Содержание

1 Setup & Scripts **2**
1.1 CMake . . . 2
1.2 wipe.sh . . . 2
1.3 Stack size & Profiling . . . 2

2 Language specific **2**
2.1 C++ . . . 2
2.1.1 G++ builtins . . . 2
2.1.2 hash . . . 3
2.2 Python . . . 3

3 Geometry **3**
3.1 Пересечение прямых . . . 3
3.2 Касательные . . . 4
3.3 Пересечение полуплоскостей . . . 4

4 Numbers **4**

5 Graphs **4**
5.1 Weighted matroid intersection . . . 4

6 Data structures **7**
6.1 Push-free segment tree . . . 7
6.2 Template DSU . . . 9
6.3 Link-Cut Tree . . . 10

7 Strings **13**
7.1 Suffix Automaton . . . 13
7.2 Palindromic Tree . . . 14

8 Number theory **15**
8.1 Chinese remainder theorem without overflows . . . 15
8.2 Integer points under a rational line . . . 16

9 Something added at the last moment **17**
9.1 Dominator Tree . . . 17
9.2 Fast LCS . . . 18
9.3 Fast Subset Convolution . . . 20

10 Karatsuba **21**

11 Hard Algorithms **23**
11.1 Two Strong Chinese . . . 23
11.2 Simplex . . . 29

12 OEIS **33**
12.1 Числа Белла . . . 33
12.2 Числа Каталана . . . 33

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 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

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

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

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

```
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 {
14     vector<int> ids;
15     for (int i = 0; i < int(es.size()); i++)
16         if (in[i])
17             ids.push_back(i);
18
19     vector<int> from, to;
20     /// Given a set that is independent in both matroids, answers
21     /// queries "If we add i-th element to the set, will it still be
22     /// independent in red/blue matroid?". Usually can be done quickly.
23     can_extend full_can(ids, n, es);
24
25     for (int i = 0; i < int(es.size()); i++)
26         if (!in[i])
27         {
28             auto new_ids = ids;
29             new_ids.push_back(i);
30
31             auto is_red = full_can.extend_red(i, es);
32             auto is_blue = full_can.extend_blue(i, es);
33
34             if (is_blue)
35                 from.push_back(i);
36             if (is_red)
37                 to.push_back(i);
38
39             if (is_red && is_blue)
40             {
41                 T swp_mask = in;
42                 swp_mask.flip(i);
43                 return swp_mask;
44             }
45     }
46
47     vector<vector<int>> g(es.size());
48     for (int j = 0; j < int(es.size()); j++)
49         if (in[j])
50         {
51             auto new_ids = ids;
52             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})
        {}
```

```
14
15          template<class It>
16          explicit pushfreesegtree(It be, It en) : arr(2 * distance(be, en) + 2,
            ↪  {Val{}, one})
17          {
18                  transform(be, en, arr.begin() + ssize(arr) / 2, [](auto x)
19                  {
20                          return pair{Val{x}, one};
21                  });
22
23                  for (int i = ssize(arr) / 2 - 1; i > 0; i--)
24                          upd(i);
25          }
26
27          auto segmult(const Change &x, size_t l, size_t r)
28          {
29                  l += arr.size() / 2;
30                  r += arr.size() / 2;
31
32                  while (true)
33                  {
34                          if (l < r)
35                          {
36                                  if (l & 1u)
37                                  {
38                                          arr[l].first *= x;
39                                          arr[l].second *= x;
40                                  }
41                                  if (r & 1u)
42                                  {
43                                          arr[r - 1].first *= x;
44                                          arr[r - 1].second *= x;
45                                  }
46                          }
47
48                          l = (l + 1) / 2;
49                          r /= 2;
50
51                          if (r == 0)
52                                  break;
53
54                          upd(l - 1);
55                          upd(r);
56                  }
57          }
58
59          [[nodiscard]] Val segsum(size_t l, size_t r) const
60          {
61                  l += arr.size() / 2;
62                  r += arr.size() / 2;
```

```
63
64                  Val ansl{}, ansr{};
65
66                  while (true)
67                  {
68                          if (l < r)
69                          {
70                                  if (l & 1u)
71                                          ansl = ansl + arr[l].first;
72                                  if (r & 1u)
73                                          ansr = arr[r - 1].first + ansr;
74                          }
75
76                          l = (l + 1) / 2;
77                          r /= 2;
78
79                          if (r == 0)
80                                  break;
81
82                          ansl *= arr[l - 1].second;
83                          ansr *= arr[r].second;
84                  }
85
86                  return ansl + ansr;
87          }
88  };
```

6.2 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  };
```

6.3 Link-Cut Tree

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

```
29                              reverse(it);
30                      }
31              }
32
33              static auto size(const nodeptr &h)
34              {
35                      return h == nullptr ? 0 : h->siz;
36              }
37
38              static void upd(node &h)
39              {
40                      h.siz = 1;
41
42                      for (auto it: h.ch)
43                      {
44                              h.siz += size(it);
45
46                              if (it != nullptr)
47                                      it->par = &h;
48                      }
49              }
50
51              static bool is_root(const node &h)
52              {
53                      return h.par == nullptr || find(h.par->ch.begin(),
                        ↪  h.par->ch.end(), &h) == h.par->ch.end();
54              }
55
56              static bool is_right(const node &h)
57              {
58                      assert(!is_root(h));
59                      push(*h.par);
60                      return get<1>(h.par->ch) == &h;
61              }
62
63              static void zig(node &h)
64              {
65                      assert(!is_root(h));
66
67                      auto &p = *h.par;
68                      push(p);
69                      push(h);
70                      auto pp = p.par;
71                      bool ind = is_right(h);
72                      auto &x = p.ch[ind];
73                      auto &b = h.ch[!ind];
74
75                      x = b;
76                      b = &p;
77                      h.par = pp;
```

```
78
79                  upd(p);
80                  upd(h);
81
82                  if (pp ≠ nullptr)
83                          for (auto &it: pp→ch)
84                                  if (it == &p)
85                                          it = &h;
86          }
87
88          static void splay(node &h)
89          {
90                  push(h);
91                  while (!is_root(h))
92                  {
93                          auto &p = *h.par;
94
95                          if (is_root(p))
96                          {
97                                  zig(h);
98                          }
99                          else if (is_right(h) == is_right(p))
100                         {
101                                 zig(p);
102                                 zig(h);
103                         }
104                         else
105                         {
106                                 zig(h);
107                                 zig(h);
108                         }
109                 }
110         }
111
112         static void expose(node &h)
113         {
114                 splay(h);
115
116                 while (h.par ≠ nullptr)
117                 {
118                         auto &p = *h.par;
119                         splay(p);
120                         get<1>(p.ch) = &h;
121                         upd(p);
122                         splay(h);
123                 }
124         }
125 };
```

7 Strings

7.1 Suffix Automaton

```
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)
                {}
        };

        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

```
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;

```

```
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 };
```

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  {
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
```

```
// 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

```
// 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);
}
```

9 Something added at the last moment

9.1 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);
```

```
48
49        for (int w : bucket[i]) {
50          int v = find(w);
51          dom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
52        }
53
54        if (i > 0)
55          unite(par[i], i);
56      }
57
58      for (int i = 1; i < n; i++) {
59        if (dom[i] != sdom[i])
60          dom[i] = dom[dom[i]];
61        tree[order[i]].pb(order[dom[i]]);
62        tree[order[dom[i]]].pb(order[i]);
63      }
64
65      T = 0;
66      tin = tout = vi(n);
67      dfs_tree(root, -1);
68    }
69
70    void unite (int u, int v) {
71      dsu[v] = u;
72    }
73
74    int find (int u, int x = 0) {
75      if (u == dsu[u])
76        return (x ? -1 : u);
77      int v = find(dsu[u], x + 1);
78      if (v == -1)
79        return u;
80      if (sdom[label[dsu[u]]] < sdom[label[u]])
81        label[u] = label[dsu[u]];
82      dsu[u] = v;
83      return (x ? v : label[u]);
84    }
85
86    bool dominated_by (int v, int by_what) {
87      return tin[by_what] <= tin[v] && tout[v] <= tout[by_what];
88    }
89  };
```

9.2 Fast LCS

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

```
6  const ll bnd = (1LL << 32);
7
8  // WARNING: invokes undefined behaviour of modifying ans through pointer to
   ↪  another data type (uint)
9  // seems to work, but be wary
10 bs sum (const bs &bl, const bs &br)
11 {
12     const int steps = M / 32;
13     const uint* l = (uint*)&bl;
14     const uint* r = (uint*)&br;
15
16     bs ans;
17     uint* res = (uint*)&ans;
18
19     int carry = 0;
20     forn (i, steps)
21     {
22         ll cur = ll(*l++) + ll(*r++) + carry;
23         carry = (cur >= bnd);
24         cur = (cur >= bnd ? cur - bnd : cur);
25         *res++ = uint(cur);
26     }
27
28     return ans;
29 }
30
31 int fast_lcs (const string &s, const string &t)
32 {
33     const int m = sz(t);
34     const int let = 26;
35
36     vector<bs> has(let);
37     vector<bs> rev = has;
38
39     forn (i, m)
40     {
41         const int pos = t[i] - 'a';
42         has[pos].set(i);
43         forn (j, let) if (j != pos)
44             rev[j].set(i);
45     }
46
47     bs row;
48     forn (i, m)
49         row.set(i);
50
51     int cnt = 0;
52     for (char ch : s)
53     {
54         const int pos = ch - 'a';
```

```
55
56          bs next = sum(row, row & has[pos]) | (row & rev[pos]);
57          cnt += next[m];
58          next[m] = 0;
59
60          row = next;
61      }
62
63      return cnt;
64  }
```

9.3 Fast Subset Convolution

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

```
38
39      forn (cnt, bits + 1)
40      {
41          mobius(trans_a[cnt], bits, +1);
42          mobius(trans_b[cnt], bits, +1);
43      }
44
45      // Not really a valid ranked mobius transform! But algorithm works anyway
46
47      forn (i, bits + 1) forn (j, bits - i + 1) forn (mask, 1 << bits)
48          add(trans_res[i + j][mask], mult(trans_a[i][mask], trans_b[j][mask]));
49
50      forn (cnt, bits + 1)
51          mobius(trans_res[cnt], bits, -1);
52
53      forn (mask, 1 << bits)
54      {
55          const int cnt = __builtin_popcount(mask);
56          a[mask] = trans_res[cnt][mask];
57      }
58
59      return a;
60  }
```

10 Karatsuba

```
1   // 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
7   void add(hcvect abegin, hcvect aend, hvect ans)
8   {
9           for (auto it = abegin; it != aend; ++it, ++ans)
10                  *ans += *it;
11  }
12
13
14  void sub(hcvect abegin, hcvect aend, hvect ans)
15  {
16          for (auto it = abegin; it != aend; ++it, ++ans)
17                  *ans -= *it;
18  }
19
20
21  void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans)
22  {
23          for (int i = 0; i < siz; i++)
24                  for (int j = 0; j < siz; j++)
```

```
25                      *(ans + i + j) += *(abegin + i) * *(bbegin + j);
26  }
27
28
29  void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin, hvect ans, hvect small,
    ↪  hvect big, hvect sum)
30  {
31          assert((siz & (siz - 1)) == 0);
32
33          if (siz <= 32)
34          {
35                  stupid(siz, abegin, bbegin, ans);
36
37                  return;
38          }
39
40          auto amid = abegin + siz / 2, aend = abegin + siz;
41          auto bmid = bbegin + siz / 2, bend = bbegin + siz;
42          auto smid = sum + siz / 2, send = sum + siz;
43
44          fill(small, small + siz, 0);
45          Karatsuba(siz / 2, abegin, bbegin, small, small + siz, big + siz, sum);
46          fill(big, big + siz, 0);
47          Karatsuba(siz / 2, amid, bmid, big, small + siz, big + siz, sum);
48
49          copy(abegin, amid, sum);
50          add(amid, aend, sum);
51          copy(bbegin, bmid, sum + siz / 2);
52          add(bmid, bend, sum + siz / 2);
53
54          Karatsuba(siz / 2, sum, smid, ans + siz / 2, small + siz, big + siz,
            ↪  send);
55
56          add(small, small + siz, ans);
57          sub(small, small + siz, ans + siz / 2);
58          add(big, big + siz, ans + siz);
59          sub(big, big + siz, ans + siz / 2);
60  }
61
62
63  void mult(vector<modulo<>> a, vector<modulo<>> b, vector<modulo<>> &c)
64  {
65          a.resize(up(max(a.size(), b.size())), 0);
66          b.resize(a.size(), 0);
67
68          c.resize(max(c.size(), a.size() * 2), 0);
69
70          vector<modulo<>> small(2 * a.size());
71          auto big = small;
72          auto sum = small;
```

```
73
74          Karatsuba(a.size(), a.begin(), b.begin(), c.begin(), small.begin(),
            ↪  big.begin(), sum.begin());
75  }
```

11 Hard Algorithms

11.1 Two Strong Chinese

```
1   template<class T, class Add>
2   class skew_heap
3   {
4           struct node
5           {
6                   using nodeptr = unique_ptr<node>;
7
8                   nodeptr l = nullptr, r = nullptr;
9                   T x;
10
11                  explicit node(T x = {}) : x(x)
12                  {}
13          };
14
15          using nodeptr = typename node::nodeptr;
16
17          static nodeptr merge(nodeptr &&a, nodeptr &&b)
18          {
19                  if (a == nullptr)
20                          return std::move(b);
21                  if (b == nullptr)
22                          return std::move(a);
23                  if (b->x < a->x)
24                          return merge(std::move(b), std::move(a));
25
26                  auto tmp = merge(std::move(a->r), std::move(b));
27                  a->r = std::move(a->l);
28                  a->l = std::move(tmp);
29
30                  return std::move(a);
31          }
32
33          void add_to_all(nodeptr &a, Add x)
34          {
35                  if (a == nullptr)
36                          return;
37
38                  a->x += x;
39                  add_to_all(a->l, x);
40                  add_to_all(a->r, x);
41          }
```

```
42
43          nodeptr root = nullptr;
44          size_t siz = 0;
45          Add to_add{};
46
47  public:
48          void add(Add x)
49          {
50                  to_add += x;
51          }
52
53          [[nodiscard]] T top() const
54          {
55                  return root->x + to_add;
56          }
57
58          [[nodiscard]] auto size() const
59          {
60                  return siz;
61          }
62
63          [[nodiscard]] auto empty() const
64          {
65                  return size() == 0;
66          }
67
68          void pop()
69          {
70                  auto q = merge(std::move(root->l), std::move(root->r));
71                  siz--;
72                  root = std::move(q);
73          }
74
75          void merge(skew_heap &&rhs)
76          {
77                  if (size() < rhs.size())
78                          swap(*this, rhs);
79
80                  siz += rhs.siz;
81                  rhs.siz = 0;
82                  rhs.add_to_all(rhs.root, rhs.to_add - to_add);
83                  auto q = merge(std::move(root), std::move(rhs.root));
84                  root = std::move(q);
85          }
86
87          void push(T x)
88          {
89                  skew_heap sh;
90                  sh.root = make_unique<node>(x);
91                  sh.siz = 1;
```

```
92
93                      merge(std::move(sh));
94              }
95      };
96
97      struct edge
98      {
99              ll w;
100             int to;
101             int id;
102
103             strong_ordering operator<=>(const edge &rhs) const
104             {
105                     return w <=> rhs.w;
106             }
107
108             edge &operator+=(ll rhs)
109             {
110                     w += rhs;
111
112                     return *this;
113             }
114
115             edge operator+(ll rhs) const
116             {
117                     return edge{w + rhs, to, id};
118             }
119     };
120
121     enum color_t
122     {
123             White = 0, Grey, Black, Cycle
124     };
125
126     vector<int> solve(size_t n, const vector<tuple<int, int, int>> &edges, int root =
        ↪  0)
127     {
128             vector<skew_heap<edge, ll>> rev(n);
129
130             for (int i = 0; i < (int) edges.size(); i++)
131             {
132                     auto [a, b, w] = edges[i];
133
134                     if (b != root)
135                             rev[b].push(edge{w, a, i});
136             }
137
138             auto mrg = [&](int a, int b)
139             {
140                     rev[a].merge(std::move(rev[b]));
```

```
141         };
142
143         dsu cc(n, mrg);
144
145         vector<color_t> color(rev.size());
146         color[root] = Black;
147
148         vector<int> ids;
149
150         function<bool(int)> dfs = [&](int v) → bool
151         {
152                 v = cc.get_class(v);
153
154                 if (color[v] == Black)
155                         return false;
156
157                 if (color[v] == Grey)
158                 {
159                         color[v] = Cycle;
160
161                         return true;
162                 }
163                 color[v] = Grey;
164
165                 while (true)
166                 {
167                         while (!rev[v].empty() && cc.get_class(rev[v].top().to) ==
                        ↪  v)
168                                 rev[v].pop();
169
170                         assert(!rev[v].empty()); // assume that the answer exists
171                         auto [w, to, id] = rev[v].top();
172
173                         ids.emplace_back(id); // ans += w; if the certificate is
                        ↪  not needed
174
175                         rev[v].add(-w);
176
177                         if (dfs(to))
178                         {
179                                 if (color[v] ≠ Cycle)
180                                 {
181                                         cc.unite(v, to);
182                                         color[cc.get_class(v)] = Cycle;
183
184                                         return true;
185                                 }
186                                 else
187                                 {
188                                         v = cc.get_class(v);
```

```
189
190                                     color[v] = Grey;
191                             }
192                     }
193                     else
194                     {
195                             color[v] = Black;
196
197                             return false;
198                     }
199             }
200     };
201
202     for (int i = 0; i < (int) rev.size(); i++)
203             dfs(i);
204
205     // finding answer, similar to Prim
206     vector<vector<int>> gr(n);
207
208     for (int i = 0; i < int(ids.size()); i++)
209     {
210             auto [a, b, _] = edges[ids[i]];
211
212             gr[a].push_back(i);
213     }
214
215     minheap<int> pq(gr[root].begin(), gr[root].end());
216     vector<bool> used(n);
217     used[root] = true;
218
219     vector<int> ans;
220
221     while (!pq.empty())
222     {
223             auto i = pq.top();
224             pq.pop();
225             auto v = get<1>(edges[ids[i]]);
226
227             if (used[v])
228                     continue;
229             used[v] = true;
230
231             ans.push_back(ids[i]);
232
233             for (auto it: gr[v])
234                     pq.push(it);
235     }
236
237     return ans;
238 }
```

```
239
240
241 void dfs(const vector<vector<pair<int, int>>> &gr, vector<bool> &used, int v)
242 {
243         if (used[v])
244                 return;
245         used[v] = true;
246
247         for (auto [u, w]: gr[v])
248                 dfs(gr, used, u);
249 }
250
251
252 void solve(istream &cin = std::cin, ostream &cout = std::cout)
253 {
254         int n, m;
255
256         cin >> n >> m;
257
258         vector<tuple<int, int, int>> edges(m);
259         vector<vector<pair<int, int>>> gr(n);
260
261         for (int i = 0; i < m; i++)
262         {
263                 auto &[a, b, w] = edges[i];
264
265                 cin >> a >> b >> w;
266                 a--;
267                 b--;
268
269                 gr[a].emplace_back(b, w);
270         }
271
272         vector<bool> used(gr.size());
273
274         dfs(gr, used, 0);
275
276         if (ranges::count(used, false))
277         {
278                 cout << "NO" << endl;
279
280                 return;
281         }
282
283         cout << "YES" << endl;
284
285         auto ids = solve(gr.size(), edges);
286
287         ll ans = 0;
288
```

```
289         for (auto it: ids)
290                 ans += get<2>(edges[it]);
291
292         for (auto &row: gr)
293                 row.clear();
294
295         for (auto it: ids)
296         {
297                 auto [a, b, w] = edges[it];
298
299                 gr[a].emplace_back(b, w);
300         }
301
302         used.assign(used.size(), false);
303
304         dfs(gr, used, 0);
305
306         assert(ranges::count(used, false) == 0);
307
308         cout << ans << endl;
309 }
```

11.2 Simplex

```
1  mt19937 mt(736);
2
3  using ld = double;
4  constexpr ld eps = 1e-9;
5
6  bool eps_nonneg(ld x)
7  {
8          return x >= -eps;
9  }
10
11 bool eps_zero(ld x)
12 {
13         return abs(x) <= eps;
14 }
15
16 bool cmp_abs(ld a, ld b)
17 {
18         return abs(a) < abs(b);
19 }
20
21 vector<ld> &add_prod(vector<ld> &lhs, const vector<ld> &rhs, ld x)
22 {
23         assert(ssize(lhs) == ssize(rhs));
24
25         for (auto i: ranges::iota_view(0, ssize(lhs)))
26                 lhs[i] += rhs[i] * x;
```

```
27
28          return lhs;
29  }
30
31  vector<ld> &operator/=(vector<ld> &lhs, ld x)
32  {
33          for (auto &it: lhs)
34                  it /= x;
35
36          return lhs;
37  }
38
39  void basis_change(vector<ld> &row, const vector<ld> &nd, int b)
40  {
41          auto mult = row[b];
42
43          add_prod(row, nd, mult);
44
45          row[b] = 0;
46  }
47
48  void pivot(vector<vector<ld>> &a, vector<int> &b, vector<ld> &func, int wh, int x)
49  {
50          a[wh][b[wh]] = -1;
51          b[wh] = x;
52          auto den = -a[wh][x];
53          a[wh][x] = 0;
54          a[wh] /= den;
55
56          for (auto i: ranges::iota_view(0, ssize(a)))
57                  if (i != wh)
58                          basis_change(a[i], a[wh], b[wh]);
59          basis_change(func, a[wh], b[wh]);
60  }
61
62  bool simplex(vector<vector<ld>> &a, vector<int> &b, vector<ld> &func)
63  {
64          while (true)
65          {
66                  vector<int> cand;
67
68                  for (auto i: ranges::iota_view(0, ssize(func) - 1))
69                          if (func[i] > eps)
70                                  cand.push_back(i);
71
72                  if (cand.empty())
73                          return true;
74
75                  auto x = cand[uniform_int_distribution<int>{0, (int) cand.size() -
                    ↪  1}(mt)];
```

```
76
77                 vector<ld> len(a.size(), numeric_limits<ld>::max());
78
79                 for (auto i: ranges::iota_view(0, ssize(len)))
80                         if (a[i][x] < -eps)
81                                 len[i] = a[i].back() / -a[i][x];
82
83                 auto wh = int(ranges::min_element(len) - len.begin());
84
85                 if (len[wh] == numeric_limits<ld>::max())
86                         return false;
87
88                 pivot(a, b, func, wh, x);
89         }
90 }
91
92 enum results
93 {
94         NO_SOLUTION, UNBOUNDED, BOUNDED
95 };
96
97 /*
98  * Solving system of linear inequalities in the form
99  * $a * x ⩽ rhs$
100  * $x ⩾ 0$
101  * $costs * x → max$
102  * assumes at least one inequality and at least one variable
103  * */
104 results global_solve(vector<vector<ld>> a, const vector<ld> &rhs, const vector<ld>
    ↪  &costs, vector<ld> &ans)
105 {
106         assert(!a.empty() && a.size() == rhs.size() && !costs.empty() &&
        ↪  ans.size() == costs.size());
107         const auto m = costs.size() + a.size() + 2;
108
109         for (auto i: ranges::iota_view(0, ssize(a)))
110         {
111                 auto &row = a[i];
112
113                 row ⩴ -1; // just finding inverse
114                 row.resize(m);
115                 row.back() = rhs[i];
116                 row.rbegin()[1] = 1;
117         }
118
119         vector<ld> func(m), lambda(m);
120         vector<int> b(a.size());
121
122         iota(b.begin(), b.end(), (int) costs.size());
123
```

```
124         lambda.rbegin()[1] = -1;
125         for (auto j: ranges::iota_view(0, ssize(costs)))
126                 func[j] = costs[j];
127
128         auto wh = int(ranges::min_element(rhs) - rhs.begin());
129
130         if (rhs[wh] < 0)
131         {
132                 pivot(a, b, lambda, wh, (int) lambda.size() - 2);
133
134                 auto q = simplex(a, b, lambda);
135
136                 assert(q);
137         }
138
139         wh = int(ranges::find(b, (int) lambda.size() - 2) - b.begin());
140
141         if (!eps_zero(lambda.back()))
142                 return NO_SOLUTION;
143
144         if (wh != size(b))
145         {
146                 if (!eps_zero(a[wh].back()))
147                         return NO_SOLUTION;
148
149                 auto q = int(ranges::find_if(a[wh], eps_nonneg) - a[wh].begin());
150
151                 if (q != ssize(a[wh]))
152                 {
153                         pivot(a, b, lambda, wh, q);
154                 }
155                 else
156                 {
157                         q = int(ranges::max_element(a[wh], cmp_abs) -
                        ↪  a[wh].begin());
158
159                         if (!eps_zero(a[wh][q]))
160                                 pivot(a, b, lambda, wh, q);
161                 }
162         }
163
164         for (auto &row: a)
165                 row.rbegin()[1] = 0;
166
167         for (auto i: ranges::iota_view(0, ssize(b)))
168                 basis_change(func, a[i], b[i]);
169
170         if (!simplex(a, b, func))
171                 return UNBOUNDED;
172
```

```
173         for (auto i: ranges::iota_view(0, ssize(a)))
174                 if (b[i] < ssize(ans))
175                         ans[b[i]] = a[i].back();
176 
177         return BOUNDED;
178 }
```

12 OEIS

12.1 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

12.2 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