

Содержание

1 Setup & Scripts	1
1.1 CMake	1
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 Bugs	3
4 Geometry	4
4.1 Пересечение прямых	4
4.2 Касательные	4
4.3 Пересечение полуплоскостей	4
5 Template dsu	4
6 Numbers	5
7 Graphs	6
7.1 Weighted matroid intersection	6
8 Push-free segment tree	9
9 Number theory	10
9.1 Chinese remainder theorem without overflows	10
9.2 Integer points under a rational line	11
10 Suffix Automaton	11
11 Palindromic Tree	13
12 Smth added at last moment	14
12.1 Dominator Tree	14
12.2 Suffix Array	16
12.3 Fast LCS	18
12.4 Fast Subset Convolution	19
13 Karatsuba	21

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 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);
        }
    };
}
```

2.2 Python

```
# stack size
import sys

sys.setrecursionlimit(10**6)

# memoize
import functools

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

3 Bugs

- `powmod :`)
- Всегда чекать Куна дважды, особенно на количество итераций
- `uniform_int_distribution` от одного параметра
- `for (char c : "NEWS")`
- Порядок верхних и нижних границ в случае, когда задача двумерна $t - b \neq b - t$
- `static` с мультитестами
- `set` со своим компаратором склеивает элементы
- Два вектора с соответствующими элементами, сортируем один, а элементы второго ссылаются на чужие. Предлагается лечить заведением структуры с компаратором на каждый чих. В целом, для этого можно написать навороченную хрень на шаблонах.

- В графе с вершинами степени не больше одного надо писать выделение цикла полностью, срезать угол на какой-нибудь тупой меморизации, потому что кажется, что он может выглядеть только одним или несколькими какими-нибудь специальными способами, не получится, а дебажить сложно.
- Структуры, основанные на указателях, не стоит хранить в векторах.
- В Карасе для того, чтобы перейти в подстроку, надо сначала идти в родителя, а только потом по суфф. ссылкам, эти вещи не коммутируют.
- Когда ходим большим количеством указателей по циклу, на единицу сдвигается только первый указатель, а остальные могут сдвинуться на много.
- `string str1(str2, 'x'); str1 = 'a';`

4 Geometry

4.1 Пересечение прямых

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

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

4.2 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 $\frac{r^2}{x}$.

4.3 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.

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

7 Graphs

7.1 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);
53         assert(p != new_ids.end());
54         new_ids.erase(p);
55
56         can_extend cur(new_ids, n, es);
57
58         for (int i = 0; i < int(es.size()); i++)
59             if (!in[i])
60             {
61                 if (cur.extend_red(i, es))
62                     g[i].push_back(j);
63                 if (cur.extend_blue(i, es))
64                     g[j].push_back(i);
65             }
66     }
67
68 auto get_cost = [&] (int x)
69 {
70     const int cost = (!in[x] ? e[x].w : -e[x].w);
71     return (ver + 1) * cost - 1;
72 };
73
74 const int inf = int(1e9);
75 vector<int> dist(ver, -inf), prev(ver, -1);
76 for (int x : from)
77     dist[x] = get_cost(x);
78
79 queue<int> q;
80
81 vector<int> used(ver);
82 for (int x : from)
83 {
84     q.push(x);
85     used[x] = 1;
86 }
87
```

```
88     while (!q.empty())
89     {
90         int cur = q.front(); used[cur] = 0; q.pop();
91
92         for (int to : g[cur])
93         {
94             int cost = get_cost(to);
95             if (dist[to] < dist[cur] + cost)
96             {
97                 dist[to] = dist[cur] + cost;
98                 prev[to] = cur;
99                 if (!used[to])
100                 {
101                     used[to] = 1;
102                     q.push(to);
103                 }
104             }
105         }
106     }
107
108     int best = -inf, where = -1;
109     for (int x : to)
110     {
111         if (dist[x] > best)
112         {
113             best = dist[x];
114             where = x;
115         }
116     }
117
118     if (best == -inf)
119         return pair<T, int>(cur_set, best);
120
121     while (where != -1)
122     {
123         cur_set ^= (T(1) << where);
124         where = prev[where];
125     }
126
127     while (best % (ver + 1))
128         best++;
129     best /= (ver + 1);
130
131     assert(oracle(cur_set, red) && oracle(cur_set, blue));
132     return pair<T, int>(cur_set, best);
133 };
```


8 Push-free segment tree

```

1  class pushfreesegtree
2  {
3      vector<modulo<>> pushed, unpushed;
4
5      modulo<> add(int l, int r, int cl, int cr, int v, const modulo<> &x)
6      {
7          if (r ≤ cl || cr ≤ l)
8              return 0;
9          if (l ≤ cl && cr ≤ r)
10         {
11             unpushed[v] += x;
12
13             return x * (cr - cl);
14         }
15
16         int ct = (cl + cr) / 2;
17
18         auto tmp = add(l, r, cl, ct, 2 * v, x) + add(l, r, ct, cr, 2 * v +
19             ↪ 1, x);
20
21         pushed[v] += tmp;
22
23         return tmp;
24     }
25
26     modulo<> sum(int l, int r, int cl, int cr, int v)
27     {
28         if (r ≤ cl || cr ≤ l)
29             return 0;
30         if (l ≤ cl && cr ≤ r)
31             return pushed[v] + unpushed[v] * (cr - cl);
32
33         int ct = (cl + cr) / 2;
34
35         return sum(l, r, cl, ct, 2 * v) + unpushed[v] * (min(r, cr) -
36             ↪ max(l, cl)) + sum(l, r, ct, cr, 2 * v + 1);
37     }
38 public:
39     pushfreesegtree(int n) : pushed(2 * up(n)), unpushed(2 * up(n))
40     {}
41
42
43     modulo<> sum(int l, int r)
44     {
45         return sum(l, r, 0, pushed.size() / 2, 1);
46     }

```

```

47
48
49     void add(int l, int r, const modulo &x)
50     {
51         add(l, r, 0, pushed.size() / 2, 1, x);
52     }
53 };

```

9 Number theory

9.1 Chinese remainder theorem without overflows

```

1  // Replace T with an appropriate type!
2  using T = long long;
3
4  // Finds x, y such that ax + by = gcd(a, b).
5  T gcdext (T a, T b, T &x, T &y)
6  {
7      if (b == 0)
8      {
9          x = 1, y = 0;
10         return a;
11     }
12
13     T res = gcdext (b, a % b, y, x);
14     y -= x * (a / b);
15     return res;
16 }
17
18 // Returns true if system x = r1 (mod m1), x = r2 (mod m2) has solutions
19 // false otherwise. In first case we know exactly that x = r (mod m)
20
21 bool crt (T r1, T m1, T r2, T m2, T &r, T &m)
22 {
23     if (m2 > m1)
24     {
25         swap(r1, r2);
26         swap(m1, m2);
27     }
28
29     T g = __gcd(m1, m2);
30     if ((r2 - r1) % g != 0)
31         return false;
32
33     T c1, c2;
34     auto nrem = gcdext(m1 / g, m2 / g, c1, c2);
35     assert(nrem == 1);
36     assert(c1 * (m1 / g) + c2 * (m2 / g) == 1);
37     T a = c1;
38     a *= (r2 - r1) / g;

```

```

39     a %= (m2 / g);
40     m = m1 / g * m2;
41     r = a * m1 + r1;
42     r = r % m;
43     if (r < 0)
44         r += m;
45
46     assert(r % m1 == r1 && r % m2 == r2);
47     return true;
48 }

```

9.2 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  {
6      if (k == 0)
7          return (b / d) * n;
8      if (k ≥ d || b ≥ d)
9          return (k / d) * n * (n - 1) / 2 + (b / d) * n + trapezoid(n, k % d, b %
10         ↪ d, d);
11     return trapezoid((k * n + b) / d, d, (k * n + b) % d, k);

```

10 Suffix Automaton

```

1  struct tomato
2  {
3      vector<map<char, int>> edges;
4      vector<int> link, length;
5      int last;
6      /// Restoring terminal states, optional, but usually needed.
7      vector<int> terminals;
8      vector<bool> is_terminal;
9      /// Optional, makes dp easier. Alternative: use dfs.
10     vector<int> order, rev_order, next_in_order;
11
12     explicit tomato(const string &s) : last(0)
13     {
14         add_vertex(map<char, int>(), 0, -1);
15         for (const char ch : s)
16             extend(ch);
17
18         int cur = last;
19         is_terminal.assign(edges.size(), false);
20         /// Assuming empty suffix should be accepted, otherwise use "while
21         ↪ (cur > 0)".
22         while (cur ≥ 0)

```

```
22         {
23             terminals.push_back(cur);
24             is_terminal[cur] = true;
25             cur = link[cur];
26         }
27
28         /// Restoring topsort and reverse topsort, optional.
29         order.push_back(0);
30         while (order.back()  $\neq$  -1)
31             order.push_back(next_in_order[order.back()]);
32         order.pop_back();
33         rev_order = order;
34         reverse(rev_order.begin(), rev_order.end());
35     }
36
37     int add_vertex(const map<char, int> &temp, const int len, const int lnk)
38     {
39         edges.emplace_back(temp);
40         length.emplace_back(len);
41         link.emplace_back(lnk);
42         next_in_order.push_back(-1);
43         return int(edges.size()) - 1;
44     }
45
46     void extend(const char ch)
47     {
48         const int new_last = add_vertex(map<char, int>(), length[last] +
49              $\hookrightarrow$  1, 0);
50         assert(next_in_order[last] == -1);
51         next_in_order[last] = new_last;
52
53         int p = last;
54         while (p  $\geq$  0 && !edges[p].count(ch))
55         {
56             edges[p][ch] = new_last;
57             p = link[p];
58         }
59
60         if (p  $\neq$  -1)
61         {
62             const int q = edges[p][ch];
63             if (length[p] + 1 == length[q])
64                 link[new_last] = q;
65             else
66             {
67                 const int clone = add_vertex(edges[q], length[p] +
68                      $\hookrightarrow$  1, link[q]);
69                 next_in_order[clone] = next_in_order[q];
70                 next_in_order[q] = clone;
```

```

70         link[q] = clone;
71         link[new_last] = clone;
72
73         while (p ≥ 0 && edges[p][ch] = q)
74         {
75             edges[p][ch] = clone;
76             p = link[p];
77         }
78     }
79 }
80
81     last = new_last;
82 }
83 };

```

11 Palindromic Tree

```

1  class treert
2  {
3      struct node
4      {
5          array<int, 26> nxt;
6          int par, link, siz;
7
8          node(int siz, int par, int link) : par(par), link(link = -1 ? 1 :
          ↪ link), siz(siz)
9          {
10             fill(nxt.begin(), nxt.end(), -1);
11         }
12     };
13
14     vector<node> mem;
15     vector<int> suff; // longest palindromic suffix
16
17 public:
18     treert(const string &str) : suff(str.size())
19     {
20         mem.emplace_back(-1, -1, 0);
21         mem.emplace_back(0, 0, 0);
22         mem[0].link = mem[1].link = 0;
23
24         auto link_walk = [&](int st, int pos)
25         {
26             while (pos - 1 - mem[st].siz < 0 || str[pos] ≠ str[pos -
              ↪ 1 - mem[st].siz])
27                 st = mem[st].link;
28
29             return st;
30         };
31

```

```

32         for (int i = 0, last = 1; i < str.size(); i++)
33         {
34             last = link_walk(last, i);
35             auto ind = str[i] - 'a';
36
37             if (mem[last].nxt[ind] == -1)
38             {
39                 // order is important
40                 mem.emplace_back(mem[last].siz + 2, last,
41                               ↪ mem[link_walk(mem[last].link, i)].nxt[ind]);
42                 mem[last].nxt[ind] = (int)mem.size() - 1;
43             }
44
45             last = mem[last].nxt[ind];
46
47             suff[i] = last;
48         }
49     };

```

12 Smth added at last moment

12.1 Dominator Tree

```

1  struct dom_tree {
2      vvi g, rg, tree, bucket;
3      vi sdom, par, dom, dsu, label, in, order, tin, tout;
4      int T = 0, root = 0, n = 0;
5
6      void dfs_tm (int x) {
7          in[x] = T;
8          order[T] = x;
9          label[T] = T, sdom[T] = T, dsu[T] = T, dom[T] = T;
10         T++;
11         for (int to : g[x]) {
12             if (in[to] == -1) {
13                 dfs_tm(to);
14                 par[in[to]] = in[x];
15             }
16             rg[in[to]].pb(in[x]);
17         }
18     }
19
20     void dfs_tree (int v, int p) {
21         tin[v] = T++;
22         for (int dest : tree[v]) {
23             if (dest != p) {
24                 dfs_tree(dest, v);
25             }
26         }

```

```

27     tout[v] = T;
28 }
29
30 dom_tree (const vvi &g_, int root_) {
31     g = g_;
32     n = sz(g);
33     assert(0 ≤ root && root < n);
34     in.assign(n, -1);
35     rg.resize(n);
36     order = sdom = par = dom = dsu = label = vi(n);
37     root = root_;
38     bucket.resize(n);
39     tree.resize(n);
40
41     dfs_tm(root);
42
43     for (int i = n - 1; i ≥ 0; i--) {
44         for (int j : rg[i])
45             sdom[i] = min(sdom[i], sdom[find(j)]);
46         if (i > 0)
47             bucket[sdom[i]].pb(i);
48
49         for (int w : bucket[i]) {
50             int v = find(w);
51             dom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
52         }
53
54         if (i > 0)
55             unite(par[i], i);
56     }
57
58     for (int i = 1; i < n; i++) {
59         if (dom[i] ≠ sdom[i])
60             dom[i] = dom[dom[i]];
61         tree[order[i]].pb(order[dom[i]]);
62         tree[order[dom[i]]].pb(order[i]);
63     }
64
65     T = 0;
66     tin = tout = vi(n);
67     dfs_tree(root, -1);
68 }
69
70 void unite (int u, int v) {
71     dsu[v] = u;
72 }
73
74 int find (int u, int x = 0) {
75     if (u == dsu[u])
76         return (x ? -1 : u);

```

```

77     int v = find(dsu[u], x + 1);
78     if (v == -1)
79         return u;
80     if (sdom[label[dsu[u]]] < sdom[label[u]])
81         label[u] = label[dsu[u]];
82     dsu[u] = v;
83     return (x ? v : label[u]);
84 }
85
86 bool dominated_by (int v, int by_what) {
87     return tin[by_what] ≤ tin[v] && tout[v] ≤ tout[by_what];
88 }
89 };

```

12.2 Suffix Array

```

1  namespace suff_arr {
2
3  const int MAXN = 2e5 + 10;
4
5  string s;
6  int n;
7  int p[MAXN];
8  int lcp[MAXN];
9  int pos[MAXN];
10 int c[MAXN];
11
12 void print() {
13     #ifndef LOCAL
14         return;
15     #endif
16     eprintf("p:\n");
17     forn(i, sz(s)) {
18         eprintf("i=%d -- %d: %s, lcp=%d, c=%d\n", i, p[i], s.substr(p[i], sz(s) -
19             ↪ p[i]).data(), lcp[i], c[p[i]]);
20     }
21     eprintf("\n");
22 }
23
24 void build(const string& s_) {
25     static int cnt[MAXN];
26     static int np[MAXN];
27     static int nc[MAXN];
28
29     s = s_;
30     n = sz(s);
31
32     memset (cnt, 0, sizeof cnt);
33     for (char ch : s) {
34         ++cnt[int(ch)];
35     }
36 }

```



```
34     }
35     forn(i, 256) {
36         cnt[i + 1] += cnt[i];
37     }
38     forn(i, sz(s)) {
39         p[--cnt[int(s[i])]] = i;
40     }
41
42     int cls = 1;
43     c[p[0]] = cls - 1;
44     for (int i = 1; i < n; ++i) {
45         if (s[p[i]] != s[p[i - 1]]) {
46             ++cls;
47         }
48         c[p[i]] = cls - 1;
49     }
50
51     for (int len = 1; len ≤ n; len *= 2) {
52         memset (cnt, 0, sizeof(int) * cls);
53         forn(i, n) {
54             ++cnt[c[i]];
55         }
56         forn(i, cls - 1) {
57             cnt[i + 1] += cnt[i];
58         }
59         ford(i, n) {
60             const int j = p[i];
61             int j2 = (j - len + n) % n;
62             np[--cnt[c[j2]]] = j2;
63         }
64         memcpy(p, np, sizeof(int) * n);
65
66         cls = 1;
67         nc[p[0]] = cls - 1;
68         for (int i = 1; i < n; ++i) {
69             if (c[p[i]] != c[p[i - 1]] || c[(p[i] + len) % n] != c[(p[i - 1] +
70                 ↪ len) % n]) {
71                 ++cls;
72             }
73             nc[p[i]] = cls - 1;
74         }
75         memcpy(c, nc, sizeof(int) * n);
76     }
77     forn(i, n) {
78         pos[p[i]] = i;
79     }
80
81     int pref = 0;
82     forn(i, n) {
```

```

83     int pi = pos[i];
84     if (pi == n - 1) {
85         continue;
86     }
87     int j = p[pi + 1];
88     while (i + pref < n && j + pref < n && s[i + pref] == s[j + pref]) {
89         ++pref;
90     }
91     lcp[pi] = pref;
92     pref = max(0, pref - 1);
93 }
94
95 //     print();
96 }
97
98 };

```

12.3 Fast LCS

```

1 // assumes that strings consist of lowercase latin letters
2 const int M = ((int)1e5 + 64) / 32 * 32;
3 // maximum value of m
4 using bs = bitset<M>;
5 using uint = unsigned int;
6 const ll bnd = (1LL << 32);
7
8 // WARNING: invokes undefined behaviour of modifying ans through pointer to
9 // ↪ another data type (uint)
10 // seems to work, but be wary
11 bs sum (const bs &bl, const bs &br)
12 {
13     const int steps = M / 32;
14     const uint* l = (uint*)&bl;
15     const uint* r = (uint*)&br;
16
17     bs ans;
18     uint* res = (uint*)&ans;
19
20     int carry = 0;
21     forn (i, steps)
22     {
23         ll cur = ll(*l++) + ll(*r++) + carry;
24         carry = (cur ≥ bnd);
25         cur = (cur ≥ bnd ? cur - bnd : cur);
26         *res++ = uint(cur);
27     }
28
29     return ans;
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 }

```

12.4 Fast Subset Convolution

```

1 // algorithm itself starts here
2 void mobius (int* a, int n, int sign)
3 {
4     forn (i, n)
5     {
6         int free = ((1 << n) - 1) ^ (1 << i);
7         for (int mask = free; mask > 0; mask = ((mask - 1) & free))
8             (sign == +1 ? add : sub)(a[mask ^ (1 << i)], a[mask]);
9         add(a[1 << i], a[0]);
10    }
11 }
12
13 // maximum number of bits allowed

```

```
14  const int B = 20;
15
16  vi fast_conv (vi a, vi b)
17  {
18      assert(!a.empty());
19      const int bits = __builtin_ctz(sz(a));
20      assert(sz(a) == (1 << bits) && sz(a) == sz(b));
21
22      static int trans_a[B + 1][1 << B];
23      static int trans_b[B + 1][1 << B];
24      static int trans_res[B + 1][1 << B];
25
26      forn (cnt, bits + 1)
27      {
28          for (auto cur : {trans_a, trans_b, trans_res})
29              fill(cur[cnt], cur[cnt] + (1 << bits), 0);
30      }
31
32      forn (mask, 1 << bits)
33      {
34          const int cnt = __builtin_popcount(mask);
35          trans_a[cnt][mask] = a[mask];
36          trans_b[cnt][mask] = b[mask];
37      }
38
39      forn (cnt, bits + 1)
40      {
41          mobius(trans_a[cnt], bits, +1);
42          mobius(trans_b[cnt], bits, +1);
43      }
44
45      // Not really a valid ranked mobius transform! But algorithm works anyway
46
47      forn (i, bits + 1) forn (j, bits - i + 1) forn (mask, 1 << bits)
48          add(trans_res[i + j][mask], mult(trans_a[i][mask], trans_b[j][mask]));
49
50      forn (cnt, bits + 1)
51          mobius(trans_res[cnt], bits, -1);
52
53      forn (mask, 1 << bits)
54      {
55          const int cnt = __builtin_popcount(mask);
56          a[mask] = trans_res[cnt][mask];
57      }
58
59      return a;
60 }
```

13 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(hvect abegin, hcvect aend, hvect ans)
8 {
9     for (auto it = abegin; it  $\neq$  aend; ++it, ++ans)
10         *ans += *it;
11 }
12
13
14 void sub(hvect abegin, hcvect aend, hvect ans)
15 {
16     for (auto it = abegin; it  $\neq$  aend; ++it, ++ans)
17         *ans -= *it;
18 }
19
20
21 void stupid(int siz, hcvect abegin, hcvect bbegin, hvect ans)
22 {
23     for (auto a = abegin; a  $\neq$  abegin + siz; ++a, ans -= (siz - 1))
24         for (auto b = bbegin; b  $\neq$  bbegin + siz; ++b, ++ans)
25             *ans += *a * *b;
26 }
27
28
29 void Karatsuba(size_t siz, hcvect abegin, hcvect bbegin, hvect ans, hvect small,
30               ↪ hvect big, hvect sum)
31 {
32     assert((siz & (siz - 1)) == 0);
33
34     if (siz  $\leq$  32)
35     {
36         stupid(siz, abegin, bbegin, ans);
37
38         return;
39     }
40
41     auto amid = abegin + siz / 2, aend = abegin + siz;
42     auto bmid = bbegin + siz / 2, bend = bbegin + siz;
43     auto smid = sum + siz / 2, send = sum + siz;
44
45     fill(small, small + siz, 0);
46     Karatsuba(siz / 2, abegin, bbegin, small, small + siz, big + siz, sum);
47     fill(big, big + siz, 0);
48     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,
55         ↪ send);
56
57     add(small, small + siz, ans);
58     sub(small, small + siz, ans + siz / 2);
59     add(big, big + siz, ans + siz);
60     sub(big, big + siz, ans + siz / 2);
61 }
62
63 void mult(vector<modulo<>> a, vector<modulo<>> b, vector<modulo<>> &c)
64 {
65     a.resize(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(),
75         ↪ big.begin(), sum.begin());
76 }
```



