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
```

1.3 Stack size & Profiling

```
1  # Print stack limit in Kb
2  ulimit -s
3
4  # Set stack limit in Kb, session-local, so resets after terminal restart
5  ulimit -S -s 131072
6
7  # Profile time
8  time ./olymp
9
10 # Profile time, memory, etc.
11 # Make sure to use the full path
12 /usr/bin/time -v ./olymp
```

2 Language specific

2.1 C++

2.1.1 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.

2.1.2 hash

```cpp
namespace std
{
        template<>
        struct hash<pnt>
        {
                std::size_t operator()(pnt const &s) const noexcept
                {
                        return std::hash<ll>{}(s.first * ll(1ull << 32u) +
                        ↪  s.second);
                }
        };
}
```

2.2 Python

```python
# stack size
import sys

sys.setrecursionlimit(10**6)

# memoize
import functools

@functools.lru_cache(maxsize=None)
```

3 Geometry

3.1 Пересечение прямых

$$AB \coloneqq A - B; CD \coloneqq C - D$$

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

3.2 Касательные

Точки пересечения общих касательных окружностей с центрами в $(0, 0)$ и $(x, 0)$ равны $\frac{xr_1}{r1 \pm r2}$. x координата точек касания из $(x, 0)$ равна $\frac{r^2}{x}$.

3.3 Пересечение полуплоскостей

Точно так же, как в выпуклой оболочке, но надо добавить bounding box (квадратичного размера относительно координат на входе) и завернуть два раза. Ответ можно найти как подотрезок от первой полуплоскости типа true до нее же самой на втором круге. Проверку на вырожденность лучше делать простой проверкой пары-тройки точек из предполагаемого ответа. Стоит быть аккуратнее с точностью.

```
using pnti = complex<int>;
using pntd = complex<ld>;

ll operator%(const pnti &a, const pnti &b)
{
        return a.real() * (ll) b.imag() - a.imag() * (ll) b.real();
}


ld operator%(const pntd &a, const pntd &b)
{
        return a.real() * b.imag() - a.imag() * b.real();
}

namespace std
{
        template<class T>
        bool operator<(const complex<T> &a, const complex<T> &b)
        {
                if (a.real() == b.real())
                        return a.imag() < b.imag();
                return a.real() < b.real();
        }
}

pntd line_intersect(const pnti &a, const pnti &b, const pnti &c, const pnti &d)
{
        auto ab = a - b;
        auto cd = c - d;

        auto det = ab % cd;

        assert(det != 0);

        pntd ans{ld(a % b * cd.real() - c % d * ab.real()),
                 ld(a % b * cd.imag() - c % d * ab.imag())};

        return ans / (ld) det;
}

pntd conv(const pnti &x)
{
        return pntd{(ld) x.real(), (ld) x.imag()};
```

```
44  }
45
46  auto halfplane_intesection(const vector<pair<pnti, pnti>> &hp)
47  {
48          vector<int> ind(hp.size());
49
50          iota(ind.begin(), ind.end(), 0);
51
52          auto type = [&hp](int j) -> bool
53          {
54                  return hp[j].first < pnti{0, 0};
55          };
56
57          sort(ind.begin(), ind.end(), [&](int a, int b)
58          {
59                  if (type(a) != type(b))
60                          return type(a) < type(b);
61                  return hp[a].first % hp[b].first > 0;
62          });
63
64          ind.resize(ind.size() * 2);
65          copy(ind.begin(), ind.begin() + ind.size() / 2, ind.begin() + ind.size() /
            ↪  2);
66
67          auto parallel = [&hp](int a, int b)
68          {
69                  return hp[a].first % hp[b].first == 0;
70          };
71
72          vector<int> ans;
73
74          auto hp_intersect = [&hp](int a, int b)
75          {
76                  return line_intersect(hp[a].second, hp[a].second + hp[a].first,
77                                        hp[b].second, hp[b].second + hp[b].first);
78          };
79
80          for (auto nxt: ind)
81          {
82                  while (!ans.empty())
83                  {
84                          auto lst = ans.back();
85
86                          if (parallel(lst, nxt))
87                          {
88                                  if (hp[nxt].first % (hp[nxt].second -
                                    ↪  hp[lst].second) > 0)
89                                          ans.pop_back();
90                                  else
91                                          break;
```

```
92                          }
93                          else if (ans.size() > 1)
94                          {
95                                  auto prv = ans[ans.size() - 2];
96
97                                  auto inter = hp_intersect(nxt, lst);
98
99                                  if (conv(hp[prv].first) % inter ⩽
                                    ↪  ld(hp[prv].first % hp[prv].second))
100                                         ans.pop_back();
101                                 else
102                                         break;
103                         }
104                         else
105                                 break;
106                 }
107
108                 if (ans.empty() || !parallel(nxt, ans.back()))
109                         ans.push_back(nxt);
110         }
111
112         auto st = find_if(ans.begin(), ans.end(), [&type](int q)
113         {
114                 return type(q);
115         });
116
117         if (st == ans.end())
118                 return vector<int>{};
119
120         auto en = find(st + 1, ans.end(), *st);
121
122         if (en == ans.end() || en - st < 3)
123                 return vector<int>{};
124
125         return vector<int>(st, en);
126 }
```

4 Template DSU

```
1  template<class ... Types>
2  class dsu
3  {
4          vector<int> par, siz;
5          tuple<Types ...> items;
6
7          template<size_t ... t>
8          void merge(int a, int b, std::index_sequence<t ...>)
9          {
10                 ((get<t>(items)(a, b)), ... );
11         }
```

```
12
13  public:
14          explicit dsu(int n, Types ... args) : par(n, -1), siz(n, 1),
            ↪ items(args ... )
15          {}
16
17          int get_class(int v)
18          {
19                  return par[v] == -1 ? v : par[v] = get_class(par[v]);
20          }
21
22          bool unite(int a, int b)
23          {
24                  a = get_class(a);
25                  b = get_class(b);
26
27                  if (a == b)
28                          return false;
29
30                  if (siz[a] < siz[b])
31                          swap(a, b);
32                  siz[a] += siz[b];
33                  par[b] = a;
34
35                  merge(a, b, make_index_sequence<sizeof ... (Types)>{});
36
37                  return true;
38          }
39  };
```

5 Numbers

- A lot of divisors
  - $\leq 20 : d(12) = 6$
  - $\leq 50 : d(48) = 10$
  - $\leq 100 : d(60) = 12$
  - $\leq 10^3 : d(840) = 32$
  - $\leq 10^4 : d(9240) = 64$
  - $\leq 10^5 : d(83160) = 128$
  - $\leq 10^6 : d(720720) = 240$
  - $\leq 10^7 : d(8648640) = 448$
  - $\leq 10^8 : d(91891800) = 768$
  - $\leq 10^9 : d(931170240) = 1344$
  - $\leq 10^{11} : d(97772875200) = 4032$
  - $\leq 10^{12} : d(963761198400) = 6720$

  - $\leq 10^{15}$: $d(866421317361600) = 26880$
  - $\leq 10^{18}$: $d(897612484786617600) = 103680$

- 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 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;

```

```
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);
```

```
        swap(m1, m2);
    }

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

7.2 Integer points under a rational line

```
// integer (x, y) : 0 ≤ x < n, 0 < y ≤ (kx + b)/d
// (real division)
// In other words, ∑_{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);
}
```

8 Suffix Automaton

```
struct tomato
{
        vector<map<char, int>> edges;
        vector<int> link, length;
        int last;
        /// Restoring terminal states, optional, but usually needed.
        vector<int> terminals;
        vector<bool> is_terminal;
        /// Optional, makes dp easier. Alternative: use dfs.
```

```
10          vector<int> order, rev_order, next_in_order;
11
12          explicit tomato(const string &s) : last(0)
13          {
14                  add_vertex(map<char, int>(), 0, -1);
15                  for (const char ch : s)
16                          extend(ch);
17
18                  int cur = last;
19                  is_terminal.assign(edges.size(), false);
20                  /// Assuming empty suffix should be accepted, otherwise use "while
                    ↪  (cur > 0)".
21                  while (cur ⩾ 0)
22                  {
23                          terminals.push_back(cur);
24                          is_terminal[cur] = true;
25                          cur = link[cur];
26                  }
27
28                  /// Restoring topsort and reverse topsort, optional.
29                  order.push_back(0);
30                  while (order.back() ≠ -1)
31                          order.push_back(next_in_order[order.back()]);
32                  order.pop_back();
33                  rev_order = order;
34                  reverse(rev_order.begin(), rev_order.end());
35          }
36
37          int add_vertex(const map<char, int> &temp, const int len, const int lnk)
38          {
39                  edges.emplace_back(temp);
40                  length.emplace_back(len);
41                  link.emplace_back(lnk);
42                  next_in_order.push_back(-1);
43                  return int(edges.size()) - 1;
44          }
45
46          void extend(const char ch)
47          {
48                  const int new_last = add_vertex(map<char, int>(), length[last] +
                    ↪  1, 0);
49                  assert(next_in_order[last] == -1);
50                  next_in_order[last] = new_last;
51
52                  int p = last;
53                  while (p ⩾ 0 && !edges[p].count(ch))
54                  {
55                          edges[p][ch] = new_last;
56                          p = link[p];
57                  }
```

```
58
59                 if (p != -1)
60                 {
61                         const int q = edges[p][ch];
62                         if (length[p] + 1 == length[q])
63                                 link[new_last] = q;
64                         else
65                         {
66                                 const int clone = add_vertex(edges[q], length[p] +
                                   ↪  1, link[q]);
67                                 next_in_order[clone] = next_in_order[q];
68                                 next_in_order[q] = clone;
69
70                                 link[q] = clone;
71                                 link[new_last] = clone;
72
73                                 while (p >= 0 && edges[p][ch] == q)
74                                 {
75                                         edges[p][ch] = clone;
76                                         p = link[p];
77                                 }
78                         }
79                 }
80
81                 last = new_last;
82         }
83 };
```

9 Palindromic Tree

```
1  class treert
2  {
3          struct node
4          {
5                  array<int, 26> nxt;
6                  int par, link, siz;
7
8                  node(int siz, int par, int link) : par(par), link(link == -1 ? 1 :
                   ↪  link), siz(siz)
9                  {
10                         fill(nxt.begin(), nxt.end(), -1);
11                 }
12         };
13
14         vector<node> mem;
15         vector<int> suff; // longest palindromic suffix
16
17 public:
18         treert(const string &str) : suff(str.size())
19         {
```

```
20                mem.emplace_back(-1, -1, 0);
21                mem.emplace_back(0, 0, 0);
22                mem[0].link = mem[1].link = 0;
23
24                auto link_walk = [&](int st, int pos)
25                {
26                        while (pos - 1 - mem[st].siz < 0 || str[pos] ≠ str[pos -
                          ↪  1 - mem[st].siz])
27                                st = mem[st].link;
28
29                        return st;
30                };
31
32                for (int i = 0, last = 1; i < str.size(); i++)
33                {
34                        last = link_walk(last, i);
35                        auto ind = str[i] - 'a';
36
37                        if (mem[last].nxt[ind] == -1)
38                        {
39                                // order is important
40                                mem.emplace_back(mem[last].siz + 2, last,
                                  ↪  mem[link_walk(mem[last].link, i)].nxt[ind]);
41                                mem[last].nxt[ind] = (int)mem.size() - 1;
42                        }
43
44                        last = mem[last].nxt[ind];
45
46                        suff[i] = last;
47                }
48        }
49 };
```

10 Smth added at last moment

10.1 Dominator Tree

```
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  };
```

10.2 Suffix Array

```
1  /// Actually sorts suffixes now.
2  vector<int> suff_array(string s)
3  {
4          /// Add something that is less than all of symbols from s
5          s += 'a' - 1;
6          /// Only now we can compute the length of the string.
7          const int n = int(s.size());
8          vector<int> p(n);
9
10         const int bound = 260;
11         vector<int> c(max(n, bound));
12         auto cnt = c, np = c, nc = c;
13
14         for (char ch : s)
15                 cnt[ch]++;
16
17         for (int i = 1; i < 256; i++)
18                 cnt[i] += cnt[i - 1];
19         for (int i = 0; i < n; i++)
20                 p[--cnt[s[i]]] = i;
21
```

```
22          int cls = 1;
23          c[p[0]] = cls - 1;
24
25          for (int i = 1; i < n; i++)
26          {
27                  if (s[p[i]] != s[p[i - 1]])
28                          ++cls;
29                  c[p[i]] = cls - 1;
30          }
31
32          for (int len = 1; len <= n; len *= 2)
33          {
34                  fill(cnt.begin(), cnt.begin() + cls, 0);
35                  for (int i = 0; i < n; i++)
36                          cnt[c[i]]++;
37
38                  for (int i = 0; i < cls - 1; i++)
39                          cnt[i + 1] += cnt[i];
40
41
42                  for (int i = n - 1; i >= 0; i--)
43                  {
44                          const int j = p[i];
45                          int j2 = (j - len + n) % n;
46                          np[--cnt[c[j2]]] = j2;
47                  }
48                  copy(np.begin(), np.begin() + n, p.begin());
49
50                  cls = 1;
51                  nc[p[0]] = cls - 1;
52                  for (int i = 1; i < n; i++)
53                  {
54                          if (c[p[i]] != c[p[i - 1]] || c[(p[i] + len) % n] !=
                            ↪  c[(p[i - 1] + len) % n])
55                                  ++cls;
56                          nc[p[i]] = cls - 1;
57                  }
58
59                  copy(nc.begin(), nc.begin() + n, c.begin());
60          }
61
62          /// Removing the extra symbol.
63          assert(p.front() == int(p.size()) - 1);
64          p.erase(p.begin());
65          return p;
66  }
67
68  vector<int> kasai_lcp(const vector<int> &p, const string &s)
69  {
70          const int n = int(p.size());
```

```
71          vector<int> lcp(n - 1);
72
73          vector<int> pos(n);
74          for (int i = 0; i < n; i++)
75                  pos[p[i]] = i;
76
77          int pref = 0;
78          for (int i = 0; i < n; i++)
79          {
80                  const int pi = pos[i];
81                  if (pi == n - 1)
82                          continue;
83
84                  const int j = p[pi + 1];
85                  while (i + pref < n && j + pref < n && s[i + pref] == s[j + pref])
86                          ++pref;
87
88                  lcp[pi] = pref;
89                  pref = max(0, pref - 1);
90          }
91
92          return lcp;
93  }
```

10.3 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);
```

```
        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];
```

```
57      }
58
59      return a;
60  }
```

11 Karatsuba

```
1   // functon Karatsuba (and stupid as well) computes c += a * b, not c = a * b
2
3   using hvect = vector<modulo<>>::iterator;
4   using hcvect = vector<modulo<>>::const_iterator;
5
6   void add(hcvect abegin, hcvect aend, hvect ans)
7   {
8           for (auto it = abegin; it != aend; ++it, ++ans)
9                   *ans += *it;
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  void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans)
19  {
20          for (auto a = abegin; a != abegin + siz; ++a, ans -= (siz - 1))
21                  for (auto b = bbegin; b != bbegin + siz; ++b, ++ans)
22                          *ans += *a * *b;
23  }
24
25  void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin, hvect ans, hvect small,
    ↪  hvect big, hvect sum)
26  {
27          assert((siz & (siz - 1)) == 0);
28
29          if (siz <= 32)
30          {
31                  stupid(siz, abegin, bbegin, ans);
32
33                  return;
34          }
35
36          auto amid = abegin + siz / 2, aend = abegin + siz;
37          auto bmid = bbegin + siz / 2, bend = bbegin + siz;
38          auto smid = sum + siz / 2, send = sum + siz;
39
40          fill(small, small + siz, 0);
41          Karatsuba(siz / 2, abegin, bbegin, small, small + siz, big + siz, sum);
42          fill(big, big + siz, 0);
```

```
43          Karatsuba(siz / 2, amid, bmid, big, small + siz, big + siz, sum);
44
45          copy(abegin, amid, sum);
46          add(amid, aend, sum);
47          copy(bbegin, bmid, sum + siz / 2);
48          add(bmid, bend, sum + siz / 2);
49
50          Karatsuba(siz / 2, sum, smid, ans + siz / 2, small + siz, big + siz,
            ↪  send);
51
52          add(small, small + siz, ans);
53          sub(small, small + siz, ans + siz / 2);
54          add(big, big + siz, ans + siz);
55          sub(big, big + siz, ans + siz / 2);
56  }
57
58  void mult(vector<modulo<>> a, vector<modulo<>> b, vector<modulo<>> &c)
59  {
60          a.resize(up(max(a.size(), b.size())), 0);
61          b.resize(a.size(), 0);
62
63          c.resize(max(c.size(), a.size() * 2), 0);
64
65          vector<modulo<>> small(2 * a.size());
66          auto big = small;
67          auto sum = small;
68
69          Karatsuba(a.size(), a.begin(), b.begin(), c.begin(), small.begin(),
            ↪  big.begin(), sum.begin());
70  }
```

12 Two Strong Chinese

```
228 void solve(istream &cin = std::cin, ostream &cout = std::cout)
229 {
230         ll ans = 0;
231
232         int n, m, root = 0;
233
234         cin >> n >> m;
235
236         vector<vector<pair<int, int>>> gr(n);
237
238         for (int i = 0; i < m; i++)
239         {
240                 int a, b, w;
241
242                 cin >> a >> b >> w;
243                 a--;
244                 b--;
```

```
245
246             gr[a].emplace_back(b, w);
247         }
248
249         vector<bool> used(n);
250
251         dfs(gr, used, root);
252
253         if (count(used.begin(), used.end(), false))
254         {
255             cout << "NO" << endl;
256
257             return;
258         }
259
260         cout << "YES" << endl;
261
262         vector<treap<pair<ll, int>>> rev(n);
263
264         for (int v = 0; v < (int) gr.size(); v++)
265             for (auto it : gr[v])
266                 if (it.first != root)
267                     rev[it.first].insert(pair{(ll) it.second, v});
268
269         fill(used.begin(), used.end(), false);
270         auto grey = used;
271         used.front() = true;
272
273         dsu group(n);
274
275         vector<ll> add(n);
276
277         auto merge_vert = [&](int a, int b, int to)
278         {
279             if (rev[a].size() < rev[b].size())
280                 swap(a, b);
281
282             auto dif = add[b] - add[a];
283             add[b] = add[a];
284
285             rev[b].apply_to_all([dif](auto &x)
286                                 {
287                                     x.first += dif;
288                                 });
289
290             treap tmp(rev[a], rev[b]);
291
292             rev[to].swap(tmp);
293         };
294
```

```
295         // returns whether we are on cycle
296         std::function<bool(int)> ans_vert = [&](int v) -> bool
297         {
298                 v = group.get_par(v);
299
300                 if (used[v])
301                         return false;
302
303                 if (grey[v])
304                 {
305                         grey[v] = false;
306
307                         return true;
308                 }
309                 grey[v] = true;
310
311                 while (true)
312                 {
313                         auto nxt = rev[v].top();
314
315                         while (group.get_par(nxt.second) == v)
316                         {
317                                 rev[v].pop();
318                                 nxt = rev[v].top();
319                         }
320
321                         ans += nxt.first + add[v];
322                         add[v] -= nxt.first + add[v];
323
324                         auto type = ans_vert(nxt.second);
325
326                         if (type)
327                         {
328                                 if (grey[v])
329                                 {
330                                         nxt.second = group.get_par(nxt.second);
331
332                                         group.unite(v, nxt.second);
333
334                                         merge_vert(v, nxt.second,
                                         ↪  group.get_par(v));
335
336                                         v = group.get_par(v);
337
338                                         grey[v] = false;
339
340                                         return true;
341                                 }
342                                 else
343                                 {
```

```
344                                     v = group.get_par(v);
345
346                                     grey[v] = true;
347                             }
348                     }
349                     else
350                     {
351                             ans += nxt.first + add[v];
352                             add[v] -= nxt.first + add[v];
353                             used[v] = true;
354                             grey[v] = false;
355
356                             return false;
357                     }
358             }
359     };
360
361     for (int i = 0; i < gr.size(); i++)
362             if (group.get_par(i) == i)
363                     ans_vert(i);
364
365     for (int v = 0; v < gr.size(); v++)
366             assert(used[v] == (group.get_par(v) == v));
367
368     cout << (long long) ans << endl;
369 }
```

13 OEIS

13.1 Числа Белла

1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322, 1382958545, 10480142147, 82864869804, 682076806159, 5832742205057, 51724158235372, 474869816156751, 4506715738447323, 44152005855084346, 445958869294805289, 4638590332229999353, 49631246523618756274

13.2 Числа Каталана

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, 18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304