

**Содержание**

<b>1 Setup &amp; Scripts</b>	<b>2</b>
1.1 CMake . . . . .	2
1.2 wipe.sh . . . . .	2
1.3 Stack size & Profiling . . . . .	2
<b>2 Language specific</b>	<b>2</b>
2.1 C++ . . . . .	2
2.1.1 G++ builtins . . . . .	2
2.1.2 Custom Hash . . . . .	2
2.1.3 Allocator . . . . .	3
2.2 Python . . . . .	4
<b>3 Geometry</b>	<b>4</b>
3.1 Пересечение прямых . . . . .	4
3.2 Касательные . . . . .	4
3.3 Пересечение полуплоскостей . . . . .	4
3.4 Формулы . . . . .	4
<b>4 Numbers</b>	<b>4</b>
<b>5 Graphs</b>	<b>5</b>
5.1 Weighted matroid intersection . . . . .	5
<b>6 Data structures</b>	<b>6</b>
6.1 Push-free segment tree . . . . .	6
6.2 Template DSU . . . . .	8
6.3 Link-Cut Tree . . . . .	8
<b>7 Strings</b>	<b>10</b>
7.1 Suffix Automaton . . . . .	10
7.2 Palindromic Tree . . . . .	10
<b>8 Number theory</b>	<b>11</b>
8.1 Chinese remainder theorem without overflows . . . . .	11
8.2 Integer points under a rational line . . . . .	12

<b>9 Nimbers</b>	<b>12</b>
<b>10 Flows, etc.</b>	<b>13</b>
10.1 Hungarian Algorithm . . . . .	13
10.2 Circulation . . . . .	14
10.3 Global Min-Cut . . . . .	14
<b>11 The Elder Scrolls</b>	<b>15</b>
11.1 Dominator Tree . . . . .	15
11.2 Fast LCS . . . . .	16
11.3 Fast Subset Convolution . . . . .	16
11.4 Berlekamp-Massey . . . . .	17
11.5 Inverse of a Perturbed Matrix . . . . .	17
<b>12 Karatsuba</b>	<b>18</b>
<b>13 Hard Algorithms</b>	<b>19</b>
13.1 Two Strong Chinese . . . . .	19
13.2 Simplex . . . . .	22
<b>14 OEIS</b>	<b>25</b>
14.1 Числа Белла . . . . .	25
14.2 Числа Каталана . . . . .	25

# 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 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

```

1 namespace std {
2     template <
3     struct hash<pnt> {

```

```

4  std::size_t operator()(pnt const &s) const noexcept {
5      return std::hash<ll>{}(s.first * ll(1ull << 32u) +
6                          s.second);
7  }
8  };
9  } // namespace std

```

### 2.1.3 Allocator

```

1  template <size_t sz>
2  class chunk_alloc {
3  public:
4      static constexpr auto chunk_size = sz;
5
6  private:
7      using chunk_t = array<byte, chunk_size>;
8
9      deque<chunk_t> mem;
10     stack<void*> emp;
11
12  public:
13     void* allocate() {
14         if (emp.empty())
15             emp.push(reinterpret_cast<void*>(&mem.emplace_back()));
16
17         auto ans = emp.top();
18         emp.pop();
19
20         return ans;
21     }
22
23     void deallocate(void* p) noexcept { emp.push(p); }
24 };
25
26 chunk_alloc<64> pool;
27

```

```

28 template <class T>
29 struct dummy_alloc {
30     using value_type = T;
31
32     dummy_alloc() noexcept = default;
33
34     template <class U>
35     explicit dummy_alloc(const dummy_alloc<U> &) noexcept {}
36
37     T* allocate(std::size_t n) {
38         if constexpr (sizeof(value_type) ==
39                     decltype(pool)::chunk_size)
40             return static_cast<T*>(pool.allocate());
41         else
42             return static_cast<T*>(
43                 ::operator new(n * sizeof(value_type)));
44     }
45
46     void deallocate(T* p, std::size_t n) {
47         if constexpr (sizeof(value_type) ==
48                     decltype(pool)::chunk_size)
49             return pool.deallocate(p);
50         else
51             ::delete (p);
52     }
53 };
54
55 template <class T, class U>
56 constexpr bool operator==(const dummy_alloc<T> &,
57                          const dummy_alloc<U> &) noexcept {
58     return true;
59 }
60
61 template <class T, class U>
62 constexpr bool operator!=(const dummy_alloc<T> &,

```

```

63         const dummy_alloc<U> &) noexcept {
64     return false;
65 }

```

## 2.2 Python

```

1  # stack size
2  import sys
3
4  sys.setrecursionlimit(10**6)
5
6  # memoize
7  import functools
8
9  @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  $\frac{xr_1}{r_1 \pm r_2}$ .  $x$  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.

## 3.4 Формулы

Площадь поверхности сферы  $4\pi R^2$ . Объем шара  $\frac{4}{3}\pi R^3$ . Площадь шапки  $2\pi Rh$ , объем  $\frac{\pi h(3a^2 + h^2)}{6}$ , где  $h$  — высота,  $a$  — радиус шапки. Объем тетраэдра  $\frac{1}{6}$  на определитель. В общем случае площадь  $S_{n-1}$  и объем  $V_n$  шарика в  $\mathbb{R}^n$  можно найти по формуле  $S_{n-1} = nC_n R^{n-1}$ ,  $V_n = C_n R^n$ , где  $C_n = \frac{\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2} + 1)}$ . Или альтернативно  $C_{2k} = \frac{\pi^k}{k!}$ ,  $C_{2k+1} = \frac{2^{k+1}\pi^k}{(2k+1)!!}$ . Также, должны быть верны формулы  $\frac{V_n}{S_{n-1}} = \frac{R}{n}$ ,  $\frac{S_{n+1}}{V_n} = 2\pi R$ .

## 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

```

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 auto expand = [&](T in) → T {
13     vector<int> ids;
14     for (int i = 0; i < int(es.size()); i++)
15         if (in[i]) ids.push_back(i);
16
17     vector<int> from, to;
18     /// Given a set that is independent in both matroids, answers

```

```

19     /// queries "If we add i-th element to the set, will it still
20     /// be independent in red/blue matroid?". Usually can be done
21     /// quickly.
22     can_extend full_can(ids, n, es);
23
24     for (int i = 0; i < int(es.size()); i++)
25         if (!in[i]) {
26             auto new_ids = ids;
27             new_ids.push_back(i);
28
29             auto is_red = full_can.extend_red(i, es);
30             auto is_blue = full_can.extend_blue(i, es);
31
32             if (is_blue) from.push_back(i);
33             if (is_red) to.push_back(i);
34
35             if (is_red && is_blue) {
36                 T swp_mask = in;
37                 swp_mask.flip(i);
38                 return swp_mask;
39             }
40         }
41
42     vector<vector<int>> g(es.size());
43     for (int j = 0; j < int(es.size()); j++)
44         if (in[j]) {
45             auto new_ids = ids;
46             auto p = find(new_ids.begin(), new_ids.end(), j);
47             assert(p ≠ new_ids.end());
48             new_ids.erase(p);
49
50             can_extend cur(new_ids, n, es);
51
52             for (int i = 0; i < int(es.size()); i++)
53                 if (!in[i]) {

```

```

54         if (cur.extend_red(i, es)) g[i].push_back(j);
55         if (cur.extend_blue(i, es)) g[j].push_back(i);
56     }
57 }
58
59 auto get_cost = [&](int x) {
60     const int cost = (!in[x] ? e[x].w : -e[x].w);
61     return (ver + 1) * cost - 1;
62 };
63
64 const int inf = int(1e9);
65 vector<int> dist(ver, -inf), prev(ver, -1);
66 for (int x : from) dist[x] = get_cost(x);
67
68 queue<int> q;
69
70 vector<int> used(ver);
71 for (int x : from) {
72     q.push(x);
73     used[x] = 1;
74 }
75
76 while (!q.empty()) {
77     int cur = q.front();
78     used[cur] = 0;
79     q.pop();
80
81     for (int to : g[cur]) {
82         int cost = get_cost(to);
83         if (dist[to] < dist[cur] + cost) {
84             dist[to] = dist[cur] + cost;
85             prev[to] = cur;
86             if (!used[to]) {
87                 used[to] = 1;
88                 q.push(to);

```

```

89     }
90 }
91 }
92 }
93
94 int best = -inf, where = -1;
95 for (int x : to) {
96     if (dist[x] > best) {
97         best = dist[x];
98         where = x;
99     }
100 }
101
102 if (best == -inf) return pair<T, int>(cur_set, best);
103
104 while (where != -1) {
105     cur_set ^= (T(1) << where);
106     where = prev[where];
107 }
108
109 while (best % (ver + 1)) best++;
110 best /= (ver + 1);
111
112 assert(oracle(cur_set, red) && oracle(cur_set, blue));
113 return pair<T, int>(cur_set, best);
114 };

```

## 6 Data structures

### 6.1 Push-free segment tree

```

1 template <class Val, class Change, Change one = Change{}>
2 class pushfreesemtree {
3     vector<pair<Val, Change>> arr;
4

```

```

5  void upd(size_t v) {
6      arr[v].first = (arr[2 * v].first + arr[2 * v + 1].first) *
7          arr[v].second;
8  }
9
10 public:
11 explicit pushfreeseptree(size_t n = 0)
12     : arr(2 * n + 2, {Val{}, one}) {}
13
14 template <class It>
15 explicit pushfreeseptree(It be, It en)
16     : arr(2 * distance(be, en) + 2, {Val{}, one}) {
17     transform(be, en, arr.begin() + ssize(arr) / 2,
18         [](auto x) {
19             return pair{Val{x}, one};
20         });
21
22     for (int i = ssize(arr) / 2 - 1; i > 0; i--) upd(i);
23 }
24
25 auto segmult(const Change &x, size_t l, size_t r) {
26     l += arr.size() / 2;
27     r += arr.size() / 2;
28
29     while (true) {
30         if (l < r) {
31             if (l & 1u) {
32                 arr[l].first *= x;
33                 arr[l].second *= x;
34             }
35             if (r & 1u) {
36                 arr[r - 1].first *= x;
37                 arr[r - 1].second *= x;
38             }
39         }
40
41         l = (l + 1) / 2;
42         r /= 2;
43
44         if (r == 0) break;
45
46         upd(l - 1);
47         upd(r);
48     }
49 }
50
51 [[nodiscard]] Val segsum(size_t l, size_t r) const {
52     l += arr.size() / 2;
53     r += arr.size() / 2;
54
55     Val ans1{}, ansr{};
56
57     while (true) {
58         if (l < r) {
59             if (l & 1u) ans1 = ans1 + arr[l].first;
60             if (r & 1u) ansr = arr[r - 1].first + ansr;
61         }
62
63         l = (l + 1) / 2;
64         r /= 2;
65
66         if (r == 0) break;
67
68         ans1 *= arr[l - 1].second;
69         ansr *= arr[r].second;
70     }
71
72     return ans1 + ansr;
73 }
74 };

```

## 6.2 Template DSU

```

1  template <class... Types>
2  class dsu {
3      vector<int> par, siz;
4      tuple<Types...> items;
5
6      template <size_t... t>
7      void merge(int a, int b, std::index_sequence<t...>) {
8          ((get<t>(items)(a, b)), ...);
9      }
10
11  public:
12      explicit dsu(int n, Types... args)
13          : par(n, -1), siz(n, 1), items(args...) {}
14
15      int get_class(int v) {
16          return par[v] == -1 ? v : par[v] = get_class(par[v]);
17      }
18
19      bool unite(int a, int b) {
20          a = get_class(a);
21          b = get_class(b);
22
23          if (a == b) return false;
24
25          if (siz[a] < siz[b]) swap(a, b);
26          siz[a] += siz[b];
27          par[b] = a;
28
29          merge(a, b, make_index_sequence<sizeof...(Types)>{});
30
31          return true;
32      }
33  };

```

## 6.3 Link-Cut Tree

```

1  class lct {
2      struct node {
3          using nodeptr = node *;
4
5          array<nodeptr, 2> ch{};
6          nodeptr par = nullptr;
7          size_t siz = 1;
8          bool rev = false;
9      };
10
11      using nodeptr = node::nodeptr;
12
13      static void reverse(const nodeptr &h) {
14          if (h != nullptr) h->rev = !h->rev;
15      }
16
17      static void push(node &h) {
18          if (h.rev) {
19              swap(h.ch.front(), h.ch.back());
20              h.rev = false;
21
22              for (auto it : h.ch) reverse(it);
23          }
24      }
25
26      static auto size(const nodeptr &h) {
27          return h == nullptr ? 0 : h->siz;
28      }
29
30      static void upd(node &h) {
31          h.siz = 1;
32
33          for (auto it : h.ch) {
34              h.siz += size(it);

```



```

35
36     if (it ≠ nullptr) it→par = &h;
37 }
38 }
39
40 static bool is_root(const node &h) {
41     return h.par == nullptr ||
42         find(h.par→ch.begin(), h.par→ch.end(), &h) =
43             h.par→ch.end();
44 }
45
46 static bool is_right(const node &h) {
47     assert(!is_root(h));
48     push(*h.par);
49     return get<1>(h.par→ch) == &h;
50 }
51
52 static void zig(node &h) {
53     assert(!is_root(h));
54
55     auto &p = *h.par;
56     push(p);
57     push(h);
58     auto pp = p.par;
59     bool ind = is_right(h);
60     auto &x = p.ch[ind];
61     auto &b = h.ch[!ind];
62
63     x = b;
64     b = &p;
65     h.par = pp;
66
67     upd(p);
68     upd(h);
69

```

```

70     if (pp ≠ nullptr)
71         for (auto &it : pp→ch)
72             if (it == &p) it = &h;
73 }
74
75 static void splay(node &h) {
76     push(h);
77     while (!is_root(h)) {
78         auto &p = *h.par;
79
80         if (is_root(p)) {
81             zig(h);
82         } else if (is_right(h) == is_right(p)) {
83             zig(p);
84             zig(h);
85         } else {
86             zig(h);
87             zig(h);
88         }
89     }
90 }
91
92 static void expose(node &h) {
93     splay(h);
94
95     while (h.par ≠ nullptr) {
96         auto &p = *h.par;
97         splay(p);
98         get<1>(p.ch) = &h;
99         upd(p);
100        splay(h);
101    }
102 }
103 };

```

## 7 Strings

### 7.1 Suffix Automaton

```

1  class tomato {
2      struct node {
3          array<int, 26> nxt{};
4          int link = -1, len = 0;
5
6          explicit node(int len = 0) : len(len) {
7              ranges::fill(nxt, -1);
8          }
9
10         explicit node(int len, node p)
11             : nxt(p.nxt), len(len), link(p.link) {}
12     };
13
14     vector<node> mem = {node(0)};
15     int last = 0;
16
17 public:
18     explicit tomato(string_view sv = "") {
19         for (auto it : sv) (*this) += it;
20     }
21
22     tomato &operator+=(char ch) {
23         const int ind = ch - 'a';
24         auto new_last = int(mem.size());
25         mem.emplace_back(mem[last].len + 1);
26
27         auto p = last;
28         while (p ≥ 0 && mem[p].nxt[ind] == -1) {
29             mem[p].nxt[ind] = new_last;
30             p = mem[p].link;
31         }
32     }

```

```

33     if (p ≠ -1) {
34         const int q = mem[p].nxt[ind];
35         if (mem[p].len + 1 == mem[q].len) {
36             mem[new_last].link = q;
37         } else {
38             auto clone = int(mem.size());
39             mem.emplace_back(mem[p].len + 1, mem[q]);
40             mem[q].link = clone;
41             mem[new_last].link = clone;
42
43             while (p ≥ 0 && mem[p].nxt[ind] == q) {
44                 mem[p].nxt[ind] = clone;
45                 p = mem[p].link;
46             }
47         }
48     } else
49         mem[new_last].link = 0;
50
51     last = new_last;
52
53     return *this;
54 }
55 };

```

### 7.2 Palindromic Tree

```

1  class treert {
2      struct node {
3          array<int, 26> nxt;
4          int par, link, siz;
5
6          node(int siz, int par, int link)
7              : par(par),
8                link(link == -1 ? 1 : link),
9                siz(siz) // note -1 case
10         {

```

```

11     fill(nxt.begin(), nxt.end(), -1);
12 }
13 };
14
15 vector<node> mem;
16 vector<int> suff; // longest palindromic suffix
17
18 public:
19 treert(const string &str) : suff(str.size()) {
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         while (pos - 1 - mem[st].siz < 0 ||
26             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         last = link_walk(last, i);
34         auto ind = str[i] - 'a';
35
36         if (mem[last].nxt[ind] == -1) {
37             // order is important
38             mem.emplace_back(
39                 mem[last].siz + 2, last,
40                 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 };

```

## 8 Number theory

### 8.1 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     if (b == 0) {
7         x = 1, y = 0;
8         return a;
9     }
10
11     T res = gcdext(b, a % b, y, x);
12     y -= x * (a / b);
13     return res;
14 }
15
16 // Returns true if system x = r1 (mod m1), x = r2 (mod m2) has
17 // solutions false otherwise. In first case we know exactly
18 // that x = r (mod m)
19
20 bool crt(T r1, T m1, T r2, T m2, T &r, T &m) {
21     if (m2 > m1) {
22         swap(r1, r2);
23         swap(m1, m2);
24     }
25
26     T g = __gcd(m1, m2);

```

```

27  if ((r2 - r1) % g != 0) return false;
28
29  T c1, c2;
30  auto nrem = gcdext(m1 / g, m2 / g, c1, c2);
31  assert(nrem == 1);
32  assert(c1 * (m1 / g) + c2 * (m2 / g) == 1);
33  T a = c1;
34  a *= (r2 - r1) / g;
35  a %= (m2 / g);
36  m = m1 / g * m2;
37  r = a * m1 + r1;
38  r = r % m;
39  if (r < 0) r += m;
40
41  assert(r % m1 == r1 && r % m2 == r2);
42  return true;
43 }

```

## 8.2 Integer points under a rational line

```

1  // integer  $(x, y) : 0 \leq x < n, 0 < y \leq (kx + b)/d$ 
2  // (real division)
3  // In other words,  $\sum_{x=0}^{n-1} \lfloor (kx + b)/d \rfloor$ 
4  ll trapezoid(ll n, ll k, ll b, ll d) {
5      if (k == 0) return (b / d) * n;
6      if (k >= d || b >= d)
7          return (k / d) * n * (n - 1) / 2 + (b / d) * n +
8              trapezoid(n, k % d, b % d, d);
9      return trapezoid((k * n + b) / d, d, (k * n + b) % d, k);
10 }

```

## 9 Nimbers

```

1  template <class T, int lvl>
2  pair<T, T> split(T x) {

```

```

3      return {x >> lvl, x & ((T{1} << lvl) - 1)};
4  }
5
6  template <class T, int lvl>
7  T combine(T a, T b) {
8      return (a << lvl) | b;
9  }
10
11 template <class T, int lvl = 8 * sizeof(T)>
12 T nim_hmul(T x) {
13     constexpr int half = lvl / 2;
14     if constexpr (lvl == 1) return x;
15
16     auto [a, b] = split<T, half>(x);
17
18     return combine<T, half>(
19         nim_hmul<T, half>(a ^ b),
20         nim_hmul<T, half>(nim_hmul<T, half>(a)));
21 }
22
23 template <class T, int lvl = 8 * sizeof(T)>
24 T nim_mul(T x, T y) {
25     constexpr int half = lvl / 2;
26     if constexpr (lvl == 1) return x & y;
27
28     auto [a, b] = split<T, half>(x);
29     auto [c, d] = split<T, half>(y);
30
31     auto ac = nim_mul<T, half>(a, c);
32     auto bd = nim_mul<T, half>(b, d);
33     auto hp = nim_mul<T, half>(a ^ b, c ^ d) ^ bd;
34
35     return combine<T, half>(hp, bd ^ nim_hmul<T, half>(ac));
36 }
37

```

```

38 template <class T, int lvl = 8 * sizeof(T)>
39 T nim_sqr(T x) {
40     return nim_mul<T, lvl>(x, x);
41 }
42
43 template <class T, int lvl = 8 * sizeof(T)>
44 T nim_sqrt(T x) {
45     constexpr int half = lvl / 2;
46     if constexpr (lvl == 1) return x;
47
48     auto [a, b] = split<T, half>(x);
49
50     return combine<T, half>(
51         nim_sqrt<T, half>(a),
52         nim_sqrt<T, half>(nim_hmul<T, half>(a ^ b)));
53 }
54
55 template <class T, int lvl = 8 * sizeof(T)>
56 T nim_recip(T x) {
57     constexpr int half = lvl / 2;
58     if constexpr (lvl == 1) return x;
59
60     auto [a, b] = split<T, half>(x);
61
62     auto ad = nim_mul<T, half>(a ^ b, b);
63     auto bc = nim_hmul<T, half>(nim_sqr<T, half>(a));
64     auto det_recip = nim_recip<T, half>(ad ^ bc);
65
66     return combine<T, half>(nim_mul(a, det_recip),
67                             nim_mul(a ^ b, det_recip));
68 }

```

## 10 Flows, etc.

### 10.1 Hungarian Algorithm

```

1  ld Hungarian(const vector<vector<ld>> &matr) {
2      vector<int> lb(matr.size(), -1), rb(matr[0].size(), -1);
3      vector<ld> rows(matr.size()), cols(rb.size());
4
5      for (int v = 0; v < ssize(matr); v++) {
6          vector<bool> lused(lb.size()), rused(rb.size());
7          vector<int> par(rb.size(), -1);
8          vector<pair<ld, int>> w(rb.size(),
9                                  {numeric_limits<ld>::max(), -1});
10
11         auto add_row = [&](int i) {
12             lused[i] = true;
13
14             for (int j = 0; j < ssize(w); j++)
15                 remin(w[j], {matr[i][j] + rows[i] + cols[j], i});
16         };
17
18         add_row(v);
19
20         while (true) {
21             int j = -1;
22
23             for (int k = 0; k < ssize(rb); k++)
24                 if (!rused[k] && (j == -1 || w[k] < w[j])) j = k;
25
26             auto [x, i] = w[j];
27
28             for (int k = 0; k < ssize(lused); k++)
29                 if (!lused[k]) rows[k] += x;
30             for (int k = 0; k < ssize(rused); k++)
31                 if (!rused[k]) {
32                     cols[k] -= x;

```

```

33     w[k].first -= x;
34 }
35
36 par[j] = i;
37 rused[j] = true;
38
39 if (rb[j] == -1) {
40     while (j != -1) {
41         rb[j] = par[j];
42         auto nxt = lb[par[j]];
43         lb[par[j]] = j;
44         j = nxt;
45     }
46
47     break;
48 }
49
50 add_row(rb[j]);
51 }
52 }
53
54 ld ans = 0;
55
56 for (int i = 0; i < ssize(lb); i++)
57     if (auto j = lb[i]; j != -1) ans += matr[i][j];
58
59 return ans;
60 }

```

## 10.2 Circulation

Можно делать алгоритм Клейна: пушим отрицательные циклы, пока они есть. ММСС: бинпоиском в Фордом-Беллманом ищем отрицательный цикл минимального среднего веса, по нему пушим. Capacity Scaling: идем по битам от больших к меньшим, добавляем по одному ребру. Один шаг такого алгоритма похож на один шаг минкоста с

Дейкстрой с потенциалами.

## 10.3 Global Min-Cut

```

1  int StoerWagner(vector<vector<int>> matr) {
2      int ans = numeric_limits<int>::max();
3
4      auto work = [&]() -> pair<int, int> {
5          vector<int> d(matr.size());
6
7          int q;
8
9          for (int i = 0; i + 1 < int(matr.size()); i++) {
10             q = int(max_element(d.begin(), d.end()) - d.begin());
11             d[q] = numeric_limits<int>::lowest();
12
13             for (int j = 0; j < int(matr.size()); j++)
14                 d[j] += matr[q][j];
15         }
16
17         auto w = int(max_element(d.begin(), d.end()) - d.begin());
18
19         ans = min(ans, d[w]);
20
21         return {q, w};
22     };
23
24     while (matr.size() > 1) {
25         int a, b;
26
27         tie(a, b) = work();
28
29         if (b < a) swap(a, b);
30
31         for (int i = 0; i < int(matr.size()); i++)
32             if (i != a && i != b) {

```

```

33     matr[i][a] += matr[i][b];
34     matr[a][i] += matr[b][i];
35 }
36
37 for (auto &row : matr) row.erase(row.begin() + b);
38 matr.erase(matr.begin() + b);
39 }
40
41 return ans;
42 }

```

## 11 The Elder Scrolls

### 11.1 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) bucket[sdom[i]].pb(i);
47
48            for (int w : bucket[i]) {
49                int v = find(w);
50                dom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
51            }
52
53            if (i > 0) unite(par[i], i);
54        }
55    }

```

```

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

```

## 11.2 Fast LCS

```

1  for (char ch : s) { // main cycle
2      const int pos = ch - 'a';
3      bs next = sum(row, row & has[pos]) | (row & rev[pos]);
4      cnt += next[m];
5      next[m] = 0;
6      row = next;

```

```

7  }

```

## 11.3 Fast Subset Convolution

```

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

```



```

33
34     forn(cnt, bits + 1) {
35         mobius(trans_a[cnt], bits, +1);
36         mobius(trans_b[cnt], bits, +1);
37     }
38
39     // Not really a valid ranked mobius transform! But algorithm
40     // works anyway
41
42     forn(i, bits + 1) forn(j, bits - i + 1) forn(mask, 1 << bits)
43         add(trans_res[i + j][mask],
44             mult(trans_a[i][mask], trans_b[j][mask]));
45
46     forn(cnt, bits + 1) mobius(trans_res[cnt], bits, -1);
47
48     forn(mask, 1 << bits) {
49         const int cnt = __builtin_popcount(mask);
50         a[mask] = trans_res[cnt][mask];
51     }
52
53     return a;
54 }

```

#### 11.4 Berlekamp-Massey

```

1  template <typename T>
2  vector<T> berlekamp(const vector<T> &s) {
3      vector<T> c, oldC;
4      int f = -1;
5      for (int i = 0; i < (int)s.size(); i++) {
6          T delta = s[i];
7          for (int j = 1; j ≤ (int)c.size(); j++)
8              delta -= c[j - 1] * s[i - j];
9          if (delta == 0) continue;
10
11         if (f == -1) {

```

```

12             c.resize(i + 1);
13             f = i;
14         } else {
15             vector<T> d = oldC;
16             for (T &x : d) x = -x;
17             d.insert(d.begin(), 1);
18             T df1 = 0;
19             for (int j = 1; j ≤ (int)d.size(); j++)
20                 df1 += d[j - 1] * s[f + 1 - j];
21             assert(df1 ≠ 0);
22             T coef = delta / df1;
23             for (T &x : d) x *= coef;
24
25             vector<T> zeros(i - f - 1);
26             zeros.insert(zeros.end(), d.begin(), d.end());
27             d = zeros;
28             vector<T> temp = c;
29             c.resize(max(c.size(), d.size()));
30             for (int j = 0; j < (int)d.size(); j++) c[j] += d[j];
31
32             if (i - (int)temp.size() > f - (int)oldC.size()) {
33                 oldC = temp;
34                 f = i;
35             }
36         }
37     }
38
39     return c;
40 }

```

#### 11.5 Inverse of a Perturbed Matrix

- $(I + UV)^{-1} = I - U(I + VU)^{-1}V$ .
- $(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$
- $(A + uv^T)^{-1} = A^{-1} - (A^{-1}uv^TA^{-1})/(1 + v^TA^{-1}u)$

- $v^T A^{-1} u = v^T x$ , где  $x$  — решение  $Ax = u$ .

## 12 Karatsuba

```

1 // function Karatsuba (and stupid as well) computes c += a * b,
2 // not c = a * b
3
4 using hvect = vector<modulo<>>::iterator;
5 using hcvect = vector<modulo<>>::const_iterator;
6
7 void add(hvect abegin, hcvect aend, hvect ans) {
8     for (auto it = abegin; it != aend; ++it, ++ans) *ans += *it;
9 }
10
11 void sub(hvect abegin, hcvect aend, hvect ans) {
12     for (auto it = abegin; it != aend; ++it, ++ans) *ans -= *it;
13 }
14
15 void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans) {
16     for (int i = 0; i < siz; i++)
17         for (int j = 0; j < siz; j++)
18             *(ans + i + j) += *(abegin + i) * *(bbegin + j);
19 }
20
21 void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin,
22                hvect ans, hvect small, hvect big, hvect sum) {
23     assert((siz & (siz - 1)) == 0);
24
25     if (siz ≤ 32) {
26         stupid(siz, abegin, bbegin, ans);
27
28         return;
29     }
30
31     auto amid = abegin + siz / 2, aend = abegin + siz;

```

```

32     auto bmid = bbegin + siz / 2, bend = bbegin + siz;
33     auto smid = sum + siz / 2, send = sum + siz;
34
35     fill(small, small + siz, 0);
36     Karatsuba(siz / 2, abegin, bbegin, small, small + siz,
37              big + siz, sum);
38     fill(big, big + siz, 0);
39     Karatsuba(siz / 2, amid, bmid, big, small + siz, big + siz,
40              sum);
41
42     copy(abegin, amid, sum);
43     add(amid, aend, sum);
44     copy(bbegin, bmid, sum + siz / 2);
45     add(bmid, bend, sum + siz / 2);
46
47     Karatsuba(siz / 2, sum, smid, ans + siz / 2, small + siz,
48              big + siz, send);
49
50     add(small, small + siz, ans);
51     sub(small, small + siz, ans + siz / 2);
52     add(big, big + siz, ans + siz);
53     sub(big, big + siz, ans + siz / 2);
54 }
55
56 void mult(vector<modulo<>> a, vector<modulo<>> b,
57           vector<modulo<>> &c) {
58     a.resize(up(max(a.size(), b.size()), 0));
59     b.resize(a.size(), 0);
60
61     c.resize(max(c.size(), a.size() * 2), 0);
62
63     vector<modulo<>> small(2 * a.size());
64     auto big = small;
65     auto sum = small;
66

```

```

67     Karatsuba(a.size(), a.begin(), b.begin(), c.begin(),
68             small.begin(), big.begin(), sum.begin());
69 }

```

## 13 Hard Algorithms

### 13.1 Two Strong Chinese

```

1  template <class T, class Add>
2  class skew_heap {
3      struct node {
4          using nodeptr = unique_ptr<node>;
5
6          nodeptr l = nullptr, r = nullptr;
7          T x;
8
9          explicit node(T x = {}) : x(x) {}
10 };
11
12 using nodeptr = typename node::nodeptr;
13
14 static nodeptr merge(nodeptr &a, nodeptr &b) {
15     if (a == nullptr) return std::move(b);
16     if (b == nullptr) return std::move(a);
17     if (b->x < a->x) return merge(std::move(b), std::move(a));
18
19     auto tmp = merge(std::move(a->r), std::move(b));
20     a->r = std::move(a->l);
21     a->l = std::move(tmp);
22
23     return std::move(a);
24 }
25
26 void add_to_all(nodeptr &a, Add x) {
27     if (a == nullptr) return;

```

```

28
29     a->x += x;
30     add_to_all(a->l, x);
31     add_to_all(a->r, x);
32 }
33
34 nodeptr root = nullptr;
35 size_t siz = 0;
36 Add to_add{};
37
38 public:
39     void add(Add x) { to_add += x; }
40
41     [[nodiscard]] T top() const { return root->x + to_add; }
42
43     [[nodiscard]] auto size() const { return siz; }
44
45     [[nodiscard]] auto empty() const { return size() == 0; }
46
47     void pop() {
48         auto q = merge(std::move(root->l), std::move(root->r));
49         siz--;
50         root = std::move(q);
51     }
52
53     void merge(skew_heap &rhs) {
54         if (size() < rhs.size()) swap(*this, rhs);
55
56         siz += rhs.siz;
57         rhs.siz = 0;
58         rhs.add_to_all(rhs.root, rhs.to_add - to_add);
59         auto q = merge(std::move(root), std::move(rhs.root));
60         root = std::move(q);
61     }
62

```

```

63  void push(T x) {
64      skew_heap sh;
65      sh.root = make_unique<node>(x);
66      sh.siz = 1;
67
68      merge(std::move(sh));
69  }
70 };
71
72 struct edge {
73     ll w;
74     int to;
75     int id;
76
77     strong_ordering operator<=>(const edge &rhs) const {
78         return w <=> rhs.w;
79     }
80
81     edge &operator+=(ll rhs) {
82         w += rhs;
83
84         return *this;
85     }
86
87     edge operator+(ll rhs) const {
88         return edge{w + rhs, to, id};
89     }
90 };
91
92 enum color_t { White = 0, Grey, Black, Cycle };
93
94 vector<int> solve(size_t n,
95                 const vector<tuple<int, int, int>> &edges,
96                 int root = 0) {
97     vector<skew_heap<edge, ll>> rev(n);

```

```

98
99     for (int i = 0; i < (int)edges.size(); i++) {
100         auto [a, b, w] = edges[i];
101
102         if (b != root) rev[b].push(edge{w, a, i});
103     }
104
105     auto mrg = [&](int a, int b) {
106         rev[a].merge(std::move(rev[b]));
107     };
108
109     dsu cc(n, mrg);
110
111     vector<color_t> color(rev.size());
112     color[root] = Black;
113
114     vector<int> ids;
115
116     function<bool(int)> dfs = [&](int v) -> bool {
117         v = cc.get_class(v);
118
119         if (color[v] == Black) return false;
120
121         if (color[v] == Grey) {
122             color[v] = Cycle;
123
124             return true;
125         }
126         color[v] = Grey;
127
128         while (true) {
129             while (!rev[v].empty() &&
130                 cc.get_class(rev[v].top().to) == v)
131                 rev[v].pop();
132

```

```

133     assert(
134         !rev[v].empty()); // assume that the answer exists
135     auto [w, to, id] = rev[v].top();
136
137     ids.emplace_back(
138         id); // ans += w; if the certificate is not needed
139
140     rev[v].add(-w);
141
142     if (dfs(to)) {
143         if (color[v] != Cycle) {
144             cc.unite(v, to);
145             color[cc.get_class(v)] = Cycle;
146
147             return true;
148         } else {
149             v = cc.get_class(v);
150
151             color[v] = Grey;
152         }
153     } else {
154         color[v] = Black;
155
156         return false;
157     }
158 }
159 };
160
161 for (int i = 0; i < (int)rev.size(); i++) dfs(i);
162
163 // finding answer, similar to Prim
164 vector<vector<int>> gr(n);
165
166 for (int i = 0; i < int(ids.size()); i++) {
167     auto [a, b, _] = edges[ids[i]];

```

```

168
169     gr[a].push_back(i);
170 }
171
172 minheap<int> pq(gr[root].begin(), gr[root].end());
173 vector<bool> used(n);
174 used[root] = true;
175
176 vector<int> ans;
177
178 while (!pq.empty()) {
179     auto i = pq.top();
180     pq.pop();
181     auto v = get<1>(edges[ids[i]]);
182
183     if (used[v]) continue;
184     used[v] = true;
185
186     ans.push_back(ids[i]);
187
188     for (auto it : gr[v]) pq.push(it);
189 }
190
191 return ans;
192 }
193
194 void dfs(const vector<vector<pair<int, int>>> &gr,
195         vector<bool> &used, int v) {
196     if (used[v]) return;
197     used[v] = true;
198
199     for (auto [u, w] : gr[v]) dfs(gr, used, u);
200 }
201
202 void solve(istream &cin = std::cin,

```

```

203         ostream &cout = std::cout) {
204     int n, m;
205
206     cin >> n >> m;
207
208     vector<tuple<int, int, int>> edges(m);
209     vector<vector<pair<int, int>>> gr(n);
210
211     for (int i = 0; i < m; i++) {
212         auto &a, b, w = edges[i];
213
214         cin >> a >> b >> w;
215         a--;
216         b--;
217
218         gr[a].emplace_back(b, w);
219     }
220
221     vector<bool> used(gr.size());
222
223     dfs(gr, used, 0);
224
225     if (ranges::count(used, false)) {
226         cout << "NO" << endl;
227
228         return;
229     }
230
231     cout << "YES" << endl;
232
233     auto ids = solve(gr.size(), edges);
234
235     ll ans = 0;
236
237     for (auto it : ids) ans += get<2>(edges[it]);

```

```

238
239     for (auto &row : gr) row.clear();
240
241     for (auto it : ids) {
242         auto [a, b, w] = edges[it];
243
244         gr[a].emplace_back(b, w);
245     }
246
247     used.assign(used.size(), false);
248
249     dfs(gr, used, 0);
250
251     assert(ranges::count(used, false) == 0);
252
253     cout << ans << endl;
254 }

```

## 13.2 Simplex

```

1  mt19937 mt(736);
2
3  using ld = double;
4  constexpr ld eps = 1e-9;
5
6  bool eps_nonneg(ld x) { return x ≥ -eps; }
7
8  bool eps_zero(ld x) { return abs(x) ≤ eps; }
9
10 bool cmp_abs(ld a, ld b) { return abs(a) < abs(b); }
11
12 vector<ld> &add_prod(vector<ld> &lhs, const vector<ld> &rhs,
13                    ld x) {
14     assert(ssize(lhs) == ssize(rhs));
15
16     for (auto i : ranges::iota_view(0, ssize(lhs)))

```

```

17     lhs[i] += rhs[i] * x;
18
19     return lhs;
20 }
21
22 vector<ld> &operator=(vector<ld> &lhs, ld x) {
23     for (auto &it : lhs) it /= x;
24
25     return lhs;
26 }
27
28 void basis_change(vector<ld> &row, const vector<ld> &nd,
29                  int b) {
30     auto mult = row[b];
31
32     add_prod(row, nd, mult);
33
34     row[b] = 0;
35 }
36
37 void pivot(vector<vector<ld>> &a, vector<int> &b,
38            vector<ld> &func, int wh, int x) {
39     a[wh][b[wh]] = -1;
40     b[wh] = x;
41     auto den = -a[wh][x];
42     a[wh][x] = 0;
43     a[wh] /= den;
44
45     for (auto i : ranges::iota_view(0, ssize(a)))
46         if (i != wh) basis_change(a[i], a[wh], b[wh]);
47     basis_change(func, a[wh], b[wh]);
48 }
49
50 bool simplex(vector<vector<ld>> &a, vector<int> &b,
51              vector<ld> &func) {

```

```

52 while (true) {
53     vector<int> cand;
54
55     for (auto i : ranges::iota_view(0, ssize(func) - 1))
56         if (func[i] > eps) cand.push_back(i);
57
58     if (cand.empty()) return true;
59
60     auto x = cand[uniform_int_distribution<int>{
61         0, (int)cand.size() - 1}(mt)];
62
63     vector<ld> len(a.size(), numeric_limits<ld>::max());
64
65     for (auto i : ranges::iota_view(0, ssize(len)))
66         if (a[i][x] < -eps) len[i] = a[i].back() / -a[i][x];
67
68     auto wh = int(ranges::min_element(len) - len.begin());
69
70     if (len[wh] == numeric_limits<ld>::max()) return false;
71
72     pivot(a, b, func, wh, x);
73 }
74 }
75
76 enum results { NO_SOLUTION, UNBOUNDED, BOUNDED };
77
78 /*
79  * Solving system of linear inequalities in the form
80  * $a * x ≤ rhs$
81  * $x ≥ 0$
82  * $costs * x → max$
83  * assumes at least one inequality and at least one variable
84  * */
85 results global_solve(vector<vector<ld>> a,
86                     const vector<ld> &rhs,

```

```

87         const vector<ld> &costs,
88         vector<ld> &ans) {
89     assert(!a.empty() && a.size() == rhs.size() &&
90           !costs.empty() && ans.size() == costs.size());
91     const auto m = costs.size() + a.size() + 2;
92
93     for (auto i : ranges::iota_view(0, ssize(a))) {
94         auto &row = a[i];
95
96         row /= -1; // just finding inverse
97         row.resize(m);
98         row.back() = rhs[i];
99         row.rbegin()[1] = 1;
100    }
101
102    vector<ld> func(m), lambda(m);
103    vector<int> b(a.size());
104
105    iota(b.begin(), b.end(), (int)costs.size());
106
107    lambda.rbegin()[1] = -1;
108    for (auto j : ranges::iota_view(0, ssize(costs)))
109        func[j] = costs[j];
110
111    auto wh = int(ranges::min_element(rhs) - rhs.begin());
112
113    if (rhs[wh] < 0) {
114        pivot(a, b, lambda, wh, (int)lambda.size() - 2);
115
116        auto q = simplex(a, b, lambda);
117
118        assert(q);
119    }
120
121    wh =

```

```

122        int(ranges::find(b, (int)lambda.size() - 2) - b.begin());
123
124    if (!eps_zero(lambda.back())) return NO_SOLUTION;
125
126    if (wh != size(b)) {
127        if (!eps_zero(a[wh].back())) return NO_SOLUTION;
128
129        auto q = int(ranges::find_if(a[wh], eps_nonneg) -
130                          a[wh].begin());
131
132        if (q != ssize(a[wh])) {
133            pivot(a, b, lambda, wh, q);
134        } else {
135            q = int(ranges::max_element(a[wh], cmp_abs) -
136                      a[wh].begin());
137
138            if (!eps_zero(a[wh][q])) pivot(a, b, lambda, wh, q);
139        }
140    }
141
142    for (auto &row : a) row.rbegin()[1] = 0;
143
144    for (auto i : ranges::iota_view(0, ssize(b)))
145        basis_change(func, a[i], b[i]);
146
147    if (!simplex(a, b, func)) return UNBOUNDED;
148
149    for (auto i : ranges::iota_view(0, ssize(a)))
150        if (b[i] < ssize(ans)) ans[b[i]] = a[i].back();
151
152    return BOUNDED;
153 }

```



## 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