UNIVERSIDADE FEDERAL
DE ALAGOAS

Universidade Federal de Alagoas

Floyd (WA)rshall

João Ayalla, Filipe Ferreira, Davi Romão

ICPC World Finals 2025

September 4, 2025

Contest (1)

template.cpp

25 lines

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

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

template <class T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

#define int long long int
#define pb push_back
#define pi pair<int, int>
#define pii pair<pi, int>
#define fir first
#define sec second
#define MAXN 1000006
#define mod 998244353

signed main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  mt19937_64 rng(chrono::steady_clock::now().time_since_epoch()
      .count());
  return 0;
}
```

runner.py

44 lines

```
import os
import subprocess

naive = "brute.cpp" # path to naive code
code = "d.cpp" # path to your code
generator = "g.cpp" # path to test generator

def compile_codes():
    os.system('g++ ' + generator + ' -o generator -O2')
    os.system('g++ ' + naive + ' -o naive -O2')
    os.system('g++ ' + code + ' -o code -O2')

def generate_case():
    os.system('./generator > in');

def get_naive_output():
    output = os.popen('./naive <in').read()
    return output

def get_code_output():
    # se tiver um runtime error, vai parar
    xalala = subprocess.run('./code <in', shell=True, text=True
        , capture_output=True, check=True)
    return xalala.stdout

def main():
    compile_codes()

    while True:
        generate_case()
        naive_output = get_naive_output()
        code_output = get_code_output()

        if naive_output == code_output:
            print('ACCEPTED')
        else :
            print('FAILED\n')
            print('ANSWER:')
            print(naive_output)
            print('\nCODE OUTPUT:')
            print(code_output)
            break

if __name__ == '__main__':
    main()
```

int128.cpp

22 lines

```
__int128 read() {
  __int128 x = 0, f = 1;
  char ch = getchar();
  while (ch < '0' || ch > '9') {
    if (ch == '-') f = -1;
    ch = getchar();
  }
  while (ch >= '0' && ch <= '9') {
    x = x * 10 + ch - '0';
    ch = getchar();
  }
  return x * f;
}
void print(__int128 x) {
  if (x < 0) {
    cout << "-";
    x = -x;
  }
  if (x > 9) print(x / 10);
  char at = (x % 10) + '0';
  cout << at;
}
```

pragma.cpp

5 lines

```
#include <iostream>
using namespace std;
#pragma GCC target("avx2")
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")
```

Mathematics (2)

modint.cpp

78 lines

```
struct modint {
  int val;
  modint(int v = 0) { val = v % mod; }
  int pow(int y) {
    modint x = val;
    modint z = 1;
    while (y) {
      if (y & 1) z *= x;
      x *= x;
      y >>= 1;
    }
    return z.val;
  }
  int inv() { return pow(mod - 2); }
  void operator=(int o) { val = o % mod; }
  void operator=(modint o) { val = o.val % mod; }
  void operator+=(modint o) { *this = *this + o; }
  void operator-=(modint o) { *this = *this - o; }
  void operator*=(modint o) { *this = *this * o; }
  void operator/=(modint o) { *this = *this / o; }
  bool operator==(modint o) { return val == o.val; }
  bool operator!=(modint o) { return val != o.val; }
  int operator*(modint o) { return ((val * o.val) % mod); }
  int operator/(modint o) { return (val * o.inv()) % mod; }
  int operator+(modint o) { return (val + o.val) % mod; }
  int operator-(modint o) { return (val - o.val + mod) % mod; }
};
modint f[MAXN];
modint inv[MAXN];
modint invfat[MAXN];
void calc() {
  f[0] = 1;
  for (int i = 1; i < MAXN; i++) {
    f[i] = f[i - 1] * i;
  }
  inv[1] = 1;
  for (int i = 2; i < MAXN; ++i) {
    int val = mod / i;
    val = (inv[mod % i] * val) % mod;
    val = mod - val;
    inv[i] = val;
  }
  invfat[0] = 1;
  invfat[MAXN - 1] = modint(f[MAXN - 1]).inv();
  for (int i = MAXN - 2; i >= 1; i--) {
    invfat[i] = invfat[i + 1] * (i + 1);
  }
}
modint ncr(int n, int k)  // combinacao
{
  modint ans = f[n] * invfat[k];
  ans *= invfat[n - k];
  return ans;
}
modint arr(int n, int k)  // arranjo
{
  modint ans = f[n] * invfat[n - k];
  return ans;
}
modint ncr(int n, int k) {
  // calcular combinacao para n grande
  // nesse problema n <= 10^12
  // em O(k)
  modint num = 1;
  modint den = 1;
  for (int i = 0; i < k; i++) {
    num = num * modint(n - i);
```

```
    den = den * modint(i + 1);
  }
  modint ans = num / den;
  return ans;
}
modint stars_and_bars(int n, int k) {
  // para pares de inteiros n e k
  // enconte a quantidade de k-tuplas com soma == n
  // x1 + x2 + ... + xk = n
  return ncr(n + k - 1, k - 1);
}
```

divisiontrick.cpp

5 lines

```
for (int l = 1, r; l <= n; l = r + 1) {
  // todos os numeros i no intervalo [l, r] possuem (n / i) ==
       x
  r = n / (n / l);
  int x = (n / l);
}
```

crivo.cpp

23 lines

```
bitset<MAXN> prime;
vector<int> nxt(MAXN);
vector<int> factors;
void crivo() {
  prime.set();
  prime[0] = false, prime[1] = false;
  for (int i = 2; i < MAXN; i++) {
    if (prime[i]) {
      nxt[i] = i;
      for (int j = 2; j * i < MAXN; j++) {
        prime[j * i] = false;
        nxt[j * i] = i;
      }
    }
  }
}
void fact(int n) {
  factors.clear();
  while (n > 1) {
    factors.pb(nxt[n]);
    n = n / nxt[n];
  }
}
```

gaussianelimination.cpp

33 lines

```
#define EPS 1e-9
vector<double> ans;
int gauss(vector<vector<double>> a) {
  int n = a.size(), m = a[0].size() - 1, ret = 1;
  ans.assign(m, 0);
  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; col++, row++)
       {
    int sel = row;
    for (int i = row; i < n; i++)
      if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;
    for (int i = col; i <= m; i++) swap(a[sel][i], a[row][i]);
    where[col] = row;
    for (int i = 0; i < n; i++) {
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j = col; j <= m; j++) a[i][j] -= a[row][j] * c
             ;
      }
    }
  }
  for (int i = 0; i < m; i++) {
    if (where[i] != -1)
      ans[i] = (a[where[i]][m] / a[where[i]][i]);
    else
      ret = 2;
  }
  for (int i = 0; i < n; i++) {
    double sum = 0;
    for (int j = 0; j < m; j++) sum += (ans[j] * a[i][j]);
    if (abs(sum - a[i][m]) > EPS) ret = 0;
  }
  return ret;  // 0 = nao existe solucao, 1 = existe uma
       solucao, 2 = existem multiplas solucoes
}
```

gaussianbitset.cpp

31 lines

```
bitset<MAXN> ans;
int gauss(vector<bitset<MAXN>>& a) {
  ans.reset();
  int n = a.size(), m = a[0].size() - 1, ret = 1;
  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; col++) {
    for (int i = row; i < n; i++) {
      if (a[i][col]) {
        swap(a[i], a[row]);
        break;
      }
    }
    if (!a[row][col]) continue;
    where[col] = row;
    for (int i = 0; i < n; i++)
      if (i != row && a[i][col]) a[i] ^= a[row];
    ++row;
  }
  for (int i = 0; i < m; i++) {
    if (where[i] != -1)
      ans[i] = (a[where[i]][m] / a[where[i]][i]);
    else
      ret = 2;
  }
  for (int i = 0; i < n; i++) {
    double sum = 0;
    for (int j = 0; j < m; j++) sum += (ans[j] * a[i][j]);
    if (abs(sum - a[i][m]) > EPS) ret = 0;
  }
  return ret;
}
```

Data structures (3)

segtree.cpp

25 lines

```
struct segtree {
  int n;
  vector<int> seg;
  int neutral() { return 0; }
  int merge(int a, int b) { return a + b; }
  void build(vector<int> &v) {
    n = 1;
    while (n < v.size()) n <<= 1;
    seg.assign(n << 1, neutral());
    for (int i = 0; i < v.size(); i++) seg[i + n] = v[i];
    for (int i = n - 1; i; i--) seg[i] = merge(seg[i << 1], seg
         [(i << 1) | 1]);
  }
  void upd(int i, int value) {
    seg[i += n] += value;
    for (i >>= 1; i; i >>= 1) seg[i] = merge(seg[i << 1], seg[(
         i << 1) | 1]);
  }
  int qry(int l, int r) {
    int ansl = neutral(), ansr = neutral();
    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
      if (l & 1) ansl = merge(ansl, seg[l++]);
      if (r & 1) ansr = merge(seg[--r], ansr);
    }
    return merge(ansl, ansr);
  }
};
```

segtreelazy.cpp

53 lines

```
struct segtree {
  int n;
  vector<int> v;
  vector<int> seg;
  vector<int> lazy;
  segtree(int sz) {
    n = sz;
    seg.assign(4 * n, 0);
    lazy.assign(4 * n, 0);
  }
  int single(int x) { return x; }
  int neutral() { return 0; }
  int merge(int a, int b) { return a + b; }
  void add(int i, int l, int r, int diff) {
    seg[i] += (r - l + 1) * diff;
    if (l != r) {
      lazy[i << 1] += diff;
      lazy[(i << 1) | 1] += diff;
    }
    lazy[i] = 0;
  }
  void update(int i, int l, int r, int ql, int qr, int diff) {
    if (lazy[i]) add(i, l, r, lazy[i]);
    if (l > r || l > qr || r < ql) return;
    if (l >= ql && r <= qr) {
      add(i, l, r, diff);
      return;
    }
    int mid = (l + r) >> 1;
    update(i << 1, l, mid, ql, qr, diff);
    update((i << 1) | 1, mid + 1, r, ql, qr, diff);
    seg[i] = merge(seg[i << 1], seg[(i << 1) | 1]);
  }
  int query(int l, int r, int ql, int qr, int i) {
    if (lazy[i]) add(i, l, r, lazy[i]);
    if (l > r || l > qr || r < ql) return neutral();
    if (l >= ql && r <= qr) return seg[i];
    int mid = (l + r) >> 1;
    return merge(query(l, mid, ql, qr, i << 1), query(mid + 1,
         r, ql, qr, (i << 1) | 1));
  }
  void build(int l, int r, int i) {
    if (l == r) {
      seg[i] = single(v[l]);
      return;
    }
    int mid = (l + r) >> 1;
    build(l, mid, i << 1);
    build(mid + 1, r, (i << 1) | 1);
    seg[i] = merge(seg[i << 1], seg[(i << 1) | 1]);
  }
  int qry(int l, int r) { return query(0, n - 1, l, r, 1); }
```

```
  void upd(int l, int r, int x) { update(1, 0, n - 1, l, r, x);
      }
};
```

fenwick.cpp

33 lines

```
struct fenw {
  int n;
  vector<int> bit;
  fenw() {}
  fenw(int sz) {
    n = sz;
    bit.assign(sz + 1, 0);
  }
  int qry(int r)  // query de prefixo a[0] + a[1] + ...  a[r]
  {
    int ret = 0;
    for (int i = r + 1; i > 0; i -= i & -i) ret += bit[i];
    return ret;
  }
  void upd(int r, int x)  // a[r] += x
  {
    for (int i = r + 1; i <= n; i += i & -i) bit[i] += x;
  }
  int bs(int x)  // retorna o maior indice i (i < n) tal que:
      qry(i) < x
  {
    int i = 0, k = 0;
    while (1 << (k + 1) <= n) k++;
    while (k >= 0) {
      int nxt_i = i + (1 << k);
      if (nxt_i <= n && bit[nxt_i] < x) {
        i = nxt_i;
        x -= bit[i];
      }
      k--;
    }
    return i - 1;
  }
};
```

treap.cpp

93 lines

```
struct treap {
  int data, priority;
  int sz, lazy2;
  bool lazy;
  treap *l, *r, *parent;
};
int size(treap *node) { return (!node) ? 0 : node->sz; }
void recalc(treap *node) {
  if (!node) return;
  node->sz = 1;
  node->parent = 0;
  if (node->l) node->sz += node->l->sz, node->l->parent = node;
  if (node->r) node->sz += node->r->sz, node->r->parent = node;
}
void lazy_propagation(treap *node) {
  if (node == NULL) return;
  if (node->lazy2) {
    if (node->l) node->l->lazy2 += node->lazy2;
    if (node->r) node->r->lazy2 += node->lazy2;
    node->data += node->lazy2;
    node->lazy2 = 0;
  }
  if (node->lazy) {
    swap(node->l, node->r);
    if (node->l) node->l->lazy = !node->l->lazy;
    if (node->r) node->r->lazy = !node->r->lazy;
    node->lazy = 0;
  }
}
void split(treap *t, treap *&l, treap *&r, int n) {
  if (!t) return void(l = r = 0);
  lazy_propagation(t);
  if (size(t->l) >= n)
    split(t->l, l, t->l, n), r = t;
  else
    split(t->r, t->r, r, n - size(t->l) - 1), l = t;
  recalc(t);
}
void merge(treap *&t, treap *l, treap *r) {
  lazy_propagation(l);
  lazy_propagation(r);
  if (!l)
    t = r;
  else if (!r)
    t = l;
  else if (l->priority > r->priority)
    merge(l->r, l->r, r), t = l;
  else
    merge(r->l, l, r->l), t = r;
  recalc(t);
}
void troca(treap *&t, int l, int r, int ll, int rr)  // sap de
    ranges
{
  treap *a0, *a1, *b0, *b1, *c0, *c1, *d0, *d1;
  split(t, a0, a1, l);
  split(a1, b0, b1, r - l + 1);
  ll -= (r + 1);
  rr -= (r + 1);
  split(b1, c0, c1, ll);
  split(c1, d0, d1, rr - ll + 1);
  merge(t, a0, d0);
  merge(t, t, c0);
  merge(t, t, b0);
  merge(t, t, d1);
}
void add(treap *&t, int l, int r) {
  treap *a0, *a1, *b0, *b1;
  split(t, a0, a1, l);
  split(a1, b0, b1, r - l + 1);
  b0->lazy ^= 1;
  b0->lazy2 += 1;
  merge(t, a0, b0);
  merge(t, t, b1);
}
void dfs(treap *t) {
  if (!t) return;
  lazy_propagation(t);
  dfs(t->l);
  solve(t->data);
  dfs(t->r);
}
treap *create_node(int data, int priority) {
  treap *ret = new treap;
  ret->data = data;
  ret->priority = priority;
  ret->l = 0;
  ret->r = 0;
  ret->sz = 1;
  ret->lazy = 0;
  ret->lazy2 = 0;
  ret->parent = 0;
  return ret;
}
```

colorupdate.cpp

74 lines

```
const int inf = 1e15;
struct color_upd {
#define left fir
#define right sec.fir
#define color sec.sec
  set<pii> ranges;
  vector<pii> erased;
  color_upd(int n)  // inicialmente, todo mundo pintado com a
      cor inf
  {
    // nao usar cores negativas!!!!!!!!!
    ranges.insert({0, {n - 1, inf}});
  }
  int get(int i) {
    auto it = ranges.upper_bound({i, {1e18, 1e18}});
    if (it == ranges.begin()) return -1;
    it--;
    return (*(it)).color;
  }
  void del(int l, int r)  // apaga o intervalo [l, r]
  {
    erased.clear();
    auto it = ranges.upper_bound({l, {0, 0}});
    if (it != ranges.begin()) {
      it--;
    }
    while (it != ranges.end()) {
      if ((*(it)).left > r)
        break;
      else if ((*(it)).right >= l)
        erased.push_back(*it);
      it++;
    }
    if (erased.size() > 0) {
      int sz = erased.size();
      auto it = ranges.lower_bound({erased[0].left, {0, 0}});
      auto it2 = ranges.lower_bound({erased[sz - 1].left, {0, 0
          }});
      pii ini = *it, fim = *it2;
      it2++;
      ranges.erase(it, it2);
      pii upd1 = {ini.left, {l - 1, ini.color}};
      pii upd2 = {r + 1, {fim.right, fim.color}};
      erased[0].left = max(erased[0].left, l);
      erased[sz - 1].right = min(erased[sz - 1].right, r);
      if (upd1.left <= upd1.right) ranges.insert(upd1);
      if (upd2.left <= upd2.right) ranges.insert(upd2);
    }
  }
  void add(int a, int b, int c) {
    auto it = ranges.lower_bound({a, {b, 0}});
    pii aa = {-1, {-1, -1}};
    pii bb = {-1, {-1, -1}};
    if (it != ranges.end()) {
      if ((*it).color == c && (*it).left == b + 1) {
        aa = *it;
        b = (*it).right;
      }
    }
    if (it != ranges.begin()) {
      it--;
      if ((*it).color == c && (*it).right == a - 1) {
        bb = *it;
        a = (*it).left;
      }
    }
    ranges.erase(aa);
```

```
    ranges.erase(bb);
    ranges.insert({a, {b, c}});
  }
  void upd(int a, int b, int c)  // pinta o intervalo [a, b]
       com a cor c
  {
    del(a, b);
    add(a, b, c);
  }
};
```

bit2d.cpp

51 lines

```
struct bit2d {
  vector<int> ord;
  vector<vector<int>> t;
  vector<vector<int>> coord;
  bit2d(vector<pi> &pts)  // recebe todos os pontos que vao ser
       inseridos pra construir, mas nao insere eles
  {
    sort(pts.begin(), pts.end());
    for (auto const &a : pts) {
      if (ord.empty() || a.fir != ord.back()) ord.pb(a.fir);
    }
    t.resize(ord.size() + 1);
    coord.resize(t.size());
    for (auto &a : pts) {
      swap(a.fir, a.sec);
    }
    sort(pts.begin(), pts.end());
    for (auto &a : pts) {
      swap(a.fir, a.sec);
      for (int on = upper_bound(ord.begin(), ord.end(), a.fir)
           - ord.begin(); on < t.size(); on += on & -on) {
        if (coord[on].empty() || coord[on].back() != a.sec)
            coord[on].push_back(a.sec);
      }
    }
    for (int i = 0; i < t.size(); i++) t[i].assign(coord[i].
        size() + 1, 0);
  }
  void add(int x, int y, int v)  // v[a][b] += v
  {
    for (int xx = upper_bound(ord.begin(), ord.end(), x) - ord.
        begin(); xx < t.size(); xx += xx & -xx) {
      for (int yy = upper_bound(coord[xx].begin(), coord[xx].
          end(), y) - coord[xx].begin(); yy < t[xx].size();
           yy += yy & -yy)
        t[xx][yy] += v;
    }
  }
  int qry(int x, int y)  // soma de todos os v[a][b] com (a <=
       x && b <= y)
  {
    int ans = 0;
    for (int xx = upper_bound(ord.begin(), ord.end(), x) - ord.
        begin(); xx > 0; xx -= xx & -xx) {
      for (int yy = upper_bound(coord[xx].begin(), coord[xx].
          end(), y) - coord[xx].begin(); yy > 0; yy -= yy & -
          yy)
        ans += t[xx][yy];
    }
    return ans;
  }
  int qry2(int x1, int y1, int x2, int y2) {
    return qry(x2, y2) - qry(x2, y1 - 1) - qry(x1 - 1, y2) +
        qry(x1 - 1, y1 - 1);
  }
  void add2(int x1, int y1, int x2, int y2, int v) {
    add(x1, y1, v);
    add(x1, y2 + 1, -v);
    add(x2 + 1, y1, -v);
    add(x2 + 1, y2 + 1, v);
  }
};
```

mo.cpp

45 lines

```
namespace mo {
struct query {
  int idx, l, r;
};
int block;
vector<query> queries;
vector<int> ans;
// bool cmp(query &x, query &y){  essa funcao de ordenacao pode
     funcionar em caso de TLE
//   int ablock = x.l / MAGIC, bblock = y.l / MAGIC;
//   if (ablock != bblock) return ablock < bblock;
//   if (ablock & 1) return x.r < y.r;
//   return x.r > y.r;
// }
bool cmp(query &x, query &y) {
  if (x.l / block != y.l / block) return x.l / block < y.l /
      block;
  return x.r < y.r;
}
void run() {
  block = (int)sqrt(n);
  sort(queries.begin(), queries.end(), cmp);
  ans.resize(queries.size());
  int cl = 0, cr = -1, sum = 0;
  auto add = [&](int x) { sum += x; };
  auto rem = [&](int x) { sum -= x; };
  for (int i = 0; i < queries.size(); i++) {
    while (cl > queries[i].l) {
      cl--;
      add(v[cl]);
    }
    while (cr < queries[i].r) {
      cr++;
      add(v[cr]);
    }
    while (cl < queries[i].l) {
      rem(v[cl]);
      cl++;
    }
    while (cr > queries[i].r) {
      rem(v[cr]);
      cr--;
    }
    ans[queries[i].idx] = sum;
  }
}
}  // namespace mo
```

segtree2d.cpp

33 lines

```
struct segtree2d {
  int n, m;
  vector<vector<int>> seg;
  int neutral() { return 0; }
  int merge(int a, int b) { return a + b; }
  segtree2d(int nn, int mm) {
    n = nn, m = mm;
    seg = vector<vector<int>>(2 * n, vector<int>(2 * m, neutral
        ()));
  }
  int qry(int x1, int y1, int x2, int y2) {
    int ret = neutral();
    int y3 = y1 + m, y4 = y2 + m;
    for (x1 += n, x2 += n; x1 <= x2; ++x1 /= 2, --x2 /= 2) {
      for (y1 = y3, y2 = y4; y1 <= y2; ++y1 /= 2, --y2 /= 2) {
        if (x1 % 2 == 1 and y1 % 2 == 1) ret = merge(ret, seg[
            x1][y1]);
        if (x1 % 2 == 1 and y2 % 2 == 0) ret = merge(ret, seg[
            x1][y2]);
        if (x2 % 2 == 0 and y1 % 2 == 1) ret = merge(ret, seg[
            x2][y1]);
        if (x2 % 2 == 0 and y2 % 2 == 0) ret = merge(ret, seg[
            x2][y2]);
      }
    }
    return ret;
  }
  void upd(int x, int y, int val) {
    int y2 = y += m;
    for (x += n; x; x /= 2, y = y2) {
      if (x >= n)
        seg[x][y] = val;
      else
        seg[x][y] = merge(seg[2 * x][y], seg[2 * x + 1][y]);
      while (y /= 2) seg[x][y] = merge(seg[x][2 * y], seg[x][2
          * y + 1]);
    }
  }
};
```

persistentseg.cpp

37 lines

```
struct node {
  int item, l, r;
  node() {}
  node(int l, int r, int item) : l(l), r(r), item(item) {}
};
int n, q;
vector<node> seg;
vector<int> roots;
void init() { seg.resize(1); }
int newleaf(int vv) {
  int p = seg.size();
  seg.pb(node(0, 0, vv));
  return p;
}
int newpar(int l, int r) {
  int p = seg.size();
  seg.pb(node(l, r, seg[l].item + seg[r].item));
  return p;
}
int upd(int i, int l, int r, int pos) {
  if (l == r) return newleaf(seg[i].item + 1);
  int mid = (l + r) >> 1;
  if (pos <= mid) return newpar(upd(seg[i].l, l, mid, pos), seg
      [i].r);
  return newpar(seg[i].l, upd(seg[i].r, mid + 1, r, pos));
}
int build(int l, int r) {
  if (l == r) return newleaf(0);
  int mid = (l + r) >> 1;
  return newpar(build(l, mid), build(mid + 1, r));
}
int qry(int vl, int vr, int l, int r, int k) {
  if (l == r) return l;
  int mid = (l + r) >> 1;
  int c = seg[seg[vr].l].item - seg[seg[vl].l].item;
  if (c >= k) return qry(seg[vl].l, seg[vr].l, l, mid, k);
  return qry(seg[vl].r, seg[vr].r, mid + 1, r, k - c);
```

```
}
```

rmq.cpp

21 lines

```
struct rmq {
  int n;
  vector<vector<pi>> m;
  vector<int> log;
  rmq() {}
  rmq(vector<pi> &v) {
    n = v.size();
    log.resize(n + 1);
    log[1] = 0;
    for (int i = 2; i <= n; i++) log[i] = log[i / 2] + 1;
    int sz = log[n] + 2;
    m = vector<vector<pi>>(sz, vector<pi>(n + 1));
    for (int i = 0; i < n; i++) {
      m[0][i] = v[i];
    }
    for (int j = 1; j < sz; j++) {
      for (int i = 0; i + (1 << j) <= n; i++) m[j][i] = min(m[j
          - 1][i], m[j - 1][i + (1 << (j - 1))]);
    }
  }
  int qry(int a, int b) { return min(m[log[b - a + 1]][a], m[
      log[b - a + 1]][b - (1 << log[b - a + 1]) + 1]).second;
      }
};
```

binarylifting.cpp

23 lines

```
item st[MAXN][21];
for (int i = 0; i < n; i++) {
  st[i][0].nxt = min(i + 1, n - 1);
  st[i][0].sum = v[st[i][0].nxt];
}
for (int i = 1; i < 21; i++) {
  for (int v = 0; v < n; v++) {
    st[v][i].nxt = st[st[v][i - 1].nxt][i - 1].nxt;
    st[v][i].sum = st[v][i - 1].sum + st[st[v][i - 1].nxt][i -
        1].sum;
  }
}
while (q--) {
  int l, r;
  cin >> l >> r;
  int ans = v[l], len = r - l;
  for (int i = 20; i >= 0; i--) {
    if (len & (1 << i)) {
      ans += st[l][i].sum;
      l = st[l][i].nxt;
    }
  }
  cout << ans << endl;
}
```

Graph (4)

floydwarshall.cpp

21 lines

```
int dist[MAXN][MAXN];
void floyd_warshall() {
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
      }
    }
  }
}
void initialize() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (i == j) {
        dist[i][j] = 0;
      } else {
        dist[i][j] = INF;
      }
    }
  }
}
```

centroiddecomp.cpp

68 lines

```
int n, k, resp;
vector<int> adj[MAXN];
vector<int> cnt;
namespace cd {
int sz;
vector<int> subtree_size;
vector<bool> visited;
void dfs(int s, int f) {
  sz++;
  subtree_size[s] = 1;
  for (auto const &v : adj[s]) {
    if (v != f && !visited[v]) {
      dfs(v, s);
      subtree_size[s] += subtree_size[v];
    }
  }
}
int get_centroid(int s, int f) {
  bool is_centroid = true;
  int heaviest_child = -1;
  for (auto const &v : adj[s]) {
    if (v != f && !visited[v]) {
      if (subtree_size[v] > sz / 2) is_centroid = false;
      if (heaviest_child == -1 || subtree_size[v] >
          subtree_size[heaviest_child]) heaviest_child = v;
    }
  }
  return (is_centroid && sz - subtree_size[s] <= sz / 2) ? s :
      get_centroid(heaviest_child, s);
}
void dfs2(int s, int f, int d) {
  while (d >= cnt.size()) cnt.pb(0);
  cnt[d]++;
  for (auto const &v : adj[s]) {
    if (v != f && !visited[v]) dfs2(v, s, d + 1);
  }
}
void solve(int s) {
  vector<int> tot;
  for (auto const &v : adj[s]) {
    if (visited[v]) continue;
    cnt.clear();
    dfs2(v, s, 1);
    for (int i = 1; i < cnt.size(); i++) {
      if (k - i < tot.size() && k - i >= 1) resp += (cnt[i] *
          tot[k - i]);
    }
    for (int i = 1; i < cnt.size(); i++) {
      while (i >= tot.size()) tot.pb(0);
      tot[i] += cnt[i];
    }
  }
  if (k < tot.size()) resp += tot[k];
}
int decompose_tree(int s) {
  sz = 0;
  dfs(s, s);
  int cend_tree = get_centroid(s, s);
  visited[cend_tree] = true;
  solve(cend_tree);
  for (auto const &v : adj[cend_tree]) {
    if (!visited[v]) decompose_tree(v);
  }
  return cend_tree;
}
void init() {
  subtree_size.resize(n);
  visited.resize(n);
  decompose_tree(0);
}
}  // namespace cd
```

dsu.cpp

24 lines

```
struct dsu {
  int tot;
  vector<int> parent;
  vector<int> sz;
  dsu(int n) {
    parent.resize(n);
    sz.resize(n);
    tot = n;
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      sz[i] = 1;
    }
  }
  int find_set(int i) { return parent[i] = (parent[i] == i) ? i
      : find_set(parent[i]); }
  void make_set(int x, int y) {
    x = find_set(x), y = find_set(y);
    if (x != y) {
      if (sz[x] > sz[y]) swap(x, y);
      parent[x] = y;
      sz[y] += sz[x];
      tot--;
    }
  }
};
```

dsubipartido.cpp

41 lines

```
struct dsu {
  vector<pi> parent;
  vector<int> rank;
  vector<int> bipartite;
  dsu(int n) {
    parent.resize(n);
    rank.resize(n);
    bipartite.resize(n);
    for (int v = 0; v < n; v++) {
      parent[v] = {v, 0};
      rank[v] = 0;
      bipartite[v] = 1;
    }
  }
  dsu() {}
  pi find_set(int v) {
    if (v != parent[v].fir) {
      int parity = parent[v].sec;
      parent[v] = find_set(parent[v].fir);
      parent[v].sec ^= parity;
```

```
    }
    return parent[v];
  }
  void add_edge(int a, int b) {
    pi pa = find_set(a);
    a = pa.fir;
    int x = pa.sec;
    pi pb = find_set(b);
    b = pb.fir;
    int y = pb.sec;
    if (a == b) {
      if (x == y) bipartite[a] = 0;
    } else {
      if (rank[a] < rank[b]) swap(a, b);
      parent[b] = {a, x ^ y ^ 1};
      bipartite[a] &= bipartite[b];
      if (rank[a] == rank[b]) rank[a]++;
    }
  }
  bool is_bipartite(int v) { return bipartite[find_set(v).fir];
       }
};
```

cycledetection.cpp

26 lines

```
int n, m, idx;
vector<int> cycles[MAXN];
vector<int> adj[MAXN];
int color[MAXN];
int parent[MAXN];
int ans[MAXN];
void dfs(int u, int p) {  // chama dfs a partir de qm tem cor 0
  if (color[u] == 2) return;
  if (color[u] == 1) {
    idx++;
    int curr = p;
    ans[curr] = idx;
    cycles[idx].pb(curr);
    while (curr != u) {
      curr = parent[curr];
      cycles[idx].pb(curr);
      ans[curr] = idx;
    }
    return;
  }
  parent[u] = p;
  color[u] = 1;
  for (auto const &v : adj[u])
    if (v != parent[u]) dfs(v, u);
  color[u] = 2;
}
```

blockcuttree.cpp

74 lines

```
struct block_cut_tree {
  // Se art[i] >= 1, i eh ponto de articulacao
  // tree - eh a propia block-cut tree
  // pos[i] responde a qual vertice da arvore vertice i
       pertence
  vector<vector<int>> g, blocks, tree;
  vector<vector<pi>> edgblocks;  // sao as arestas do bloco i
  stack<int> s;
  stack<pi> s2;
  vector<int> id, art, pos;

  block_cut_tree(vector<vector<int>> g_) : g(g_) {
    int n = g.size();
    id.resize(n, -1), art.resize(n), pos.resize(n);
    build();
  }
  int dfs(int i, int &t, int p = -1) {
    int lo = id[i] = t++;
    s.push(i);
    if (p != -1) {
      s2.emplace(i, p);
    }
    for (int j : g[i]) {
      if (j != p and id[j] != -1) s2.emplace(i, j);
    }
    for (int j : g[i]) {
      if (j != p) {
        if (id[j] == -1) {
          int val = dfs(j, t, i);
          lo = min(lo, val);
          if (val >= id[i]) {
            art[i]++;
            blocks.emplace_back(1, i);
            while (blocks.back().back() != j) {
              blocks.back().pb(s.top());
              s.pop();
            }
            edgblocks.emplace_back(1, s2.top());
            s2.pop();
            pi aux = {j, i};
            while (edgblocks.back().back() != aux) {
              edgblocks.back().pb(s2.top());
              s2.pop();
            }
          }
          // if (val > id[i]) aresta i-j eh ponte
        } else {
          lo = min(lo, id[j]);
        }
      }
    }
    if (p == -1 and art[i]) {
      art[i]--;
    }
    return lo;
  }
  void build() {
    int t = 0;
    for (int i = 0; i < g.size(); i++) {
      if (id[i] == -1) dfs(i, t, -1);
    }
    tree.resize(blocks.size());
    for (int i = 0; i < g.size(); i++) {
      if (art[i]) pos[i] = tree.size(), tree.emplace_back();
    }
    for (int i = 0; i < blocks.size(); i++) {
      for (int j : blocks[i]) {
        if (!art[j])
          pos[j] = i;
        else
          tree[i].pb(pos[j]), tree[pos[j]].pb(i);
      }
    }
  }
};
```

bridges.cpp

21 lines

```
nt n, m, timer;
vector<pi> edges;
vector<bool> is_bridge;
vector<pi> adj[MAXN];
int tin[MAXN];
int low[MAXN];            // memset -1
bool vis[MAXN];           // memset -1
void dfs(int v, int p) {  // chama de quem nao foi vis ainda
  vis[v] = true;
  tin[v] = timer, low[v] = timer++;
  for (auto const &u : adj[v]) {
    if (u.fir == p) continue;
    if (vis[u.fir]) {
      low[v] = min(low[v], tin[u.fir]);
      continue;
    }
    dfs(u.fir, v);
    low[v] = min(low[v], low[u.fir]);
    if (low[u.fir] > tin[v]) is_bridge[u.sec] = 1;
  }
}
```

dinic.cpp

83 lines

```
#define INF 1e9
struct edge {
  int to, from, flow, capacity, id;
};
struct dinic {
  int n, src, sink;
  vector<vector<edge>> adj;
  vector<int> level;
  vector<int> ptr;

  dinic(int sz) {
    n = sz;
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
  }
  void add_edge(int a, int b, int c, int id) {
    adj[a].pb({b, (int)adj[b].size(), c, c, id});
    adj[b].pb({a, (int)adj[a].size() - 1, 0, 0, id});
  }
  bool bfs() {
    level.assign(n, -1);
    level[src] = 0;
    queue<int> q;
    q.push(src);
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (auto at : adj[u]) {
        if (at.flow && level[at.to] == -1) {
          q.push(at.to);
          level[at.to] = level[u] + 1;
        }
      }
    }
    return level[sink] != -1;
  }
  int dfs(int u, int flow) {
    if (u == sink || flow == 0) return flow;
    for (int &p = ptr[u]; p < adj[u].size(); p++) {
      edge &at = adj[u][p];
      if (at.flow && level[u] == level[at.to] - 1) {
        int kappa = dfs(at.to, min(flow, at.flow));
        at.flow -= kappa;
        adj[at.to][at.from].flow += kappa;
        if (kappa != 0) return kappa;
      }
    }
    return 0;
  }
  int run() {
```

```
    int max_flow = 0;
    while (bfs()) {
      ptr.assign(n, 0);
      while (1) {
        int flow = dfs(src, INF);
        if (flow == 0) break;
        max_flow += flow;
      }
    }
    return max_flow;
  }
  vector<pii> cut_edges()  // arestas do corte minimo
  {
    bfs();
    vector<pii> ans;
    for (int i = 0; i < n; i++) {
      for (auto const &j : adj[i]) {
        if (level[i] != -1 && level[j.to] == -1 && j.capacity >
             0) ans.pb({j.capacity, {i, j.to}});
      }
    }
    return ans;
  }
  vector<int> flow_edges(int n, int m)  // fluxo em cada aresta
       , na ordem da entrada
  {
    vector<int> ans(m);
    for (int i = 0; i < n; i++) {
      for (auto const &j : adj[i])
        if (!j.capacity) ans[j.id] = j.flow;
    }
    return ans;
  }
};
```

hopcroftkarp.cpp

137 lines

```
#define INF 1e9
struct hopcroft_karp {
  vector<int> match;
  vector<int> dist;
  vector<vector<int>> adj;
  int n, m, t;
  hopcroft_karp(int a, int b) {
    n = a, m = b;
    t = n + m + 1;
    match.assign(t, n + m);
    dist.assign(t, 0);
    adj.assign(t, vector<int>{});
  }
  void add_edge(int u, int v) {
    adj[u].pb(v);
    adj[v].pb(u);
  }
  bool bfs() {
    queue<int> q;
    for (int u = 0; u < n; u++) {
      if (match[u] == n + m)
        dist[u] = 0, q.push(u);
      else
        dist[u] = INF;
    }
    dist[n + m] = INF;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (dist[u] < dist[n + m]) {
        for (auto const &v : adj[u]) {
          if (dist[match[v]] == INF) {
            dist[match[v]] = dist[u] + 1;
            q.push(match[v]);
          }
        }
      }
    }
    return dist[n + m] < INF;
  }
  bool dfs(int u) {
    if (u < n + m) {
      for (auto const &v : adj[u]) {
        if (dist[match[v]] == dist[u] + 1 && dfs(match[v])) {
          match[v] = u;
          match[u] = v;
          return true;
        }
      }
      dist[u] = INF;
      return false;
    }
    return true;
  }
  vector<pi> run() {
    int cnt = 0;
    while (bfs())
      for (int u = 0; u < n; u++)
        if (match[u] == n + m && dfs(u)) cnt++;
    vector<pi> ans;
    for (int v = n; v < n + m; v++)
      if (match[v] < n + m) ans.pb({match[v], v});
    return ans;
  }
  vector<int> mvc()  // minimum vertex cover
  {
    vector<pi> ans = run();
    vector<bool> vis(n + m, 0);
    for (int i = 0; i < n; i++) {
      if (match[i] == n + m) {
        queue<int> q;
        q.push(i);
        while (!q.empty()) {
          int x = q.front();
          q.pop();
          vis[x] = 1;
          for (auto const &y : adj[x]) {
            if (!vis[y]) {
              vis[y] = 1;
              q.push(match[y]);
            }
          }
        }
      }
    }
    vector<int> vc;
    for (int i = 0; i < n; i++) {
      if (!vis[i]) vc.pb(i);
    }
    for (int i = n; i < n + m; i++) {
      if (vis[i]) vc.pb(i);
    }
    return vc;
  }
  vector<pi> mec()  // minimum edge cover
  {
    vector<pi> ans = run();
    for (int i = 0; i < n + m; i++) {
      if (match[i] == n + m && adj[i].size() > 0) {
        if (i < n)
          ans.pb({i, adj[i][0]});
        else
          ans.pb({adj[i][0], i});
      }
    }
    return ans;
  }
};
// minimum path cover on dag
// minimum set of paths such that each of the vertices belongs
     to exactly one path
vector<vector<int>> mpc(int n, vector<pi> &e) {
  hopcroft_karp h(n, n);
  for (auto const &i : e) h.add_edge(i.fir, n + i.sec);
  vector<pi> mat = h.run();
  vector<int> prv(n, -1);
  vector<int> nxt(n, -1);
  for (int i = 0; i < mat.size(); i++) {
    nxt[mat[i].fir] = mat[i].sec - n;
    prv[mat[i].sec - n] = mat[i].fir;
  }
  vector<vector<int>> ans;
  for (int i = 0; i < n; i++) {
    if (prv[i] == -1 && nxt[i] == -1) {
      ans.pb({i});
    } else if (prv[i] == -1) {
      vector<int> curr;
      int x = i;
      while (1) {
        curr.pb(x);
        if (nxt[x] == -1) break;
        x = nxt[x];
      }
      ans.pb(curr);
    }
  }
  return ans;
}
```

pushrelabel.cpp

93 lines

```
#define INF 1e9
struct edge {
  int dest, back, f, c, id;
};
struct push_relabel {
  int n;
  vector<vector<edge>> g;
  vector<int> ec;
  vector<edge *> cur;
  vector<vector<int>> hs;
  vector<int> H;
  push_relabel(int sz) : g(sz), ec(sz), cur(sz), hs(2 * sz), H(
       sz) { n = sz; }
  void add_edge(int s, int t, int cap, int rcap, int id) {
    if (s == t) return;
    g[s].pb({t, (int)g[t].size(), 0, cap, id});
    g[t].pb({s, (int)g[s].size() - 1, 0, rcap, -1});
  }
  void add_flow(edge &e, int f) {
    edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f;
    e.c -= f;
    ec[e.dest] += f;
    back.f -= f;
    back.c += f;
    ec[back.dest] -= f;
  }
  int calc(int s, int t) {
```

```
    int v = g.size();
    H[s] = v;
    ec[t] = 1;
    vector<int> co(2 * v);
    co[0] = v - 1;
    for (int i = 0; i < v; i++) cur[i] = g[i].data();
    for (edge &e : g[s]) add_flow(e, e.c);
    for (int hi = 0;;) {
      while (hs[hi].empty())
        if (!hi--) return -ec[s];
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0) {
        if (cur[u] == g[u].data() + g[u].size()) {
          H[u] = INF;
          for (edge &e : g[u])
            if (e.c && H[u] > H[e.dest] + 1) H[u] = H[e.dest] +
                1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            for (int i = 0; i < v; i++)
              if (hi < H[i] && H[i] < v) --co[H[i]], H[i] = v +
                  1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest] + 1)
          add_flow(*cur[u], min(ec[u], cur[u]->c));
        else
          ++cur[u];
      }
    }
  }
  vector<int> flow_edges(int m)  // fluxo em cada aresta
  {
    vector<int> ans(m);
    for (int i = 0; i < n; i++) {
      for (auto const &j : g[i]) {
        if (j.id != -1) ans[j.id] = j.f;
      }
    }
    return ans;
  }
};
struct flow_with_demands {
  push_relabel pr;
  vector<int> in, out;
  int n;

  flow_with_demands(int sz) : n(sz), pr(sz + 2), in(sz), out(sz
      ) {}
  void add_edge(int u, int v, int cap, int dem, int id) {
    pr.add_edge(u, v, cap - dem, 0, id);
    out[u] += dem, in[v] += dem;
  }
  int run(int s, int t) {
    pr.add_edge(t, s, INF, 0, -1);
    for (int i = 0; i < n; i++) {
      pr.add_edge(n, i, in[i], 0, -1);
      pr.add_edge(i, n + 1, out[i], 0, -1);
    }
    return pr.calc(n, n + 1);
  }
  bool check()  // todas as constraints foram satisfeitas?
  {
    for (auto const &i : pr.g[n]) {
      if (i.c > 0) return 0;
    }
    return 1;
  }
};
```

hld.cpp

65 lines

```
struct hld {
  int n, cur_pos;
  segtree seg;
  vector<vector<int>> adj;
  vector<int> parent, depth, heavy, head, pos, sz;
  int dfs(int s) {
    int size = 1, max_c_size = 0;
    for (auto const &c : adj[s]) {
      if (c != parent[s]) {
        parent[c] = s;
        depth[c] = depth[s] + 1;
        int c_size = dfs(c);
        size += c_size;
        if (c_size > max_c_size) max_c_size = c_size, heavy[s]
            = c;
      }
    }
    return sz[s] = size;
  }
  void decompose(int s, int h) {
    head[s] = h;
    pos[s] = cur_pos++;
    if (heavy[s] != -1) decompose(heavy[s], h);
    for (int c : adj[s]) {
      if (c != parent[s] && c != heavy[s]) decompose(c, c);
    }
  }
  hld(vector<vector<int>> &g) {
    n = g.size();
    adj = g;
    seg = segtree(n);
    parent.assign(n, -1);
    depth.assign(n, -1);
    heavy.assign(n, -1);
    head.assign(n, -1);
    pos.assign(n, -1);
    sz.assign(n, 1);
    cur_pos = 0;
    dfs(0);
    decompose(0, 0);
  }
  int query_path(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
      if (depth[head[a]] > depth[head[b]]) swap(a, b);
      res += seg.query(0, n - 1, pos[head[b]], pos[b], 1);
    }
    if (depth[a] > depth[b]) swap(a, b);
    res += seg.query(0, n - 1, pos[a], pos[b], 1);
    return res;
  }
  void update_path(int a, int b, int x) {
    for (; head[a] != head[b]; b = parent[head[b]]) {
      if (depth[head[a]] > depth[head[b]]) swap(a, b);
      seg.update(1, 0, n - 1, pos[head[b]], pos[b], x);
    }
    if (depth[a] > depth[b]) swap(a, b);
    seg.update(1, 0, n - 1, pos[a], pos[b], x);
  }
  void update_subtree(int a, int x) { seg.update(1, 0, n - 1,
      pos[a], pos[a] + sz[a] - 1, x); }
  int query_subtree(int a) { return seg.query(0, n - 1, pos[a],
       pos[a] + sz[a] - 1, 1); }
  int lca(int a, int b) {
    if (pos[a] < pos[b]) swap(a, b);
    return (head[a] == head[b]) ? b : lca(parent[head[a]], b);
  }
};
```

hldedge.cpp

67 lines

```
namespace hld {
int cur_pos;
vector<int> parent, depth, heavy, head, pos, sz, up;
int dfs(int s) {
  int size = 1, max_c_size = 0;
  for (auto const &c : adj[s]) {
    if (c.fir != parent[s]) {
      parent[c.fir] = s;
      depth[c.fir] = depth[s] + 1;
      int c_size = dfs(c.fir);
      size += c_size;
      if (c_size > max_c_size) max_c_size = c_size, heavy[s] =
          c.fir;
    }
  }
  return sz[s] = size;
}
void decompose(int s, int h) {
  head[s] = h;
  pos[s] = cur_pos++;
  seg::v[pos[s]] = up[s];
  for (auto const &c : adj[s]) {
    if (c.fir != parent[s] && c.fir == heavy[s]) {
      up[c.fir] = c.sec;
      decompose(heavy[s], h);
    }
  }
  for (auto const &c : adj[s]) {
    if (c.fir != parent[s] && c.fir != heavy[s]) {
      up[c.fir] = c.sec;
      decompose(c.fir, c.fir);
    }
  }
}
void init() {
  parent.assign(MAXN, -1);
  depth.assign(MAXN, -1);
  heavy.assign(MAXN, -1);
  head.assign(MAXN, -1);
  pos.assign(MAXN, -1);
  sz.assign(MAXN, 1);
  up.assign(MAXN, 0);
  cur_pos = 0;
  dfs(0);
  decompose(0, 0);
  seg::build(0, n - 1, 1);
}
int query_path(int a, int b) {
  int res = -1;
  for (; head[a] != head[b]; b = parent[head[b]]) {
    if (depth[head[a]] > depth[head[b]]) swap(a, b);
    res = max(res, seg::query(0, n - 1, pos[head[b]], pos[b],
        1));
  }
  if (depth[a] > depth[b]) swap(a, b);
  res = max(res, seg::query(0, n - 1, pos[a] + 1, pos[b], 1));
  return res;
}
void update_path(int a, int b, int x) {
  for (; head[a] != head[b]; b = parent[head[b]]) {
    if (depth[head[a]] > depth[head[b]]) swap(a, b);
    seg::update(1, 0, n - 1, pos[head[b]], pos[b], x);
  }
  if (depth[a] > depth[b]) swap(a, b);
  seg::update(1, 0, n - 1, pos[a] + 1, pos[b], x);
```

```
}
void update_subtree(int a, int x) { seg::update(1, 0, n - 1,
     pos[a] + 1, pos[a] + sz[a] - 1, x); }
int query_subtree(int a, int x) { return seg::query(0, n - 1,
     pos[a] + 1, pos[a] + sz[a] - 1, 1); }
}  // namespace hld
```

strongorientation.cpp

25 lines

```
// encontrar uma orientacao para as arestas tal que o numero
// minimo de scc eh o menor possivel
// numero minimo de scc = numero de componentes conexas +
     numero de pontes
int n, m, timer, comps, bridges;
vector<pi> edges;
vector<pi> adj[MAXN];
int tin[MAXN];  // memset -1
int low[MAXN];  // memset -1
bool vis[MAXN];
char orient[MAXN];
void find_bridges(int v) {  // chama se tem tin == -1
  low[v] = timer, tin[v] = timer++;
  for (auto const &p : adj[v]) {
    if (vis[p.sec]) continue;
    vis[p.sec] = true;
    orient[p.sec] = (v == edges[p.sec].first) ? '>' : '<';
    if (tin[p.fir] == -1) {
      find_bridges(p.fir);
      low[v] = min(low[v], low[p.fir]);
      if (low[p.fir] > tin[v]) bridges++;
    } else {
      low[v] = min(low[v], low[p.fir]);
    }
  }
}
```

twosat.cpp

74 lines

```
struct two_sat {
  int n;
  vector<vector<int>> g, gr;   // gr is the reversed graph
  vector<int> comp, ord, ans;  // comp[v]: ID of the SCC
       containing node v
  vector<bool> vis;
  two_sat() {}
  two_sat(int sz) {
    n = sz;
    g.assign(2 * n, vector<int>());
    gr.assign(2 * n, vector<int>());
    comp.resize(2 * n);
    vis.resize(2 * n);
    ans.resize(2 * n);
  }
  void add_edge(int u, int v) {
    g[u].push_back(v);
    gr[v].push_back(u);
  }
  // int x, bool val: if 'val' is true, we take the variable to
       be x. Otherwise we take it to be x's complement (not x)
       .
  void implies(int i, bool f, int j, bool g)  // a -> b
  {
    add_edge(i + (f ? 0 : n), j + (g ? 0 : n));
    add_edge(j + (g ? n : 0), i + (f ? n : 0));
  }
  void add_clause_or(int i, bool f, int j, bool g)  // At least
       one of them is true
  {
    add_edge(i + (f ? n : 0), j + (g ? 0 : n));
    add_edge(j + (g ? n : 0), i + (f ? 0 : n));
  }
  void add_clause_xor(int i, bool f, int j, bool g)  // only
       one of them is true
  {
    add_clause_or(i, f, j, g);
    add_clause_or(i, !f, j, !g);
  }
  void add_clause_and(int i, bool f, int j, bool g)  // both of
       them have the same value
  {
    add_clause_xor(i, !f, j, g);
  }
  void set(int i, bool f)  // Set a variable
  {
    add_clause_or(i, f, i, f);
  }
  void top_sort(int u) {
    vis[u] = 1;
    for (auto const &v : g[u]) {
      if (!vis[v]) top_sort(v);
    }
    ord.push_back(u);
  }
  void scc(int u, int id) {
    vis[u] = 1;
    comp[u] = id;
    for (auto const &v : gr[u]) {
      if (!vis[v]) scc(v, id);
    }
  }
  bool solve() {
    fill(vis.begin(), vis.end(), 0);
    for (int i = 0; i < 2 * n; i++) {
      if (!vis[i]) top_sort(i);
    }
    fill(vis.begin(), vis.end(), 0);
    reverse(ord.begin(), ord.end());
    int id = 0;
    for (const auto &v : ord) {
      if (!vis[v]) scc(v, id++);
    }
    for (int i = 0; i < n; i++) {
      if (comp[i] == comp[i + n]) return 0;
      ans[i] = (comp[i] > comp[i + n]) ? 1 : 0;
    }
    return 1;
  }
};
```

sack.cpp

47 lines

```
vector<int> adj[MAXN];
vector<int> v[MAXN];
int c[MAXN];
int cnt[MAXN];
int sz[MAXN];
void dfs_sz(int x, int p) {
  sz[x] = 1;
  for (auto const &i : adj[x]) {
    if (i != p) {
      dfs_sz(i, x);
      sz[x] += sz[i];
    }
  }
}
void modify(int c, int val) { cnt[c] += val; }
void dfs(int x, int p, bool keep) {
  int best = -1, big_child = -1;
  for (auto const &i : adj[x]) {
    if (i != p && sz[i] > best) {
      best = sz[i];
      big_child = i;
    }
  }
  for (auto const &i : adj[x]) {
    if (i != p && i != big_child) dfs(i, x, 0);
  }
  if (big_child != -1) {
    dfs(big_child, x, 1);
    swap(v[x], v[big_child]);  // O(1)
  }
  v[x].pb(x);
  modify(c[x], 1);  // adiciona
  for (auto const &i : adj[x]) {
    if (i != p && i != big_child) {
      for (auto const &j : v[i]) {
        v[x].pb(j);
        modify(c[j], 1);  // adiciona
      }
    }
  }
  // a cor c aparece cnt[c] vezes na subtree de x
  // dai vc pode fazer algo tendo essa informacao
  // seja responser queries ou algo do tipo aqui
  if (!keep) {
    for (auto const &i : v[x]) modify(c[i], -1);  // remove
  }
}
```

reroot.cpp

23 lines

```
int n;
vector<int> adj[MAXN];
int sz[MAXN];
int dp[MAXN];
int dfs(int u, int v) {
  sz[u] = 1;
  for (auto const &i : adj[u])
    if (i != v) sz[u] += dfs(i, u);
  return sz[u];
}
void reroot(int u, int v) {
  for (auto const &i : adj[u]) {
    if (i != v) {
      int a = sz[u], b = sz[i];
      dp[i] = dp[u];
      dp[i] -= sz[u], dp[i] -= sz[i];
      sz[u] -= sz[i], sz[i] = n;
      dp[i] += sz[u], dp[i] += sz[i];
      reroot(i, u);
      sz[u] = a, sz[i] = b;
    }
  }
}
```

hungarian.cpp

45 lines

```
struct hungarian {  // declarar algo como hungarian h(n), n de
     cada lado
  int n, inf;
  vector<vector<int>> a;
  vector<int> u, v, p, way;
  hungarian(int n_) : n(n_), u(n + 1), v(n + 1), p(n + 1), way(
       n + 1) {
    a = vector<vector<int>>(n, vector<int>(n));
    inf = numeric_limits<int>::max();
  }
```

```
  void add_edge(int x, int y, int c) { a[x][y] = c; }
  pair<int, vector<int>> run() {
    for (int i = 1; i <= n; i++) {
      p[0] = i;
      int j0 = 0;
      vector<int> minv(n + 1, inf);
      vector<int> used(n + 1, 0);
      do {
        used[j0] = true;
        int i0 = p[j0], j1 = -1;
        int delta = inf;
        for (int j = 1; j <= n; j++) {
          if (!used[j]) {
            int cur = a[i0 - 1][j - 1] - u[i0] - v[j];
            if (cur < minv[j]) minv[j] = cur, way[j] = j0;
            if (minv[j] < delta) delta = minv[j], j1 = j;
          }
        }
        for (int j = 0; j <= n; j++) {
          if (used[j])
            u[p[j]] += delta, v[j] -= delta;
          else
            minv[j] -= delta;
        }
        j0 = j1;
      } while (p[j0] != 0);
      do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
      } while (j0);
    }
    vector<int> ans(n);
    for (int j = 1; j <= n; j++) ans[p[j] - 1] = j - 1;
    return make_pair(-v[0], ans);
  }
};
```

mincostmaxflow.cpp

64 lines

```
namespace mcf {
struct edge {
  int to, capacity, cost, res;
};
int source, destiny;
vector<edge> adj[MAXN];
vector<int> dist;
vector<int> parent;
vector<int> edge_index;
vector<bool> in_queue;
void add_edge(int a, int b, int c, int d) {
  adj[a].pb({b, c, d, (int)adj[b].size()});
  adj[b].pb({a, 0, -d, (int)adj[a].size() - 1});
}
bool dijkstra(int s) {
  dist.assign(MAXN, INF);
  parent.assign(MAXN, -1);
  edge_index.assign(MAXN, -1);
  in_queue.assign(MAXN, false);
  dist[s] = 0;
  queue<int> q;
  q.push(s);
  while (!q.empty()) {
    int u = q.front(), idx = 0;
    q.pop();
    in_queue[u] = false;
    for (auto const &v : adj[u]) {
      if (v.capacity && dist[v.to] > dist[u] + v.cost) {
        dist[v.to] = dist[u] + v.cost;
        parent[v.to] = u;
        edge_index[v.to] = idx;
        if (!in_queue[v.to]) {
          in_queue[v.to] = true;
          q.push(v.to);
        }
      }
      idx++;
    }
  }
  return dist[destiny] != INF;
}
int get_cost() {
  int flow = 0, cost = 0;
  while (dijkstra(source)) {
    int curr_flow = INF, curr = destiny;
    while (curr != source) {
      int p = parent[curr];
      curr_flow = min(curr_flow, adj[p][edge_index[curr]].
          capacity);
      curr = p;
    }
    flow += curr_flow;
    cost += curr_flow * dist[destiny];
    curr = destiny;
    while (curr != source) {
      int p = parent[curr];
      int res_idx = adj[p][edge_index[curr]].res;
      adj[p][edge_index[curr]].capacity -= curr_flow;
      adj[curr][res_idx].capacity += curr_flow;
      curr = p;
    }
  }
  return cost;
}
}  // namespace mcf
```

scc.cpp

54 lines

```
int n, m;
bool vis[MAXN];
int root[MAXN];
vector<int> order;
vector<int> roots;
vector<int> comp;
vector<vector<int>> comps;
vector<int> adj[MAXN];
vector<int> adj_rev[MAXN];
vector<int> adj_scc[MAXN];
void dfs(int v) {
  vis[v] = true;
  for (auto const &u : adj[v])
    if (!vis[u]) dfs(u);
  order.pb(v);
}
void dfs2(int v) {
  comp.pb(v);
  vis[v] = true;
  for (auto const &u : adj_rev[v])
    if (!vis[u]) dfs2(u);
}
//...
cin >> n >> m;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  adj[a].pb(b);
  adj_rev[b].pb(a);
}
for (int i = 0; i < n; i++) {
  if (!vis[i]) dfs(i);
}
reverse(order.begin(), order.end());
memset(vis, false, sizeof(vis));
for (auto const &v : order) {
  if (!vis[v]) {
    comp.clear();
    dfs2(v);
    comps.pb(comp);
    // making condensation graph
    int r = comp.back();
    for (auto const &u : comp) root[u] = r;
    roots.push_back(r);
  }
}
// making condensation graph
for (int v = 0; v < n; v++) {
  for (auto const &u : adj[v]) {
    int root_v = roots[v];
    int root_u = roots[u];
    if (root_u != root_v) adj_scc[root_v].pb(root_u);
  }
}
```

articulationpoints.cpp

23 lines

```
int n, m, timer;
vector<int> adj[MAXN];
bool is_cutpoint[MAXN];
int tin[MAXN];  // memset -1
int low[MAXN];  // memset -1
bool vis[MAXN];
void dfs(int v, int p) {  // chama pros nao vis
  vis[v] = true;
  tin[v] = timer, low[v] = timer++;
  int childs = 0;
  for (auto const &u : adj[v]) {
    if (u == p) continue;
    if (vis[u]) {
      low[v] = min(low[v], tin[u]);
    } else {
      dfs(u, v);
      low[v] = min(low[v], low[u]);
      if (low[u] >= tin[v] && p != -1) is_cutpoint[v] = true;
      childs++;
    }
  }
  if (p == -1 && childs > 1) is_cutpoint[v] = true;
}
```

notes.md

98 lines

```
## Bipartite Graph

A bipartite graph is a graph that does not contain any odd-
     length cycles.

## Directed acyclic graph (DAG)

Is a directed graph with no directed cycles.

## Independent Set

Is a set of vertices in a graph, no two of which are adjacent.
     That is, it is a set S of vertices such that for every two
      vertices in S, there is no edge connecting the two.

## Clique
```

```
Is a subset of vertices of an undirected graph such that every
     two distinct vertices in the clique are adjacent.

## Vertex Cover

Is a set of vertices that includes at least one endpoint of
     every edge of the graph.

## Edge Cover

Is a set of edges such that every vertex of the graph is
     incident to at least one edge of the set.

## Path Cover

Given a directed graph G = (V, E), a path cover is a set of
     directed paths such that every vertex v belongs to at
     least one path.

## Koning's Theorem

In any bipartite graph, the number of edges in a maximum
     matching equals the number of vertices in a minimum vertex
      cover.

## Properties

- Every tree is a bipartite graph.
- Any NxM grid is a bipartite graph.
- A set of vertices is a vertex cover if and only if its
     complement is an independent set.
- The number of vertices of a graph is equal to its minimum
     vertex cover number plus the size of a maximum independent
      set.
- In bipartite graphs, the size of the minimum edge cover is
     equal to the size of the maximum independent set
- In bipartite graphs, the size of the minimum edge cover plus
     the size of the minimum vertex cover is equal to the
     number of vertices.
- In bipartite graphs, maximum clique size is two.

## Min-cut

The smallest total weight of the edges which if removed would
     disconnect the source from the sink.

## Max-flow min-cut theorem

In a flow network, the maximum amount of flow passing from the
     source to the sink is equal to the total weight of the
     edges in a minimum cut.

## Maximum flow with vertex capacities

In other words, the amount of flow passing through a vertex
     cannot exceed its capacity. To find the maximum flow, we
     can transform the problem into the maximum flow problem by
      expanding the network. Each vertex v is replaced by v-in
     and v-out, where v-in is connected by edges going into v
     and v-out is connected to edges coming out from v. Then
     assign capacity c(v) to the edge connecting v-in and v-out
     .

## Undirected edge-disjoint paths problem

We are given an undirected graph G = (V, E) and two vertices s
     and t, and we have to find the maximum number of edge-
     disjoint s-t paths in G.

## Undirected vertex-disjoint paths problem

We are given an undirected graph G = (V, E) and two vertices s
     and t, and we have to find the maximum number of vertex-
     disjoint (except for s and t) paths in G.

## Menger's theorem

The maximum number of edge-disjoint s-t paths in an undirected
     graph is equal to the minimum number of edges in an s-t
     cut-set.

## Undirected vertex-disjoint paths solution

We can construct a network N=(V,E) from G with vertex
     capacities, where the capacities of all vertices and all
     edges are 1. Then the value of the maximum flow is equal
     to the maximum number of independent paths from s to t.

## Minimum vertex-disjoint path cover in directed acyclic graph
      (DAG)

Given a directed acyclic graph G=(V, E), we are to find the
     minimum number of vertex-disjoint paths to cover each
     vertex in V. We can construct a bipartite graph G' from G.
      Each vertex v is replaced by v-in and v-out, where v-in
     is connected by edges going into v and v-out is connected
     to edges coming out from v. Then it can be shown that G'
     has a matching M of size m if and only if G has a vertex-
     disjoint path cover C of containing m edges and n-m paths.

## Minimum general path cover in directed acyclic graph (DAG)

A general path cover is a path cover where a vertex can belong
     to more than one path. A minimum general path cover may be
      smaller than a minimum vertex-disjoint path cover. A
     minimum general path cover can be found almost like a
     minimum vertex-disjoint path cover. It suffices to add
     some new edges to the matching graph so that there is an
     edge a to b always when there is a path from a to b in the
      original graph.

## Dilworths theorem and maximum antichain

An antichain is a set of nodes of a graph such that there is no
      path from any node to another node using the edges of the
      graph. Dilworths theorem states that in a directed
     acyclic graph, the size of a minimum general path cover
     equals the size of a maximum antichain.

Or in other words: For a DAG G that if has edges from vertex i
     to vertex j and vertex j to k, then it also has a edge
     from vertex i to vertex k, the size of a minimum path
     cover is equal to the size of a maximum independent set.

## Maximum weighted antichain

In this problem, each vertex has a cost a[i]. The cost of an
     antichain is equal to the sum of the costs of the vertices
      present in it. We need to find the maximum cost of a
     antichain. We can construct the same bipartite of the
     maximum antichain problem from a dag G, these edges have
     an infinite capacity. We also need to create a source
     vertex and a sink, and we need to add edges source to v-in
      with capacity a[v] and v-out to sink with capacity a[v].
     The answer is equal to the sum of all a[i] minus the
     maximum flow on this network.

## Halls Theorem

Halls theorem can be used to find out whether a bipartite graph
      has a matching that contains all left or right nodes.
     Assume that we want to find a matching that contains all
     left nodes. Let X be any set of left nodes and let f(X) be
      the set of their neighbors. According to Halls theorem, a
      matching that contains all left nodes exists exactly when
      for
each X, the condition |X| <= |f(X)| holds.

## Extra (Getting Confidence Trick)

<p>If you need to maximize a number x = (a * b * c * ...), then
      you can write it as x = (e^log(a) * e^log(b) * e^log(c) *
      ...), and then the number is x = e^(log(a) + log(b) + log
     (c) + ...), and the problem now becomes a problem of
     maximizing the sum of (log(a) + log(b) + log(c) + ...).<p
     />

Use exp() and log() C++ functions :)
```

Geometry (5)

convexhull.cpp

47 lines

```
struct point {
  int x, y, id;
  point(int x, int y, int id) : x(x), y(y), id(id) {}
  point() {}
  point operator-(point const &o) const { return {x - o.x, y -
       o.y, -1}; }
  bool operator<(point const &o) const {
    if (x == o.x) return y < o.y;
    return x < o.x;
  }
  int operator^(point const &o) const { return x * o.y - y * o.
       x; }
};
int ccw(point const &a, point const &b, point const &x) {
  auto p = (b - a) ^ (x - a);
  return (p > 0) - (p < 0);
}
vector<point> convex_hull(vector<point> P)  // sem colineares
{
  sort(P.begin(), P.end());
  vector<point> L, U;
  for (auto p : P) {
    while (L.size() >= 2 && ccw(L.end()[-2], L.end()[-1], p) ==
         -1) L.pop_back();
    L.push_back(p);
  }
  reverse(P.begin(), P.end());
  for (auto p : P) {
    while (U.size() >= 2 && ccw(U.end()[-2], U.end()[-1], p) ==
         -1) U.pop_back();
    U.push_back(p);
  }
  L.insert(L.end(), U.begin(), U.end() - 1);
  return L;
}
vector<point> convex_hull_no_collinears(vector<point> P)  //
     com colineares
{
  sort(P.begin(), P.end());
  vector<point> L, U;
  for (auto p : P) {
```

```
    while (L.size() >= 2 && ccw(L.end()[-2], L.end()[-1], p) <=
          0) L.pop_back();
    L.push_back(p);
  }
  reverse(P.begin(), P.end());
  for (auto p : P) {
    while (U.size() >= 2 && ccw(U.end()[-2], U.end()[-1], p) <=
          0) U.pop_back();
    U.push_back(p);
  }
  L.insert(L.end(), U.begin(), U.end() - 1);
  return L;
}
```

linetrick.cpp — 13 lines

```
pi get_line(pi x, pi y) {  // um jeito normalizado de
     representar a reta entre 2 pontos
  int xx = x.fir - y.fir;
  int yy = x.sec - y.sec;
  int g = __gcd(abs(xx), abs(yy));
  if (g != 0) {
    xx /= g, yy /= g;
  }
  if (xx < 0) {
    xx *= -1;
    yy *= -1;
  }
  return {xx, yy};
}
```

polygonarea.cpp — 19 lines

```
double area(vector<pi> fig) {
  double res = 0;
  for (unsigned i = 0; i < fig.size(); i++) {
    pi p = i ? fig[i - 1] : fig.back();
    pi q = fig[i];
    res += (p.fir - q.fir) * (p.sec + q.sec);
  }
  return fabs(res) / 2;
}
int cross(pi a, pi b) { return a.fir * b.sec - a.sec * b.fir; }
double area2(vector<pi> fig) {
  double res = 0;
  for (unsigned i = 0; i < fig.size(); i++) {
    pi p = i ? fig[i - 1] : fig.back();
    pi q = fig[i];
    res += cross(p, q);
  }
  return fabs(res) / 2;
}
```

Strings (6)

Various (7)