator <(const PT &p) const {
                if(fabs(x - p.x) >= eps) return x < p.x;
                return fabs(y - p.y) >= eps && y < p.y;
        }
        bool operator ==(const PT &p) const {
                return fabs(x - p.x) < eps && fabs(y - p.y) < eps;
        }
};

double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double dist(PT p, PT q) {return hypot(p.x-q.x, p.y-q.y); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }

// Rotaciona o ponto CCW ou CW ao redor da origem
PT rotateCCW90(PT p) { return PT(-p.y, p.x); }
PT rotateCW90(PT p) { return PT(p.y, -p.x); }
PT rotateCCW(PT p, double d) {
    return PT(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}

// Projeta ponto c na linha a - b assumindo a != b
PT projectPointLine(PT a, PT b, PT c) {
    return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}

// Projeta ponto c no segmento a - b
PT projectPointSegment(PT a, PT b, PT c) {
    double r = dot(b - a, b - a);
    if(abs(r) < eps) return a;
    r = dot(c - a, b - a) / r;
    if(r < 0) return a;
    if(r > 1) return b;
    return a + (b - a) * r;
}
// Calcula distancia entre o ponto c e o segmento a - b
double distancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, projectPointSegment(a, b, c)));
}

// Calcula distancia entre o ponto (x, y, z) e o plano ax + by + cz = d
double distancePointPlane(double x, double y, double z, double a, double b, double c, double
    d) {
    return abs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool linesParallel(PT a, PT b, PT c, PT d) {
    return abs(cross(b - a, c - d)) < eps;
}

bool linesCollinear(PT a, PT b, PT c, PT d) {
    return linesParallel(a, b, c, d) && abs(cross(a - b, a - c)) < eps && abs(cross(c - d, c
        - a)) < eps;
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect(PT a, PT b, PT c, PT d) {
    if(linesCollinear(a, b, c, d)) {
        if(dist2(a, c) < eps || dist2(a, d) < eps || dist2(b, c) < eps || dist2(b, d) < eps)
            return true;
        if(dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c - b, d - b) > 0) return
            false;
        return true;
    }
    if(cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
    if(cross(a - c, d - c) * cross(b - c, d - c) > 0) return false;
    return true;
}

// Calcula a intersecao entre as retas a - b e c - d assumindo que uma unica intersecao
    existe
// Para intersecao de segmentos, cheque primeiro se os segmentos se intersectam e que nao
    paralelos
PT computeLineIntersection(PT a, PT b, PT c, PT d) {
    b = b - a; d = c - d; c = c - a;
    assert(cross(b, d) != 0); // garante que as retas nao sao paralelas, remover pra evitar
        tle
    return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos
PT computeCircleCenter(PT a, PT b, PT c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return computeLineIntersection(b, b + rotateCW90(a - b), c, c + rotateCW90(a - c));
}


// Determina se o ponto esta num poligno possivelmente nao-convexo
// Retorna 1 para pontos estritamente dentro, 0 para pontos estritamente fora do poligno
// e 0 ou 1 para os pontos restantes
// Eh possivel converter num teste exato usando inteiros e tomando cuidado com a divisao
// e entao usar testes exatos para checar se esta na borda do poligno
bool pointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    if((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// Determina se o ponto esta na borda do poligno
bool pointOnPolygon(const vector<PT> &p, PT q) {
  for(int i = 0; i < p.size(); i++)
    if(dist2(projectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < eps)
      return true;
    return false;
}

// Calcula intersecao da linha a - b com o circulo centrado em c com raio r > 0
vector<PT> circleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ans;
  b = b - a;
  a = a - c;
  double x = dot(b, b);
  double y = dot(a, b);
  double z = dot(a, a) - r * r;
  double w = y * y - x * z;
  if (w < -eps) return ans;
  ans.push_back(c + a + b * (-y + sqrt(w + eps)) / x);
  if (w > eps)
    ans.push_back(c + a + b * (-y - sqrt(w)) / x);
  return ans;
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado em b com raio R
vector<PT> circleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ans;
  double d = sqrt(dist2(a, b));
  if (d > r + R || d + min(r, R) < max(r, R)) return ans;
  double x = (d * d - R * R + r * r)/(2 * d);
  double y = sqrt(r * r - x * x);
  PT v = (b - a) / d;
  ans.push_back(a + v * x + rotateCCW90(v) * y);
  if (y > 0)
    ans.push_back(a + v * x - RotateCCW90(v) * y);
```

```
  return ans;
}

// Calcula a area ou o centroide de um poligono (possivelmente nao-convexo)
// assumindo que as coordenadas estao listada em ordem horaria ou anti-horaria
// O centroide eh equivalente a o centro de massa ou centro de gravidade
double computeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    area += p[i].x * p[j].y - p[j].x * p[i].y;
  }
  return area / 2.0;
}

double computeArea(const vector<PT> &p) {
  return abs(computeSignedArea(p));
}

PT computeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
  }
  return c / scale;
}

// Testa se o poligno listada em ordem CW ou CCW eh simples (nenhuma linha se intersecta)
bool isSimple(const vector<PT> &p) {
  for(int i = 0; i < p.size(); i++) {
    for(int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if (i == l || j == k) continue;
      if (segmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

## 5.2 Convex Hull

```
vector<PT> convexHull(vector<PT> p)) {
    int n = p.size(), k = 0;
    vector<PT> h(2 * n);
    sort(p.begin(), p.end());
    for(int i = 0; i < n; i++) {
        while(k >= 2 && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    for(int i = n - 2, t = k + 1; i >= 0; i--) {
        while(k >= t && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    h.resize(k);
    return h;
}
```

## 5.3 ClosestPair

```
double closestPair(vector<PT> p) {
    int n = p.size(), k = 0;
    sort(p.begin(), p.end());
    double d = inf;
    set<PT> ptsInv;
    for(int i = 0; i < n; i++) {
        while(k < i && p[k].x < p[i].x - d) {
            ptsInv.erase(swapCoord(p[k++]));
        }
        for(auto it = ptsInv.lower_bound(PT(p[i].y - d, p[i].x - d));
            it != ptsInv.end() && it->x <= p[i].y + d; it++) {
            d = min(d, !(p[i] - swapCoord(*it)));
        }
        ptsInv.insert(swapCoord(p[i]));
    }
    return d;
}
```

## 5.4 Intersection Points

```
// Utiliza uma seg tree

int intersectionPoints(vector<pair<PT, PT>> v) {
    int n = v.size();
    vector<pair<int, int>> events, vertInt;
    for(int i = 0; i < n; i++) {
        if(v.first.x == v.second.x) { // Segmento Vertical
            int y0 = min(v.first.y, v.second.y), y1 = max(v.first.y, v.second.y);
            events.push_back({v.first.x, vertInt.size()}); // Tipo = Indice no array
            vertInt.push_back({y0, y1});
        } else { // Segmento Horizontal
            int x0 = min(v.first.x, v.second.x), x1 = max(v.first.x, v.second.x);
            events.push_back({x0, -1}); // Inicio de Segmento
            events.push_back({x1, inf}); // Final de Segmento
        }
    }
    sort(events.begin(), events.end());
    int ans = 0;
    for(int i = 0; i < events.size(); i++) {
        int t = events[i].second;
        if(t == -1) {
            segUpdate(events[i].first, 1);
        } else if(t == inf) {
            segUpdate(events[i].first, 0);
        } else {
            ans += segQuery(vertInt[t].first, vertInt[t].second);
        }
    }
    return ans;
}
```

# 6 Miscellaneous

## 6.1 LIS - Longest Increasing Subsequence

```
int arr[ms], lisArr[ms], n;
// int bef[ms], pos[ms];

int lis() {
    int len = 1;
    lisArr[0] = arr[0];
    // bef[0] = -1;
    for(int i = 1; i < n; i++) {
        // upper_bound se non-decreasing
        int x = lower_bound(lisArr, lisArr + len, arr[i]) - lisArr;
        len = max(len, x + 1);
        lisArr[x] = arr[i];
        // pos[x] = i;
        // bef[i] = x ? pos[x-1] : -1;
    }
    return len;
}

vi getLis() {
    int len = lis();
    vi ans;
    for(int i = pos[lisArr[len - 1]]; i >= 0; i = bef[i]) {
        ans.push_back(arr[i]);
    }
```

```
    reverse(ans.begin(), ans.end());
    return ans;
}
```

## 6.2 Binary Search

```
int smallestSolution() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(!ok(x+b)) x += b;
    }
    return x + 1;
}

int maximumValue() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(f(x+b) < f(x+b+1)) x += b;
    }
    return x + 1;
}
```

## 6.3 Ternary Search

```
// R
for(int i = 0; i < LOG; i++){
        long double m1 = (A * 2 + B) / 3.0;
        long double m2 = (A + 2 * B) / 3.0;

        if(f(m1) > f(m2))
                A = m1;
        else
                B = m2;
}
ans = f(A);

// Z
while(B - A > 4){
        int m1 = (A + B) / 2;
        int m2 = (A + B) / 2 + 1;
        if(f(m1) > f(m2))
                A = m1;
        else
                B = m2;
}
ans = inf;
for(int i = A; i <= B; i++) ans = min(ans , f(i));
```

# 7 Teoremas e formulas uteis

## 7.1 Grafos

```
Formula de Euler: V - E + F = 2 (para grafo planar)
Handshaking: Numero par de vertices tem grau impar
Kirchhoff's Theorem: Monta matriz onde Mi,i = Grau[i] e Mi,j = -1 se houver aresta i-j ou 0
      caso contrario, remove uma linha e uma coluna qualquer e o numero de spanning trees
      nesse grafo eh o det da matriz

Grafo contem caminho hamiltoniano se:
Dirac's theorem: Se o grau de cada vertice for pelo menos n/2
Ore's theorem: Se a soma dos graus que cada par nao-adjacente de vertices for pelo menos n

Trees:
Tem Catalan(N) Binary trees de N vertices
Tem Catalan(N-1) Arvores enraizadas com N vertices
Caley Formula: n^(n-2) arvores em N vertices com label
Prufer code: Cada etapa voce remove a folha com menor label e o label do vizinho eh
      adicionado ao codigo ate ter 2 vertices

Flow:
Max Edge-disjoint paths: Max flow com arestas com peso 1
Max Node-disjoint paths: Faz a mesma coisa mas separa cada vertice em um com as arestas de
      chegadas e um com as arestas de saida e uma aresta de peso 1 conectando o vertice com
      aresta de chegada com ele mesmo com arestas de saida
Konig's Theorem: minimum node cover = maximum matching se o grafo for bipartido, complemento
      eh o maximum independent set
Min Node disjoint path cover: formar grafo bipartido de vertices duplicados, onde aresta sai
      do vertice tipo A e chega em tipo B, entao o path cover eh N - matching
Min General path cover: Mesma coisa mas colocando arestas de A pra B sempre que houver
      caminho de A pra B
Dilworth's Theorem: Min General Path cover = Max Antichain (set de vertices tal que nao
      existe caminho no grafo entre vertices desse set)
Hall's marriage: um grafo tem um matching completo do lado X se para cada subconjunto W de X,
    |W| <= |vizinhosW| onde |W| eh quantos vertices tem em W
```

## 7.2 Math

```
Goldbach's: todo numero par n > 2 pode ser representado com n = a + b onde a e b sao primos
Twin prime: existem infinitos pares p, p + 2 onde ambos sao primos
Legendre's: sempre tem um primo entre n^2 e (n+1)^2
Lagrange's: todo numero inteiro pode ser inscrito como a soma de 4 quadrados
Zeckendorf's: todo numero pode ser representado pela soma de dois numeros de fibonnacis
      diferentes e nao consecutivos
Euclid's: toda tripla de pitagoras primitiva pode ser gerada com
    (n^2 - m^2, 2nm, n^2+m^2) onde n, m sao coprimos e um deles eh par
Wilson's: n eh primo quando (n-1)! mod n = n - 1
Mcnugget: Para dois coprimos x, y o maior inteiro que nao pode ser escrito como ax + by eh (x
      -1)(y-1)/2

Fermat: Se p eh primo entao a^(p-1) % p = 1
Se x e m tambem forem coprimos entao x^k % m = x^(k mod(m-1)) % m
Euler's theorem: x^(phi(m)) mod m = 1 onde phi(m) eh o totiente de euler

Chinese remainder theorem:
Para equacoes no formato x = a1 mod m1, ... , x = an mod mn onde todos os pares m1, ... , mn
      sao coprimos
Deixe Xk = m1*m2*..*mn/mk e Xk^-1 mod mk = inverso de Xk mod mk, entao
x = somatorio com k de 1 ate n de ak*Xk*(Xk,mk^-1 mod mk)
Para achar outra solucao so somar m1*m2*..*mn a solucao existente

Catalan number: exemplo expressoes de parenteses bem formadas
C0 = 1, Cn = somatorio de i=0 -> n-1 de Ci*C(n-1+1)
outra forma: Cn = (2n escolhe n)/(n+1)
Bertrand's ballot theorem: p votos tipo A e q votos tipo B com p>q, prob de em todo ponto ter
       mais As do que Bs antes dele = (p-q)/(p+q)
Se puder empates entao prob = (p+1-q)/(p+1), para achar quantidade de possibilidades nos dois
       casos basta multiplicar por (p + q escolhe q)

Hockey-stick: Somatorio de i = r -> n de (i escolhe r) = (n + 1 escolhe r + 1)
Vandermonde: (m+n escolhe r) = somatorio de k = 0 -> r de (m escolhe k) * (n escolhe r - k)

Burnside lemma: colares diferentes nao contando rotacoes quando m = cores e n = comprimento
(m^n + somatorio i =1 - > n-1 de m^gcd(i, n))/n

Distribuicao uniforme a,a+1, ..., b Expected[X] = (a+b)/2
Distribuicao binomial com n tentativas de probabilidade p, X = sucessos:
    P(X = x) = p^x * (1-p)^(n-x) * (n escolhe x) e E[X] = p*n
Distribuicao geometrica onde continuamos ate ter sucesso, X = tentativas:
    P(X = x) = (1-p)^(x-1) * p e E[X] = 1/p
Linearity of expectation: Tendo duas variaveis X e Y e constantes a e b, o valor esperado de
      aX + bY = a*E[X] + b*E[X]
```

## 7.3 Geometry

```
Formula de Euler: V - E + F = 2
Pick Theorem: Para achar pontos em coords inteiras num poligono Area = i + b/2 - 1 onde i eh
      o o numero de pontos dentro do poligono e b de pontos no perimetro do poligono
```

Two ears theorem: Todo poligono simples com mais de 3 vertices tem pelo menos 2 orelhas, vertices que podem ser removidos sem criar um crossing, remover orelhas repetidamente triangula o poligono

Incentro triangulo: (a(Xa, Ya) + b(Xb, Yb) + c(Xc, Yc))/(a+b+c) onde a = lado oposto ao vertice a, incentro eh onde cruzam as bissetrizes, eh o centro da circunferencia inscrita e eh equidistante aos lados