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 Stack size & Profiling

```
# Print stack limit in Kb
ulimit -s

# Set stack limit in Kb, session-local, so resets after terminal restart
ulimit -S -s 131072

# Profile time
time ./olymp

# Profile time, memory, etc.
# Make sure to use the full path
/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 вызывать 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 Custom Hash

```
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);
    }
  };
}
```

2.1.3 Allocator

```
template<size_t sz>
class chunk_alloc
{
public:
  static constexpr auto chunk_size = sz;

private:
  using chunk_t = array<byte, chunk_size>;

  deque<chunk_t> mem;
  stack<void *> emp;

public:
  void *allocate()
  {
    if (emp.empty())
      emp.push(reinterpret_cast<void *>(&mem.emplace_back()));

    auto ans = emp.top();
    emp.pop();

    return ans;
  }

  void deallocate(void *p) noexcept
  {
    emp.push(p);
  }
};


chunk_alloc<64> pool;


template<class T>
struct dummy_alloc
{
  using value_type = T;

  dummy_alloc() noexcept = default;

  template<class U>
  explicit dummy_alloc(const dummy_alloc<U> &) noexcept
  {}

  T *allocate(std::size_t n)
  {
```

```cpp
    if constexpr (sizeof(value_type) == decltype(pool)::chunk_size)
      return static_cast<T *>(pool.allocate());
    else
      return static_cast<T *>(::operator new(n * sizeof(value_type)));
  }

  void deallocate(T *p, std::size_t n)
  {
    if constexpr (sizeof(value_type) == decltype(pool)::chunk_size)
      return pool.deallocate(p);
    else
      ::delete (p);
  }
};

template<class T, class U>
constexpr bool operator==(const dummy_alloc<T> &, const dummy_alloc<U> &) noexcept
{ return true; }

template<class T, class U>
constexpr bool operator!=(const dummy_alloc<T> &, const dummy_alloc<U> &) noexcept
{ return false; }
```

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

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

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

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.

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

5 Graphs

5.1 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 in) -> T
{
    vector<int> ids;
    for (int i = 0; i < int(es.size()); i++)
        if (in[i])
            ids.push_back(i);

    vector<int> from, to;
    /// Given a set that is independent in both matroids, answers
    /// queries "If we add i-th element to the set, will it still be
    /// independent in red/blue matroid?". Usually can be done quickly.
    can_extend full_can(ids, n, es);

    for (int i = 0; i < int(es.size()); i++)
        if (!in[i])
```

```
        {
            auto new_ids = ids;
            new_ids.push_back(i);

            auto is_red = full_can.extend_red(i, es);
            auto is_blue = full_can.extend_blue(i, es);

            if (is_blue)
                from.push_back(i);
            if (is_red)
                to.push_back(i);

            if (is_red && is_blue)
            {
                T swp_mask = in;
                swp_mask.flip(i);
                return swp_mask;
            }
    }

    vector<vector<int>> g(es.size());
    for (int j = 0; j < int(es.size()); j++)
        if (in[j])
        {
            auto new_ids = ids;
            auto p = find(new_ids.begin(), new_ids.end(), j);
            assert(p != new_ids.end());
            new_ids.erase(p);

            can_extend cur(new_ids, n, es);

            for (int i = 0; i < int(es.size()); i++)
                if (!in[i])
                {
                    if (cur.extend_red(i, es))
                        g[i].push_back(j);
                    if (cur.extend_blue(i, es))
                        g[j].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);
};
```

6 Data structures

6.1 Push-free segment tree

```
template<class Val, class Change, Change one = Change{}>
class pushfreesegtree
{
  vector<pair<Val, Change>> arr;

  void upd(size_t v)
  {
    arr[v].first = (arr[2 * v].first + arr[2 * v + 1].first) * arr[v].second;
  }

public:
  explicit pushfreesegtree(size_t n = 0) : arr(2 * n + 2, {Val{}, one})
  {}

  template<class It>
  explicit pushfreesegtree(It be, It en) : arr(2 * distance(be, en) + 2, {Val{}, one})
```

```cpp
  {
    transform(be, en, arr.begin() + ssize(arr) / 2, [](auto x)
    {
      return pair{Val{x}, one};
    });

    for (int i = ssize(arr) / 2 - 1; i > 0; i--)
      upd(i);
  }

  auto segmult(const Change &x, size_t l, size_t r)
  {
    l += arr.size() / 2;
    r += arr.size() / 2;

    while (true)
    {
      if (l < r)
      {
        if (l & 1u)
        {
          arr[l].first *= x;
          arr[l].second *= x;
        }
        if (r & 1u)
        {
          arr[r - 1].first *= x;
          arr[r - 1].second *= x;
        }
      }

      l = (l + 1) / 2;
      r /= 2;

      if (r == 0)
        break;

      upd(l - 1);
      upd(r);
    }
  }

  [[nodiscard]] Val segsum(size_t l, size_t r) const
  {
    l += arr.size() / 2;
    r += arr.size() / 2;

    Val ansl{}, ansr{};

    while (true)
    {
      if (l < r)
      {
        if (l & 1u)
          ansl = ansl + arr[l].first;
        if (r & 1u)
          ansr = arr[r - 1].first + ansr;
      }

      l = (l + 1) / 2;
      r /= 2;

      if (r == 0)
        break;

```

```
      ansl *= arr[l - 1].second;
      ansr *= arr[r].second;
    }

    return ansl + ansr;
  }
};
```

6.2 Template DSU

```
template<class ... Types>
class dsu
{
  vector<int> par, siz;
  tuple<Types ...> items;

  template<size_t ... t>
  void merge(int a, int b, std::index_sequence<t...>)
  {
    ((get<t>(items)(a, b)), ... );
  }

public:
  explicit dsu(int n, Types ... args) : par(n, -1), siz(n, 1), items(args...)
  {}

  int get_class(int v)
  {
    return par[v] == -1 ? v : par[v] = get_class(par[v]);
  }

  bool unite(int a, int b)
  {
    a = get_class(a);
    b = get_class(b);

    if (a == b)
      return false;

    if (siz[a] < siz[b])
      swap(a, b);
    siz[a] += siz[b];
    par[b] = a;

    merge(a, b, make_index_sequence<sizeof...(Types)>{});

    return true;
  }
};
```

6.3 Link-Cut Tree

```
class lct
{
  struct node
  {
    using nodeptr = node *;

    array<nodeptr, 2> ch{};
    nodeptr par = nullptr;
    size_t siz = 1;
    bool rev = false;
  };

  using nodeptr = node::nodeptr;
```

```cpp

static void reverse(const nodeptr &h)
{
  if (h != nullptr)
    h->rev = !h->rev;
}

static void push(node &h)
{
  if (h.rev)
  {
    swap(h.ch.front(), h.ch.back());
    h.rev = false;

    for (auto it: h.ch)
      reverse(it);
  }
}

static auto size(const nodeptr &h)
{
  return h == nullptr ? 0 : h->siz;
}

static void upd(node &h)
{
  h.siz = 1;

  for (auto it: h.ch)
  {
    h.siz += size(it);

    if (it != nullptr)
      it->par = &h;
  }
}

static bool is_root(const node &h)
{
  return h.par == nullptr || find(h.par->ch.begin(), h.par->ch.end(), &h) == h.par->ch.end();
}

static bool is_right(const node &h)
{
  assert(!is_root(h));
  push(*h.par);
  return get<1>(h.par->ch) == &h;
}

static void zig(node &h)
{
  assert(!is_root(h));

  auto &p = *h.par;
  push(p);
  push(h);
  auto pp = p.par;
  bool ind = is_right(h);
  auto &x = p.ch[ind];
  auto &b = h.ch[!ind];

  x = b;
  b = &p;
  h.par = pp;

```

```cpp
    upd(p);
    upd(h);

    if (pp != nullptr)
      for (auto &it: pp->ch)
        if (it == &p)
          it = &h;
  }

  static void splay(node &h)
  {
    push(h);
    while (!is_root(h))
    {
      auto &p = *h.par;

      if (is_root(p))
      {
        zig(h);
      }
      else if (is_right(h) == is_right(p))
      {
        zig(p);
        zig(h);
      }
      else
      {
        zig(h);
        zig(h);
      }
    }
  }

  static void expose(node &h)
  {
    splay(h);

    while (h.par != nullptr)
    {
      auto &p = *h.par;
      splay(p);
      get<1>(p.ch) = &h;
      upd(p);
      splay(h);
    }
  }
};
```

7 Strings

7.1 Suffix Automaton

```cpp
class tomato
{
  struct node
  {
    array<int, 26> nxt{};
    int link = -1, len = 0;

    explicit node(int len = 0) : len(len)
    {
      ranges::fill(nxt, -1);
    }

    explicit node(int len, node p) : nxt(p.nxt), len(len), link(p.link)
```

```cpp
    {}
  };

  vector<node> mem = {node(0)};
  int last = 0;

public:
  explicit tomato(string_view sv = "")
  {
    for (auto it: sv)
      (*this) += it;
  }


  tomato &operator+=(char ch)
  {
    const int ind = ch - 'a';
    auto new_last = int(mem.size());
    mem.emplace_back(mem[last].len + 1);

    auto p = last;
    while (p >= 0 && mem[p].nxt[ind] == -1)
    {
      mem[p].nxt[ind] = new_last;
      p = mem[p].link;
    }

    if (p != -1)
    {
      const int q = mem[p].nxt[ind];
      if (mem[p].len + 1 == mem[p].len)
      {
        mem[new_last].link = q;
      }
      else
      {
        auto clone = int(mem.size());
        mem.emplace_back(mem[p].len + 1, mem[q]);
        mem[q].link = clone;
        mem[new_last].link = clone;

        while (p >= 0 && mem[p].nxt[ind] == q)
        {
          mem[p].nxt[ind] = clone;
          p = mem[p].link;
        }
      }
    }
    else
      mem[new_last].link = 0;

    last = new_last;

    return *this;
  }
};
```

7.2 Palindromic Tree

```cpp
class treert
{
  struct node
  {
    array<int, 26> nxt;
    int par, link, siz;

```

```
    node(int siz, int par, int link) : par(par), link(link == -1 ? 1 : link), siz(siz) // note -1 case
    {
      fill(nxt.begin(), nxt.end(), -1);
    }
  };

  vector<node> mem;
  vector<int> suff; // longest palindromic suffix

public:
  treert(const string &str) : suff(str.size())
  {
    mem.emplace_back(-1, -1, 0);
    mem.emplace_back(0, 0, 0);
    mem[0].link = mem[1].link = 0;

    auto link_walk = [&](int st, int pos)
    {
      while (pos - 1 - mem[st].siz < 0 || str[pos] != str[pos - 1 - mem[st].siz])
        st = mem[st].link;

      return st;
    };

    for (int i = 0, last = 1; i < str.size(); i++)
    {
      last = link_walk(last, i);
      auto ind = str[i] - 'a';

      if (mem[last].nxt[ind] == -1)
      {
        // order is important
        mem.emplace_back(mem[last].siz + 2, last, mem[link_walk(mem[last].link, i)].nxt[ind]);
        mem[last].nxt[ind] = (int)mem.size() - 1;
      }

      last = mem[last].nxt[ind];

      suff[i] = last;
    }
  }
};
```

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;
```

```cpp
}

// 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);
    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} floor((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 Nimbers

```cpp
template<class T, int lvl>
pair<T, T> split(T x)
{
  return {x >> lvl, x & ((T{1} << lvl) - 1)};
}

template<class T, int lvl>
T combine(T a, T b)
{
  return (a << lvl) | b;
}

```

```cpp
template<class T, int lvl = 8 * sizeof(T)>
T nim_hmul(T x)
{
  constexpr int half = lvl / 2;
  if constexpr (lvl == 1)
    return x;

  auto [a, b] = split<T, half>(x);

  return combine<T, half>(nim_hmul<T, half>(a ^ b),
   nim_hmul<T, half>(nim_hmul<T, half>(a)));
}

template<class T, int lvl = 8 * sizeof(T)>
T nim_mul(T x, T y)
{
  constexpr int half = lvl / 2;
  if constexpr (lvl == 1)
    return x & y;

  auto [a, b] = split<T, half>(x);
  auto [c, d] = split<T, half>(y);

  auto ac = nim_mul<T, half>(a, c);
  auto bd = nim_mul<T, half>(b, d);
  auto hp = nim_mul<T, half>(a ^ b, c ^ d) ^ bd;

  return combine<T, half>(hp, bd ^ nim_hmul<T, half>(ac));
}

template<class T, int lvl = 8 * sizeof(T)>
T nim_sqr(T x)
{
  return nim_mul<T, lvl>(x, x);
}

template<class T, int lvl = 8 * sizeof(T)>
T nim_sqrt(T x)
{
  constexpr int half = lvl / 2;
  if constexpr (lvl == 1)
    return x;

  auto [a, b] = split<T, half>(x);

  return combine<T, half>(nim_sqrt<T, half>(a), nim_sqrt<T,
   half>(nim_hmul<T, half>(a) ^ b));
}

template<class T, int lvl = 8 * sizeof(T)>
T nim_recip(T x)
{
  constexpr int half = lvl / 2;
  if constexpr (lvl == 1)
    return x;

  auto [a, b] = split<T, half>(x);

  auto ad = nim_mul<T, half>(a ^ b, b);
  auto bc = nim_hmul<T, half>(nim_sqr<T, half>(a));
  auto det_recip = nim_recip<T, half>(ad ^ bc);

  return combine<T, half>(nim_mul(a, det_recip), nim_mul(a
   ^ b, det_recip));
}
```

10 Flows, etc.

10.1 Hungarian Algorithm

```
ld Hungarian(const vector<vector<ld>> &matr)
{
  vector<int> lb(matr.size(), -1), rb(matr[0].size(), -1);
  vector<ld> rows(matr.size()), cols(rb.size());

  for (int v = 0; v < ssize(matr); v++)
  {
    vector<bool> lused(lb.size()), rused(rb.size());
    vector<int> par(rb.size(), -1);
    vector<pair<ld, int>> w(rb.size(),
    ↪  {numeric_limits<ld>::max(), -1});

    auto add_row = [&](int i)
    {
      lused[i] = true;

      for (int j = 0; j < ssize(w); j++)
        remin(w[j], {matr[i][j] + rows[i] + cols[j], i});
    };

    add_row(v);

    while (true)
    {
      int j = -1;

      for (int k = 0; k < ssize(rb); k++)
        if (!rused[k] && (j == -1 || w[k] < w[j]))
          j = k;

      auto [x, i] = w[j];

      for (int k = 0; k < ssize(lused); k++)
        if (!lused[k])
          rows[k] += x;
      for (int k = 0; k < ssize(rused); k++)
        if (!rused[k])
        {
          cols[k] -= x;
          w[k].first -= x;
        }

      par[j] = i;
      rused[j] = true;

      if (rb[j] == -1)
      {
        while (j != -1)
        {
          rb[j] = par[j];
          auto nxt = lb[par[j]];
          lb[par[j]] = j;
          j = nxt;
        }

        break;
      }

      add_row(rb[j]);
    }
  }

```

```
  ld ans = 0;

  for (int i = 0; i < ssize(lb); i++)
    if (auto j = lb[i]; j != -1)
      ans += matr[i][j];

  return ans;
}
```

11 Something added at the last moment

11.1 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;
    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];
  }
};
```

11.2 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;
}
```

11.3 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;
}
```

12 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 (int i = 0; i < siz; i++)
    for (int j = 0; j < siz; j++)
      *(ans + i + j) += *(abegin + i) * *(bbegin + j);
}


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.resize(max(c.size(), a.size() * 2), 0);

  vector<modulo<>> small(2 * a.size());
  auto big = small;
  auto sum = small;

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

13 Hard Algorithms

13.1 Two Strong Chinese

```cpp
template<class T, class Add>
class skew_heap
{
  struct node
  {
    using nodeptr = unique_ptr<node>;

    nodeptr l = nullptr, r = nullptr;
    T x;

    explicit node(T x = {}) : x(x)
    {}
  };

  using nodeptr = typename node::nodeptr;

  static nodeptr merge(nodeptr &&a, nodeptr &&b)
  {
    if (a == nullptr)
      return std::move(b);
    if (b == nullptr)
      return std::move(a);
    if (b->x < a->x)
      return merge(std::move(b), std::move(a));

    auto tmp = merge(std::move(a->r), std::move(b));
    a->r = std::move(a->l);
    a->l = std::move(tmp);

    return std::move(a);
  }

  void add_to_all(nodeptr &a, Add x)
  {
    if (a == nullptr)
      return;

    a->x += x;
    add_to_all(a->l, x);
    add_to_all(a->r, x);
  }

  nodeptr root = nullptr;
  size_t siz = 0;
  Add to_add{};

public:
  void add(Add x)
  {
    to_add += x;
  }

  [[nodiscard]] T top() const
  {
    return root->x + to_add;
  }

  [[nodiscard]] auto size() const
  {
    return siz;
  }

```

```cpp
  [[nodiscard]] auto empty() const
  {
    return size() == 0;
  }

  void pop()
  {
    auto q = merge(std::move(root->l), std::move(root->r));
    siz--;
    root = std::move(q);
  }

  void merge(skew_heap &&rhs)
  {
    if (size() < rhs.size())
      swap(*this, rhs);

    siz += rhs.siz;
    rhs.siz = 0;
    rhs.add_to_all(rhs.root, rhs.to_add - to_add);
    auto q = merge(std::move(root), std::move(rhs.root));
    root = std::move(q);
  }

  void push(T x)
  {
    skew_heap sh;
    sh.root = make_unique<node>(x);
    sh.siz = 1;

    merge(std::move(sh));
  }
};

struct edge
{
  ll w;
  int to;
  int id;

  strong_ordering operator<=>(const edge &rhs) const
  {
    return w <=> rhs.w;
  }

  edge &operator+=(ll rhs)
  {
    w += rhs;

    return *this;
  }

  edge operator+(ll rhs) const
  {
    return edge{w + rhs, to, id};
  }
};

enum color_t
{
  White = 0, Grey, Black, Cycle
};

vector<int> solve(size_t n, const vector<tuple<int, int,
    int>> &edges, int root = 0)
{
```

```
  vector<skew_heap<edge, ll>> rev(n);

  for (int i = 0; i < (int) edges.size(); i++)
  {
    auto [a, b, w] = edges[i];

    if (b != root)
      rev[b].push(edge{w, a, i});
  }

  auto mrg = [&](int a, int b)
  {
    rev[a].merge(std::move(rev[b]));
  };

  dsu cc(n, mrg);

  vector<color_t> color(rev.size());
  color[root] = Black;

  vector<int> ids;

  function<bool(int)> dfs = [&](int v) -> bool
  {
    v = cc.get_class(v);

    if (color[v] == Black)
      return false;

    if (color[v] == Grey)
    {
      color[v] = Cycle;

      return true;
    }
    color[v] = Grey;

    while (true)
    {
      while (!rev[v].empty() && cc.get_class(rev[v].top().to) == v)
        rev[v].pop();

      assert(!rev[v].empty()); // assume that the answer exists
      auto [w, to, id] = rev[v].top();

      ids.emplace_back(id); // ans += w; if the certificate is not needed

      rev[v].add(-w);

      if (dfs(to))
      {
        if (color[v] != Cycle)
        {
          cc.unite(v, to);
          color[cc.get_class(v)] = Cycle;

          return true;
        }
        else
        {
          v = cc.get_class(v);

          color[v] = Grey;
```

```
        }
      }
      else
      {
        color[v] = Black;

        return false;
      }
    }
  };

  for (int i = 0; i < (int) rev.size(); i++)
    dfs(i);

  // finding answer, similar to Prim
  vector<vector<int>> gr(n);

  for (int i = 0; i < int(ids.size()); i++)
  {
    auto [a, b, _] = edges[ids[i]];

    gr[a].push_back(i);
  }

  minheap<int> pq(gr[root].begin(), gr[root].end());
  vector<bool> used(n);
  used[root] = true;

  vector<int> ans;

  while (!pq.empty())
  {
    auto i = pq.top();
    pq.pop();
    auto v = get<1>(edges[ids[i]]);

    if (used[v])
      continue;
    used[v] = true;

    ans.push_back(ids[i]);

    for (auto it: gr[v])
      pq.push(it);
  }

  return ans;
}


void dfs(const vector<vector<pair<int, int>>> &gr, vector<bool> &used, int v)
{
  if (used[v])
    return;
  used[v] = true;

  for (auto [u, w]: gr[v])
    dfs(gr, used, u);
}


void solve(istream &cin = std::cin, ostream &cout = std::cout)
{
  int n, m;
```

```cpp

  cin >> n >> m;

  vector<tuple<int, int, int>> edges(m);
  vector<vector<pair<int, int>>> gr(n);

  for (int i = 0; i < m; i++)
  {
    auto &[a, b, w] = edges[i];

    cin >> a >> b >> w;
    a--;
    b--;

    gr[a].emplace_back(b, w);
  }

  vector<bool> used(gr.size());

  dfs(gr, used, 0);

  if (ranges::count(used, false))
  {
    cout << "NO" << endl;

    return;
  }

  cout << "YES" << endl;

  auto ids = solve(gr.size(), edges);

  ll ans = 0;

  for (auto it: ids)
    ans += get<2>(edges[it]);

  for (auto &row: gr)
    row.clear();

  for (auto it: ids)
  {
    auto [a, b, w] = edges[it];

    gr[a].emplace_back(b, w);
  }

  used.assign(used.size(), false);

  dfs(gr, used, 0);

  assert(ranges::count(used, false) == 0);

  cout << ans << endl;
}
```

13.2 Simplex

```cpp
mt19937 mt(736);

using ld = double;
constexpr ld eps = 1e-9;

bool eps_nonneg(ld x)
{
  return x >= -eps;
```

```cpp
}

bool eps_zero(ld x)
{
  return abs(x) <= eps;
}

bool cmp_abs(ld a, ld b)
{
  return abs(a) < abs(b);
}

vector<ld> &add_prod(vector<ld> &lhs, const vector<ld> &rhs, ld x)
{
  assert(ssize(lhs) == ssize(rhs));

  for (auto i: ranges::iota_view(0, ssize(lhs)))
    lhs[i] += rhs[i] * x;

  return lhs;
}

vector<ld> &operator/=(vector<ld> &lhs, ld x)
{
  for (auto &it: lhs)
    it /= x;

  return lhs;
}

void basis_change(vector<ld> &row, const vector<ld> &nd, int b)
{
  auto mult = row[b];

  add_prod(row, nd, mult);

  row[b] = 0;
}

void pivot(vector<vector<ld>> &a, vector<int> &b, vector<ld> &func, int wh, int x)
{
  a[wh][b[wh]] = -1;
  b[wh] = x;
  auto den = -a[wh][x];
  a[wh][x] = 0;
  a[wh] /= den;

  for (auto i: ranges::iota_view(0, ssize(a)))
    if (i != wh)
      basis_change(a[i], a[wh], b[wh]);
  basis_change(func, a[wh], b[wh]);
}

bool simplex(vector<vector<ld>> &a, vector<int> &b, vector<ld> &func)
{
  while (true)
  {
    vector<int> cand;

    for (auto i: ranges::iota_view(0, ssize(func) - 1))
      if (func[i] > eps)
        cand.push_back(i);
```

```cpp

    if (cand.empty())
      return true;

    auto x = cand[uniform_int_distribution<int>{0, (int) cand.size() - 1}(mt)];

    vector<ld> len(a.size(), numeric_limits<ld>::max());

    for (auto i: ranges::iota_view(0, ssize(len)))
      if (a[i][x] < -eps)
        len[i] = a[i].back() / -a[i][x];

    auto wh = int(ranges::min_element(len) - len.begin());

    if (len[wh] == numeric_limits<ld>::max())
      return false;

    pivot(a, b, func, wh, x);
  }
}

enum results
{
  NO_SOLUTION, UNBOUNDED, BOUNDED
};

/*
 * Solving system of linear inequalities in the form
 * $a * x <= rhs$
 * $x >= 0$
 * $costs * x -> max$
 * assumes at least one inequality and at least one variable
 * */
results global_solve(vector<vector<ld>> a, const vector<ld> &rhs, const vector<ld> &costs, vector<ld> &ans)
{
  assert(!a.empty() && a.size() == rhs.size() && !costs.empty() && ans.size() == costs.size());
  const auto m = costs.size() + a.size() + 2;

  for (auto i: ranges::iota_view(0, ssize(a)))
  {
    auto &row = a[i];

    row /= -1; // just finding inverse
    row.resize(m);
    row.back() = rhs[i];
    row.rbegin()[1] = 1;
  }

  vector<ld> func(m), lambda(m);
  vector<int> b(a.size());

  iota(b.begin(), b.end(), (int) costs.size());

  lambda.rbegin()[1] = -1;
  for (auto j: ranges::iota_view(0, ssize(costs)))
    func[j] = costs[j];

  auto wh = int(ranges::min_element(rhs) - rhs.begin());

  if (rhs[wh] < 0)
  {
```

```
    pivot(a, b, lambda, wh, (int) lambda.size() - 2);

    auto q = simplex(a, b, lambda);

    assert(q);
  }

  wh = int(ranges::find(b, (int) lambda.size() - 2) - b.begin());

  if (!eps_zero(lambda.back()))
    return NO_SOLUTION;

  if (wh != size(b))
  {
    if (!eps_zero(a[wh].back()))
      return NO_SOLUTION;

    auto q = int(ranges::find_if(a[wh], eps_nonneg) - a[wh].begin());

    if (q != ssize(a[wh]))
    {
      pivot(a, b, lambda, wh, q);
    }
    else
    {
      q = int(ranges::max_element(a[wh], cmp_abs) - a[wh].begin());

      if (!eps_zero(a[wh][q]))
        pivot(a, b, lambda, wh, q);
    }
  }

  for (auto &row: a)
    row.rbegin()[1] = 0;

  for (auto i: ranges::iota_view(0, ssize(b)))
    basis_change(func, a[i], b[i]);

  if (!simplex(a, b, func))
    return UNBOUNDED;

  for (auto i: ranges::iota_view(0, ssize(a)))
    if (b[i] < ssize(ans))
      ans[b[i]] = a[i].back();

  return BOUNDED;
}
```

14 OEIS

14.1 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

14.2 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