Содержание

1 Setup & Scripts

1.1 CMake

```
cmake_minimum_required(VERSION 3.14)
project(olymp)

set(CMAKE_CXX_STANDARD 17)
add_compile_definitions(LOCAL)
#set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=undefined -fno-sanitize-recover")
#sanitizers: address, leak, thread, undefined, memory

add_executable(olymp f.cpp)
```

1.2 wipe.sh

```
touch {a..l}.cpp

for file in ?.cpp ; do
    cat template.cpp > $file ;
done
```

1.3 Builtins

2 G++ builtins

- __builtin_popcount(x) — количество единичных бит в двоичном представлении 32-битного (знакового или беззнакового) целого числа.
- __builtin_popcountll(x) — то же самое для 64-битных типов.
- __builtin_ctz(x) — количество нулей на конце двоичного представления 32-битного целого числа. Например, для 5 вернётся 0, для $272 = 256 + 16$ — 4 и т. д. Может не работать для нуля (вообще не стоит вызывать для $x = 0$, по-моему это и упасть может).
- __builtin_ctzll(x) — то же самое для 64-битных типов.
- __builtin_clz(x) — количество нулей в начале двоичного представления 32-битного целого числа. Например, для 2^{31} или -2^{31} вернётся 0, для 1 — 31 и т. д. Тоже не надо вызвывать с $x = 0$.
- __builtin_clzll(x) — то же самое для 64-битных типов.
- bitset<N>._Find_first() — номер первой позиции с единицей в битсете или его размер (то есть N), если на всех позициях нули.
- bitset<N>._Find_next(x) — номер первой позиции с единицей среди позиций с номерами строго больше x; если такой нет, то N.

3 Bugs

- powmod :)
- Всегда чекать Куна дважды, особенно на количество итераций
- uniform_int_distribution от одного параметра
- for (char c : ”NEWS”)
- Порядок верхних и нижних границ в случае, когда задача двумерна $t - b \neq b - t$
- static с мультитестами
- set со своим компаратором склеивает элементы

4 Geometry

4.1 Пересечение прямых

$$AB = A - B; CD = C - D$$

$$(A \times B \cdot CD.x - C \times D \cdot AB.x : A \times B \cdot CD.y - C \times D \cdot AB.y : AB \times CD)$$

5 Numbers

- A lot of divisors
  - $\leq 20 : d(12) = 6$
  - $\leq 50 : d(48) = 10$
  - $\leq 100 : d(60) = 12$
  - $\leq 1000 : d(840) = 32$
  - $\leq 10^4 : d(9240) = 64$
  - $\leq 10^5 : d(83160) = 128$
  - $\leq 10^6 : d(720720) = 240$
  - $\leq 10^7 : d(8648640) = 448$
  - $\leq 10^8 : d(91891800) = 768$
  - $\leq 10^9 : d(931170240) = 1344$
  - $\leq 10^{11} : d(97772875200) = 4032$
  - $\leq 10^{12} : d(963761198400) = 6720$
  - $\leq 10^{15} : d(866421317361600) = 26880$
  - $\leq 10^{18} : d(897612484786617600) = 103680$
- Numeric integration
  - simple: $F(0)$
  - simpson: $\frac{F(-1)+4\cdot F(0)+F(1)}{6}$
  - runge2: $\frac{F(-\sqrt{\frac{1}{3}})+F(\sqrt{\frac{1}{3}})}{2}$
  - runge3: $\frac{F(-\sqrt{\frac{3}{5}})\cdot 5+F(0)\cdot 8+F(\sqrt{\frac{3}{5}})\cdot 5}{18}$

6 Graphs

6.1 Weighted matroid intersection

```
// here we use T = __int128 to store the independent set
// calling expand k times to an empty set finds the maximum
// cost of the set with size exactly k,
// that is independent in blue and red matroids
// ver is the number of the elements in the matroid,
// e[i].w is the cost of the i-th element
```

```
// first return value is new independent set
// second return value is difference between
// new and old costs
// oracle(set, red) and oracle(set, blue) check whether
// or not the set lies in red or blue matroid respectively

auto expand = [&] (T cur_set) -> pair<T, int>
{
    vector<int> in(ver);
    for (int i = 0; i < ver; i++)
        in[i] = ((cur_set >> i) & 1);

    const int red = 1;
    const int blue = 2;

    vector<vector<int>> g(ver);
    for (int i = 0; i < ver; i++)
    for (int j = 0; j < ver; j++)
    {
        T swp_mask = (cur_set ^ (T(1) << i) ^ (T(1) << j));
        if (!in[i] && in[j])
        {
            if (oracle(swp_mask, red))
                g[i].push_back(j);
            if (oracle(swp_mask, blue))
                g[j].push_back(i);
        }
    }

    vector<int> from, to;
    for (int i = 0; i < ver; i++) if (!in[i])
    {
        T add_mask = cur_set ^ (T(1) << i);
        if (oracle(add_mask, blue))
            from.push_back(i);
        if (oracle(add_mask, red))
            to.push_back(i);
    }

    auto get_cost = [&] (int x)
    {
        const int cost = (!in[x] ? e[x].w : -e[x].w);
        return (ver + 1) * cost - 1;
    };

    const int inf = int(1e9);
    vector<int> dist(ver, -inf), prev(ver, -1);
    for (int x : from)
        dist[x] = get_cost(x);

```

```
    queue<int> q;

    vector<int> used(ver);
    for (int x : from)
    {
        q.push(x);
        used[x] = 1;
    }

    while (!q.empty())
    {
        int cur = q.front(); used[cur] = 0; q.pop();

        for (int to : g[cur])
        {
            int cost = get_cost(to);
            if (dist[to] < dist[cur] + cost)
            {
                dist[to] = dist[cur] + cost;
                prev[to] = cur;
                if (!used[to])
                {
                    used[to] = 1;
                    q.push(to);
                }
            }
        }
    }

    int best = -inf, where = -1;
    for (int x : to)
    {
        if (dist[x] > best)
        {
            best = dist[x];
            where = x;
        }
    }

    if (best == -inf)
        return pair<T, int>(cur_set, best);

    while (where != -1)
    {
        cur_set ^= (T(1) << where);
        where = prev[where];
    }

    while (best % (ver + 1))
        best++;
```

```
    best /= (ver + 1);

    assert(oracle(cur_set, red) && oracle(cur_set, blue));
    return pair<T, int>(cur_set, best);
};
```

7 Push-free segment tree

```
class pushfreesegtree
{
    vector<modulo<>> pushed, unpushed;

    modulo<> add(int l, int r, int cl, int cr, int v, const modulo<> &x)
    {
        if (r <= cl || cr <= l)
            return 0;
        if (l <= cl && cr <= r)
        {
            unpushed[v] += x;

            return x * (cr - cl);
        }

        int ct = (cl + cr) / 2;

        auto tmp = add(l, r, cl, ct, 2 * v, x) + add(l, r, ct, cr, 2 * v + 1, x);

        pushed[v] += tmp;

        return tmp;
    }


    modulo<> sum(int l, int r, int cl, int cr, int v)
    {
        if (r <= cl || cr <= l)
            return 0;
        if (l <= cl && cr <= r)
            return pushed[v] + unpushed[v] * (cr - cl);

        int ct = (cl + cr) / 2;

        return sum(l, r, cl, ct, 2 * v) + unpushed[v] * (min(r, cr) - max(l, cl)) + sum(l, r, ct, cr, 2 * v + 1);
    }

public:
    pushfreesegtree(int n) : pushed(2 * up(n)), unpushed(2 * up(n))
```

```
    {}


    modulo<> sum(int l, int r)
    {
        return sum(l, r, 0, pushed.size() / 2, 1);
    }


    void add(int l, int r, const modulo<> &x)
    {
        add(l, r, 0, pushed.size() / 2, 1, x);
    }
};
```

8 Number theory

8.1 Chinese remainder theorem without overflows

```
// Replace T with an appropriate type!
using T = long long;

// Finds x, y such that ax + by = gcd(a, b).
T gcdext (T a, T b, T &x, T &y)
{
    if (b == 0)
    {
        x = 1, y = 0;
        return a;
    }

    T res = gcdext (b, a % b, y, x);
    y -= x * (a / b);
    return res;
}

// Returns true if system x = r1 (mod m1), x = r2 (mod m2) has solutions
// false otherwise. In first case we know exactly that x = r (mod m )

bool crt (T r1, T m1, T r2, T m2, T &r, T &m)
{
    if (m2 > m1)
    {
        swap(r1, r2);
        swap(m1, m2);
    }

    T g = __gcd(m1, m2);
```

```cpp
    if ((r2 - r1) % g != 0)
        return false;

    T c1, c2;
    auto nrem = gcdext(m1 / g, m2 / g, c1, c2);
    assert(nrem == 1);
    assert(c1 * (m1 / g) + c2 * (m2 / g) == 1);
    T a = c1;
    a *= (r2 - r1) / g;
    a %= (m2 / g);
    m = m1 / g * m2;
    r = a * m1 + r1;
    r = r % m;
    if (r < 0)
        r += m;

    assert(r % m1 == r1 && r % m2 == r2);
    return true;
}
```

8.2 Integer points under a rational line

```cpp
// integer (x, y) : 0 <= x < n, 0 < y <= (kx + b) / d
// (real division)
// In other words, sum_{x=0}^{n-1} [(kx+b)/d]
ll trapezoid (ll n, ll k, ll b, ll d)
{
    if (k == 0)
        return (b / d) * n;
    if (k >= d || b >= d)
        return (k / d) * n * (n - 1) / 2 + (b / d) * n + trapezoid(n, k % d, b % d, d);
    return trapezoid((k * n + b) / d, d, (k * n + b) % d, k);
}
```

9 Suffix Automaton

```cpp
struct Vx{
    static const int AL = 26;
    int len, suf;
    int next[AL];
    Vx(){}
    Vx(int l, int s):len(l), suf(s){}
};

struct SA{
    static const int MAX_LEN = 1e5 + 100, MAX_V = 2 * MAX_LEN;
    int last, vcnt;
    Vx v[MAX_V];

    SA(){
```

```
        vcnt = 1;
        last = newV(0, 0); // root = vertex with number 1
    }
    int newV(int len, int suf){
        v[vcnt] = Vx(len, suf);
        return vcnt++;
    }

    int add(char ch){
        int p = last, c = ch - 'a';
        last = newV(v[last].len + 1, 0);
        while(p && !v[p].next[c]) //added p &&
            v[p].next[c] = last, p = v[p].suf;
        if(!p)
            v[last].suf = 1;
        else{
            int q = v[p].next[c];
            if (v[q].len == v[p].len + 1)
                v[last].suf = q;
            else{
                int r = newV(v[p].len + 1, v[q].suf);
                v[last].suf = v[q].suf = r;
                memcpy(v[r].next, v[q].next, sizeof(v[r].next));
                while(p && v[p].next[c] == q)
                    v[p].next[c] = r, p = v[p].suf;
            }
        }
        return last;
    }
};
```

10 Smth added at last moment

10.1 Dominator Tree

```
struct dom_tree {
  vvi g, rg, tree, bucket;
  vi sdom, par, dom, dsu, label, in, order, tin, tout;
  int T = 0, root = 0, n = 0;

  void dfs_tm (int x) {
    in[x] = T;
    order[T] = x;
    label[T] = T, sdom[T] = T, dsu[T] = T, dom[T] = T;
    T++;
    for (int to : g[x]) {
      if (in[to] == -1) {
        dfs_tm(to);
        par[in[to]] = in[x];
      }
```

```
    rg[in[to]].pb(in[x]);
  }
}

void dfs_tree (int v, int p) {
  tin[v] = T++;
  for (int dest : tree[v]) {
    if (dest != p) {
      dfs_tree(dest, v);
    }
  }
  tout[v] = T;
}

dom_tree (const vvi &g_, int root_) {
  g = g_;
  n = sz(g);
  assert(0 <= root && root < n);
  in.assign(n, -1);
  rg.resize(n);
  order = sdom = par = dom = dsu = label = vi(n);
  root = root_;
  bucket.resize(n);
  tree.resize(n);

  dfs_tm(root);

  for (int i = n - 1; i >= 0; i--) {
    for (int j : rg[i])
      sdom[i] = min(sdom[i], sdom[find(j)]);
    if (i > 0)
      bucket[sdom[i]].pb(i);

    for (int w : bucket[i]) {
      int v = find(w);
      dom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
    }

    if (i > 0)
      unite(par[i], i);
  }

  for (int i = 1; i < n; i++) {
    if (dom[i] != sdom[i])
      dom[i] = dom[dom[i]];
    tree[order[i]].pb(order[dom[i]]);
    tree[order[dom[i]]].pb(order[i]);
  }

  T = 0;
```

```cpp
    tin = tout = vi(n);
    dfs_tree(root, -1);
  }

  void unite (int u, int v) {
    dsu[v] = u;
  }

  int find (int u, int x = 0) {
    if (u == dsu[u])
      return (x ? -1 : u);
    int v = find(dsu[u], x + 1);
    if (v == -1)
      return u;
    if (sdom[label[dsu[u]]] < sdom[label[u]])
      label[u] = label[dsu[u]];
    dsu[u] = v;
    return (x ? v : label[u]);
  }

  bool dominated_by (int v, int by_what) {
    return tin[by_what] <= tin[v] && tout[v] <= tout[by_what];
  }
};
```

10.2 Suffix Array

```cpp
namespace suff_arr {

const int MAXN = 2e5 + 10;

string s;
int n;
int p[MAXN];
int lcp[MAXN];
int pos[MAXN];
int c[MAXN];

void print() {
#ifndef LOCAL
    return;
#endif
    eprintf("p:\n");
    forn(i, sz(s)) {
        eprintf("i=%d -- %d: %s, lcp=%d, c=%d\n", i, p[i], s.substr(p[i], sz(s) - p[i]).data(), lcp[i], c[p[i]]);
    }
    eprintf("\n");
}

void build(const string& s_) {
```

```
    static int cnt[MAXN];
    static int np[MAXN];
    static int nc[MAXN];

    s = s_;
    n = sz(s);

    memset (cnt, 0, sizeof cnt);
    for (char ch : s) {
        ++cnt[int(ch)];
    }
    forn(i, 256) {
        cnt[i + 1] += cnt[i];
    }
    forn(i, sz(s)) {
        p[--cnt[int(s[i])]] = i;
    }

    int cls = 1;
    c[p[0]] = cls - 1;
    for (int i = 1; i < n; ++i) {
        if (s[p[i]] != s[p[i - 1]]) {
            ++cls;
        }
        c[p[i]] = cls - 1;
    }

    for (int len = 1; len <= n; len *= 2) {
        memset (cnt, 0, sizeof(int) * cls);
        forn(i, n) {
            ++cnt[c[i]];
        }
        forn(i, cls - 1) {
            cnt[i + 1] += cnt[i];
        }
        ford(i, n) {
            const int j = p[i];
            int j2 = (j - len + n) % n;
            np[--cnt[c[j2]]] = j2;
        }
        memcpy(p, np, sizeof(int) * n);

        cls = 1;
        nc[p[0]] = cls - 1;
        for (int i = 1; i < n; ++i) {
            if (c[p[i]] != c[p[i - 1]] || c[(p[i] + len) % n] != c[(p[i - 1] + len) % n]) {
                ++cls;
            }
            nc[p[i]] = cls - 1;
```

```
        }
        memcpy(c, nc, sizeof(int) * n);
    }

    forn(i, n) {
        pos[p[i]] = i;
    }

    int pref = 0;
    forn(i, n) {
        int pi = pos[i];
        if (pi == n - 1) {
            continue;
        }
        int j = p[pi + 1];
        while (i + pref < n && j + pref < n && s[i + pref] == s[j + pref]) {
            ++pref;
        }
        lcp[pi] = pref;
        pref = max(0, pref - 1);
    }

//         print();
}

};
```

10.3 Fast LCS

```
// assumes that strings consist of lowercase latin letters
const int M = ((int)1e5 + 64) / 32 * 32;
// maximum value of m
using bs = bitset<M>;
using uint = unsigned int;
const ll bnd = (1LL << 32);

// WARNING: invokes undefined behaviour of modifying ans through pointer to another data type (uint)
// seems to work, but be wary
bs sum (const bs &bl, const bs &br)
{
    const int steps = M / 32;
    const uint* l = (uint*)&bl;
    const uint* r = (uint*)&br;

    bs ans;
    uint* res = (uint*)&ans;

    int carry = 0;
    forn (i, steps)
```

```
    {
        ll cur = ll(*l++) + ll(*r++) + carry;
        carry = (cur >= bnd);
        cur = (cur >= bnd ? cur - bnd : cur);
        *res++ = uint(cur);
    }

    return ans;
}

int fast_lcs (const string &s, const string &t)
{
    const int m = sz(t);
    const int let = 26;

    vector<bs> has(let);
    vector<bs> rev = has;

    forn (i, m)
    {
        const int pos = t[i] - 'a';
        has[pos].set(i);
        forn (j, let) if (j != pos)
            rev[j].set(i);
    }

    bs row;
    forn (i, m)
        row.set(i);

    int cnt = 0;
    for (char ch : s)
    {
        const int pos = ch - 'a';

        bs next = sum(row, row & has[pos]) | (row & rev[pos]);
        cnt += next[m];
        next[m] = 0;

        row = next;
    }

    return cnt;
}
```

10.4 Fast Subset Convolution

```
// algorithm itself starts here
void mobius (int* a, int n, int sign)
{
    forn (i, n)
```

```
    {
        int free = ((1 << n) - 1) ^ (1 << i);
        for (int mask = free; mask > 0; mask = ((mask - 1) & free))
            (sign == +1 ? add : sub)(a[mask ^ (1 << i)], a[mask]);
        add(a[1 << i], a[0]);
    }
}

// maximum number of bits allowed
const int B = 20;

vi fast_conv (vi a, vi b)
{
    assert(!a.empty());
    const int bits = __builtin_ctz(sz(a));
    assert(sz(a) == (1 << bits) && sz(a) == sz(b));

    static int trans_a[B + 1][1 << B];
    static int trans_b[B + 1][1 << B];
    static int trans_res[B + 1][1 << B];

    forn (cnt, bits + 1)
    {
        for (auto cur : {trans_a, trans_b, trans_res})
            fill(cur[cnt], cur[cnt] + (1 << bits), 0);
    }

    forn (mask, 1 << bits)
    {
        const int cnt = __builtin_popcount(mask);
        trans_a[cnt][mask] = a[mask];
        trans_b[cnt][mask] = b[mask];
    }

    forn (cnt, bits + 1)
    {
        mobius(trans_a[cnt], bits, +1);
        mobius(trans_b[cnt], bits, +1);
    }

    // Not really a valid ranked mobius transform! But algorithm works anyway

    forn (i, bits + 1) forn (j, bits - i + 1) forn (mask, 1 << bits)
        add(trans_res[i + j][mask], mult(trans_a[i][mask], trans_b[j][mask]));

    forn (cnt, bits + 1)
        mobius(trans_res[cnt], bits, -1);
```

```

    forn (mask, 1 << bits)
    {
        const int cnt = __builtin_popcount(mask);
        a[mask] = trans_res[cnt][mask];
    }

    return a;
}
```

11 Karatsuba

```
// functon Karatsuba (and stupid as well) computes c += a * b, not c = a * b

using hvect = vector<modulo<>>::iterator;
using hcvect = vector<modulo<>>::const_iterator;


void add(hcvect abegin, hcvect aend, hvect ans)
{
    for (auto it = abegin; it != aend; ++it, ++ans)
        *ans += *it;
}


void sub(hcvect abegin, hcvect aend, hvect ans)
{
    for (auto it = abegin; it != aend; ++it, ++ans)
        *ans -= *it;
}


void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans)
{
    for (auto a = abegin; a != abegin + siz; ++a, ans -= (siz - 1))
        for (auto b = bbegin; b != bbegin + siz; ++b, ++ans)
            *ans += *a * *b;
}


void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin, hvect ans, hvect small, hvect big, hvect sum)
{
    assert((siz & (siz - 1)) == 0);

    if (siz <= 32)
    {
        stupid(siz, abegin, bbegin, ans);

```

```
        return;
    }

    auto amid = abegin + siz / 2, aend = abegin + siz;
    auto bmid = bbegin + siz / 2, bend = bbegin + siz;
    auto smid = sum + siz / 2, send = sum + siz;

    fill(small, small + siz, 0);
    Karatsuba(siz / 2, abegin, bbegin, small, small + siz, big + siz, sum);
    fill(big, big + siz, 0);
    Karatsuba(siz / 2, amid, bmid, big, small + siz, big + siz, sum);

    copy(abegin, amid, sum);
    add(amid, aend, sum);
    copy(bbegin, bmid, sum + siz / 2);
    add(bmid, bend, sum + siz / 2);

    Karatsuba(siz / 2, sum, smid, ans + siz / 2, small + siz, big + siz, send);

    add(small, small + siz, ans);
    sub(small, small + siz, ans + siz / 2);
    add(big, big + siz, ans + siz);
    sub(big, big + siz, ans + siz / 2);
}


void mult(vector<modulo<>> a, vector<modulo<>> b, vector<modulo<>> &c)
{
    a.resize(up(max(a.size(), b.size())), 0);
    b.resize(a.size(), 0);

    c.assign(a.size() * 2, 0);

    auto small = c;
    auto big = c;
    auto sum = c;

    Karatsuba(a.size(), a.begin(), b.begin(), c.begin(), small.begin(), big.begin(), sum.begin());
}
```