Содержание

1 Setup & Scripts

1.1 CMake

```
1 cmake_minimum_required(VERSION 3.14)
2 project(olymp)
3
4 set(CMAKE_CXX_STANDARD 17)
5 add_compile_definitions(LOCAL)
6 #set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=undefined -fno-
    ↪ sanitize-recover")
7 #sanitizers: address, leak, thread, undefined, memory
8
9 add_executable(olymp f.cpp)
```

1.2 wipe.sh

```
1 touch {a..l}.cpp
2
3 for file in ?.cpp ; do
```

```
4        cat template.cpp > $file ;
5    done
```

2   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

3   Geometry

3.1   Пересечение прямых

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

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

4   Numbers

- A lot of divisors

  - $\leq 20 : d(12) = 6$
  - $\leq 50 : d(48) = 10$
  - $\leq 100 : d(60) = 12$
  - $\leq 1000 : d(840) = 32$
  - $\leq 10^4 : d(9240) = 64$
  - $\leq 10^5 : d(83160) = 128$
  - $\leq 10^6 : d(720720) = 240$
  - $\leq 10^7 : d(8648640) = 448$
  - $\leq 10^8 : d(91891800) = 768$
  - $\leq 10^9 : d(931170240) = 1344$
  - $\leq 10^{11} : d(97772875200) = 4032$
  - $\leq 10^{12} : d(963761198400) = 6720$
  - $\leq 10^{15} : d(86642317361600) = 26880$
  - $\leq 10^{18} : d(897612484786617600) = 103680$

- 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: $\frac{F(-\sqrt{\frac{3}{5}})\cdot 5+F(0)\cdot 8+F(\sqrt{\frac{3}{5}})\cdot 5}{18}$

5 Graphs

5.1 Weighted matroid intersection

```
1  // here we use T = __int128 to store the independent set
2  // calling expand k times to an empty set finds the maximum
3  // cost of the set with size exactly k,
4  // that is independent in blue and red matroids
5  // ver is the number of the elements in the matroid,
6  // e[i].w is the cost of the i-th element
7  // first return value is new independent set
8  // second return value is difference between
9  // new and old costs
10 // oracle(set, red) and oracle(set, blue) check whether
11 // or not the set lies in red or blue matroid respectively
12
13 auto expand = [&] (T cur_set) → pair<T, int>
14 {
15     vector<int> in(ver);
16     for (int i = 0; i < ver; i++)
17         in[i] = ((cur_set >> i) & 1);
18
19     const int red = 1;
20     const int blue = 2;
21
22     vector<vector<int>> g(ver);
23     for (int i = 0; i < ver; i++)
24     for (int j = 0; j < ver; j++)
25     {
26         T swp_mask = (cur_set ^ (T(1) << i) ^ (T(1) << j));
27         if (!in[i] && in[j])
28         {
29             if (oracle(swp_mask, red))
30                 g[i].push_back(j);
31             if (oracle(swp_mask, blue))
32                 g[j].push_back(i);
33         }
34     }
35
36     vector<int> from, to;
37     for (int i = 0; i < ver; i++) if (!in[i])
38     {
39         T add_mask = cur_set ^ (T(1) << i);
40         if (oracle(add_mask, blue))
```

```
41              from.push_back(i);
42          if (oracle(add_mask, red))
43              to.push_back(i);
44      }
45
46      auto get_cost = [&] (int x)
47      {
48          const int cost = (!in[x] ? e[x].w : -e[x].w);
49          return (ver + 1) * cost - 1;
50      };
51
52      const int inf = int(1e9);
53      vector<int> dist(ver, -inf), prev(ver, -1);
54      for (int x : from)
55          dist[x] = get_cost(x);
56
57      queue<int> q;
58
59      vector<int> used(ver);
60      for (int x : from)
61      {
62          q.push(x);
63          used[x] = 1;
64      }
65
66      while (!q.empty())
67      {
68          int cur = q.front(); used[cur] = 0; q.pop();
69
70          for (int to : g[cur])
71          {
72              int cost = get_cost(to);
73              if (dist[to] < dist[cur] + cost)
74              {
75                  dist[to] = dist[cur] + cost;
76                  prev[to] = cur;
77                  if (!used[to])
78                  {
79                      used[to] = 1;
80                      q.push(to);
81                  }
82              }
83          }
84      }
85
86      int best = -inf, where = -1;
87      for (int x : to)
88      {
89          if (dist[x] > best)
90          {
```

```
 91                 best = dist[x];
 92                 where = x;
 93             }
 94         }
 95
 96     if (best == -inf)
 97         return pair<T, int>(cur_set, best);
 98
 99     while (where ≠ -1)
100     {
101         cur_set ^= (T(1) << where);
102         where = prev[where];
103     }
104
105     while (best % (ver + 1))
106         best++;
107     best /= (ver + 1);
108
109     assert(oracle(cur_set, red) && oracle(cur_set, blue));
110     return pair<T, int>(cur_set, best);
111 };
```

6 Push-free segment tree

```
 1 class pushfreesegtree
 2 {
 3     vector<modulo<>> pushed, unpushed;
 4
 5     modulo<> add(int l, int r, int cl, int cr, int v, const modulo
       ↪ <> &x)
 6     {
 7         if (r ≤ cl || cr ≤ l)
 8             return 0;
 9         if (l ≤ cl && cr ≤ r)
10         {
11             unpushed[v] += x;
12
13             return x * (cr - cl);
14         }
15
16         int ct = (cl + cr) / 2;
17
18         auto tmp = add(l, r, cl, ct, 2 * v, x) + add(l, r, ct, cr,
       ↪ 2 * v + 1, x);
19
20         pushed[v] += tmp;
21
22         return tmp;
23     }
24
```

```
25
26      modulo◇ sum(int l, int r, int cl, int cr, int v)
27      {
28          if (r ⩽ cl || cr ⩽ l)
29              return 0;
30          if (l ⩽ cl && cr ⩽ r)
31              return pushed[v] + unpushed[v] * (cr - cl);
32
33          int ct = (cl + cr) / 2;
34
35          return sum(l, r, cl, ct, 2 * v) + unpushed[v] * (min(r, cr)
                ↪ - max(l, cl)) + sum(l, r, ct, cr, 2 * v + 1);
36      }
37
38  public:
39      pushfreesegtree(int n) : pushed(2 * up(n)), unpushed(2 * up(n))
40      {}
41
42
43      modulo◇ sum(int l, int r)
44      {
45          return sum(l, r, 0, pushed.size() / 2, 1);
46      }
47
48
49      void add(int l, int r, const modulo◇ &x)
50      {
51          add(l, r, 0, pushed.size() / 2, 1, x);
52      }
53  };
```

7   Number theory

7.1   Chinese remainder theorem without overflows

```
1  // Replace T with an appropriate type!
2  using T = long long;
3
4  // Finds x, y such that ax + by = gcd(a, b).
5  T gcdext (T a, T b, T &x, T &y)
6  {
7      if (b == 0)
8      {
9          x = 1, y = 0;
10         return a;
11     }
12
13     T res = gcdext (b, a % b, y, x);
14     y -= x * (a / b);
15     return res;
```

```
16  }
17
18  // Returns true if system x = r1 (mod m1), x = r2 (mod m2) has
        ↪ solutions
19  // false otherwise. In first case we know exactly that x = r (mod m
        ↪ )
20
21  bool crt (T r1, T m1, T r2, T m2, T &r, T &m)
22  {
23      if (m2 > m1)
24      {
25          swap(r1, r2);
26          swap(m1, m2);
27      }
28
29      T g = __gcd(m1, m2);
30      if ((r2 - r1) % g ≠ 0)
31          return false;
32
33      T c1, c2;
34      auto nrem = gcdext(m1 / g, m2 / g, c1, c2);
35      assert(nrem == 1);
36      assert(c1 * (m1 / g) + c2 * (m2 / g) == 1);
37      T a = c1;
38      a *= (r2 - r1) / g;
39      a %= (m2 / g);
40      m = m1 / g * m2;
41      r = a * m1 + r1;
42      r = r % m;
43      if (r < 0)
44          r += m;
45
46      assert(r % m1 == r1 && r % m2 == r2);
47      return true;
48  }
```

7.2 Integer points under a rational line

```
1  // integer (x, y) : 0 ≤ x < n, 0 < y ≤ (kx + b) / d
2  // (real division)
3  // In other words, sum_{x=0}^{n-1} [(kx+b)/d]
4  ll trapezoid (ll n, ll k, ll b, ll d)
5  {
6      if (k == 0)
7          return (b / d) * n;
8      if (k ≥ d || b ≥ d)
9          return (k / d) * n * (n - 1) / 2 + (b / d) * n + trapezoid(
                ↪ n, k % d, b % d, d);
10     return trapezoid((k * n + b) / d, d, (k * n + b) % d, k);
11 }
```

8   Suffix Automaton

```
1  struct Vx{
2      static const int AL = 26;
3      int len, suf;
4      int next[AL];
5      Vx(){}
6      Vx(int l, int s):len(l), suf(s){}
7  };
8
9  struct SA{
10     static const int MAX_LEN = 1e5 + 100, MAX_V = 2 * MAX_LEN;
11     int last, vcnt;
12     Vx v[MAX_V];
13
14     SA(){
15         vcnt = 1;
16         last = newV(0, 0); // root = vertex with number 1
17     }
18     int newV(int len, int suf){
19         v[vcnt] = Vx(len, suf);
20         return vcnt++;
21     }
22
23     int add(char ch){
24         int p = last, c = ch - 'a';
25         last = newV(v[last].len + 1, 0);
26         while(p && !v[p].next[c]) //added p &&
27             v[p].next[c] = last, p = v[p].suf;
28         if(!p)
29             v[last].suf = 1;
30         else{
31             int q = v[p].next[c];
32             if (v[q].len == v[p].len + 1)
33                 v[last].suf = q;
34             else{
35                 int r = newV(v[p].len + 1, v[q].suf);
36                 v[last].suf = v[q].suf = r;
37                 memcpy(v[r].next, v[q].next, sizeof(v[r].next));
38                 while(p && v[p].next[c] == q)
39                     v[p].next[c] = r, p = v[p].suf;
40             }
41         }
42         return last;
43     }
44 };
```

9   Smth added at last moment

9.1   Dominator Tree

```cpp
1   struct dom_tree {
2     vvi g, rg, tree, bucket;
3     vi sdom, par, dom, dsu, label, in, order, tin, tout;
4     int T = 0, root = 0, n = 0;
5
6     void dfs_tm (int x) {
7       in[x] = T;
8       order[T] = x;
9       label[T] = T, sdom[T] = T, dsu[T] = T, dom[T] = T;
10      T++;
11      for (int to : g[x]) {
12        if (in[to] == -1) {
13          dfs_tm(to);
14          par[in[to]] = in[x];
15        }
16        rg[in[to]].pb(in[x]);
17      }
18    }
19
20    void dfs_tree (int v, int p) {
21      tin[v] = T++;
22      for (int dest : tree[v]) {
23        if (dest != p) {
24          dfs_tree(dest, v);
25        }
26      }
27      tout[v] = T;
28    }
29
30    dom_tree (const vvi &g_, int root_) {
31      g = g_;
32      n = sz(g);
33      assert(0 <= root && root < n);
34      in.assign(n, -1);
35      rg.resize(n);
36      order = sdom = par = dom = dsu = label = vi(n);
37      root = root_;
38      bucket.resize(n);
39      tree.resize(n);
40
41      dfs_tm(root);
42
43      for (int i = n - 1; i >= 0; i--) {
44        for (int j : rg[i])
45          sdom[i] = min(sdom[i], sdom[find(j)]);
46        if (i > 0)
47          bucket[sdom[i]].pb(i);
48
49        for (int w : bucket[i]) {
50          int v = find(w);
```

```
51            dom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
52          }
53
54          if (i > 0)
55            unite(par[i], i);
56        }
57
58        for (int i = 1; i < n; i++) {
59          if (dom[i] != sdom[i])
60            dom[i] = dom[dom[i]];
61          tree[order[i]].pb(order[dom[i]]);
62          tree[order[dom[i]]].pb(order[i]);
63        }
64
65        T = 0;
66        tin = tout = vi(n);
67        dfs_tree(root, -1);
68      }
69
70      void unite (int u, int v) {
71        dsu[v] = u;
72      }
73
74      int find (int u, int x = 0) {
75        if (u == dsu[u])
76          return (x ? -1 : u);
77        int v = find(dsu[u], x + 1);
78        if (v == -1)
79          return u;
80        if (sdom[label[dsu[u]]] < sdom[label[u]])
81          label[u] = label[dsu[u]];
82        dsu[u] = v;
83        return (x ? v : label[u]);
84      }
85
86      bool dominated_by (int v, int by_what) {
87        return tin[by_what] <= tin[v] && tout[v] <= tout[by_what];
88      }
89    };
```

9.2 Suffix Array

```
1  namespace suff_arr {
2
3  const int MAXN = 2e5 + 10;
4
5  string s;
6  int n;
7  int p[MAXN];
8  int lcp[MAXN];
9  int pos[MAXN];
```

```
10  int c[MAXN];
11
12  void print() {
13  #ifndef LOCAL
14      return;
15  #endif
16      eprintf("p:\n");
17      forn(i, sz(s)) {
18          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]]);
19      }
20      eprintf("\n");
21  }
22
23  void build(const string& s_) {
24      static int cnt[MAXN];
25      static int np[MAXN];
26      static int nc[MAXN];
27
28      s = s_;
29      n = sz(s);
30
31      memset (cnt, 0, sizeof cnt);
32      for (char ch : s) {
33          ++cnt[int(ch)];
34      }
35      forn(i, 256) {
36          cnt[i + 1] += cnt[i];
37      }
38      forn(i, sz(s)) {
39          p[--cnt[int(s[i])]] = i;
40      }
41
42      int cls = 1;
43      c[p[0]] = cls - 1;
44      for (int i = 1; i < n; ++i) {
45          if  (s[p[i]] ≠ s[p[i - 1]]) {
46              ++cls;
47          }
48          c[p[i]] = cls - 1;
49      }
50
51      for (int len = 1; len ≤ n; len *= 2) {
52          memset (cnt, 0, sizeof(int) * cls);
53          forn(i, n) {
54              ++cnt[c[i]];
55          }
56          forn(i, cls - 1) {
57              cnt[i + 1] += cnt[i];
58          }
```

```
59          ford(i, n) {
60              const int j = p[i];
61              int j2 = (j - len + n) % n;
62              np[--cnt[c[j2]]] = j2;
63          }
64          memcpy(p, np, sizeof(int) * n);
65
66          cls = 1;
67          nc[p[0]] = cls - 1;
68          for (int i = 1; i < n; ++i) {
69              if  (c[p[i]] ≠ c[p[i - 1]] || c[(p[i] + len) % n] ≠ c
                  ↪ [(p[i - 1] + len) % n]) {
70                  ++cls;
71              }
72              nc[p[i]] = cls - 1;
73          }
74          memcpy(c, nc, sizeof(int) * n);
75      }
76
77      forn(i, n) {
78          pos[p[i]] = i;
79      }
80
81      int pref = 0;
82      forn(i, n) {
83          int pi = pos[i];
84          if (pi == n - 1) {
85              continue;
86          }
87          int j = p[pi + 1];
88          while (i + pref < n && j + pref < n && s[i + pref] == s[j +
              ↪  pref]) {
89              ++pref;
90          }
91          lcp[pi] = pref;
92          pref = max(0, pref - 1);
93      }
94
95 //        print();
96 }
97
98 };
```

9.3 Fast LCS

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

```
 7
 8   // WARNING: invokes undefined behaviour of modifying ans through
       ↪ pointer to another data type (uint)
 9   // seems to work, but be wary
10   bs sum (const bs &bl, const bs &br)
11   {
12       const int steps = M / 32;
13       const uint* l = (uint*)&bl;
14       const uint* r = (uint*)&br;
15
16       bs ans;
17       uint* res = (uint*)&ans;
18
19       int carry = 0;
20       forn (i, steps)
21       {
22           ll cur = ll(*l++) + ll(*r++) + carry;
23           carry = (cur ⩾ bnd);
24           cur = (cur ⩾ bnd ? cur - bnd : cur);
25           *res++ = uint(cur);
26       }
27
28       return ans;
29   }
30
31   int fast_lcs (const string &s, const string &t)
32   {
33       const int m = sz(t);
34       const int let = 26;
35
36       vector<bs> has(let);
37       vector<bs> rev = has;
38
39       forn (i, m)
40       {
41           const int pos = t[i] - 'a';
42           has[pos].set(i);
43           forn (j, let) if (j ≠ pos)
44               rev[j].set(i);
45       }
46
47       bs row;
48       forn (i, m)
49           row.set(i);
50
51       int cnt = 0;
52       for (char ch : s)
53       {
54           const int pos = ch - 'a';
55
```

```
56            bs next = sum(row, row & has[pos]) | (row & rev[pos]);
57            cnt += next[m];
58            next[m] = 0;
59
60            row = next;
61        }
62
63        return cnt;
64    }
```

9.4   Fast Subset Convolution

```
1  // algorithm itself starts here
2  void mobius (int* a, int n, int sign)
3  {
4      forn (i, n)
5      {
6          int free = ((1 << n) - 1) ^ (1 << i);
7          for (int mask = free; mask > 0; mask = ((mask - 1) & free))
8              (sign == +1 ? add : sub)(a[mask ^ (1 << i)], a[mask]);
9          add(a[1 << i], a[0]);
10      }
11 }
12
13 // maximum number of bits allowed
14 const int B = 20;
15
16 vi fast_conv (vi a, vi b)
17 {
18     assert(!a.empty());
19     const int bits = __builtin_ctz(sz(a));
20     assert(sz(a) == (1 << bits) && sz(a) == sz(b));
21
22     static int trans_a[B + 1][1 << B];
23     static int trans_b[B + 1][1 << B];
24     static int trans_res[B + 1][1 << B];
25
26     forn (cnt, bits + 1)
27     {
28         for (auto cur : {trans_a, trans_b, trans_res})
29             fill(cur[cnt], cur[cnt] + (1 << bits), 0);
30     }
31
32     forn (mask, 1 << bits)
33     {
34         const int cnt = __builtin_popcount(mask);
35         trans_a[cnt][mask] = a[mask];
36         trans_b[cnt][mask] = b[mask];
37     }
38
39     forn (cnt, bits + 1)
```

```
40        {
41            mobius(trans_a[cnt], bits, +1);
42            mobius(trans_b[cnt], bits, +1);
43        }
44
45        // Not really a valid ranked mobius transform! But algorithm
              ↪ works anyway
46
47        forn (i, bits + 1) forn (j, bits - i + 1) forn (mask, 1 << bits
              ↪ )
48            add(trans_res[i + j][mask], mult(trans_a[i][mask], trans_b[
                  ↪ j][mask]));
49
50        forn (cnt, bits + 1)
51            mobius(trans_res[cnt], bits, -1);
52
53        forn (mask, 1 << bits)
54        {
55            const int cnt = __builtin_popcount(mask);
56            a[mask] = trans_res[cnt][mask];
57        }
58
59        return a;
60 }
```

10   Karatsuba

```
1  using hvect = vector<modulo◇>::iterator;
2  using hcvect = vector<modulo◇>::const_iterator;
3
4
5  void add(hcvect abegin, hcvect aend, hvect ans)
6  {
7      for (auto it = abegin; it ≠ aend; ++it, ++ans)
8          *ans += *it;
9  }
10
11
12 void sub(hcvect abegin, hcvect aend, hvect ans)
13 {
14     for (auto it = abegin; it ≠ aend; ++it, ++ans)
15         *ans -= *it;
16 }
17
18
19 void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans)
20 {
21     for (auto a = abegin; a ≠ abegin + siz; ++a, ans -= (siz - 1))
22         for (auto b = bbegin; b ≠ bbegin + siz; ++b, ++ans)
23             *ans += *a * *b;
```

```
24  }
25
26
27  void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin, hvect ans,
        ↪  hvect small, hvect big, hvect sum)
28  {
29      assert((siz & (siz - 1)) == 0);
30
31      fill(ans, ans + 2 * siz, 0);
32
33      if (siz < 60)
34      {
35          stupid(siz, abegin, bbegin, ans);
36
37          return;
38      }
39
40      auto amid = abegin + siz / 2, aend = abegin + siz;
41      auto bmid = bbegin + siz / 2, bend = bbegin + siz;
42      auto smid = sum + siz / 2, send = sum + siz;
43
44      Karatsuba(siz / 2, abegin, bbegin, small, small + siz, big +
          ↪  siz, sum);
45      Karatsuba(siz / 2, amid, bmid, big, small + siz, big + siz, sum
          ↪  );
46
47      copy(abegin, amid, sum);
48      add(amid, aend, sum);
49      copy(bbegin, bmid, sum + siz / 2);
50      add(bmid, bend, sum + siz / 2);
51
52      Karatsuba(siz / 2, sum, smid, ans + siz / 2, small + siz, big +
          ↪   siz, send);
53
54      add(small, small + siz, ans);
55      sub(small, small + siz, ans + siz / 2);
56      add(big, big + siz, ans + siz);
57      sub(big, big + siz, ans + siz / 2);
58  }
59
60
61  void mult(vector<modulo◇> a, vector<modulo◇> b, vector<modulo◇>
        ↪  &c)
62  {
63      a.resize(up(max(a.size(), b.size())));
64      b.resize(a.size());
65
66      c.resize(a.size() * 2);
67
68      auto small = c;
```

```
69        auto big = c;
70        auto sum = c;
71
72        Karatsuba(a.size(), a.begin(), b.begin(), c.begin(), small.
             ↪ begin(), big.begin(), sum.begin());
73 }
```