
Contents

1	Misc	3	2.3 Heavy-Light Decomposition	10
1.1	Contest	3		
1.1.1	Makefile	3		
1.2	How Did We Get Here?	3		
1.2.1	Macros	3		
1.2.2	Fast I/O	3		
1.2.3	constexpr	5		
1.2.4	Bump Allocator	5		
1.3	Tools	6		
1.3.1	Floating Point Binary Search	6		
1.3.2	SplitMix64	6		
1.3.3	<random>	6		
1.3.4	x86 Stack Hack	6		
1.4	Algorithms	7		
1.4.1	Bit Hacks	7		
1.4.2	Infinite Grid Knight Distance	7		
1.4.3	Longest Increasing Subsequence	7		
1.4.4	Mo's Algorithm on Tree	7		
2	Data Structures	9		
2.1	GNU PBDS	9		
2.2	2D Partial Sums	9		
3	Graph	12		
3.1	Matching/Flows	12		
3.1.1	Kuhn-Munkres algorithm	12		
3.2	Shortest Path Faster Algorithm	13		
3.3	Strongly Connected Components	14		
3.4	Biconnected Components	14		
3.4.1	Articulation Points	14		
3.4.2	Bridges	15		
3.5	Centroid Decomposition	15		
4	Math	17		
4.1	Number Theory	17		
4.1.1	Mod Struct	17		
4.1.2	Miller-Rabin	17		
4.1.3	Linear Sieve	18		
4.1.4	Get Factors	18		
4.1.5	Binary GCD	19		
4.1.6	Extended GCD	19		
4.1.7	Chinese Remainder Theorem	19		
4.1.8	Pollard's Rho	19		
4.1.9	Rational Number Binary Search	20		

4.1.10	De Bruijn Sequence	20	6.5	Aho-Corasick Automaton	32
4.1.11	Long Long Multiplication	21		7 Debug List	34
5	Geometry	22			
5.1	Point	22			
5.1.1	Quaternion	22			
5.1.2	Spherical Coordinates	23			
5.2	Segments	24			
5.3	Convex Hull	24	5.3.1	3D Hull	24
5.4	Angular Sort	25			
5.5	Convex Polygon Minkowski Sum	26			
5.6	Point In Polygon	26	5.6.1	Convex Version	26
			5.6.2	Offline Multiple Points Version	27
5.7	Closest Pair	29			
5.8	Minimum Enclosing Circle	30			
6	Strings	31			
6.1	Knuth-Morris-Pratt Algorithm	31			
6.2	Z Value	31			
6.3	Manacher's Algorithm	31			
6.4	Minimum Rotation	32			

1. Misc

1.1. Contest

1.1.1. Makefile

```

1 .PRECIOUS: ./p%
3 %: p%
  ulimit -s unlimited && ./$<
5 p%: p%.cpp
  g++ -o $@ $<-std=c++17 -Wall -Wextra -Wshadow \
7   -fsanitize=address,undefined

```

1.2. How Did We Get Here?

1.2.1. Macros

Use vectorizations and math optimizations at your own peril.

For `gcc ≥ 9`, there are `[[likely]]` and `[[unlikely]]` attributes.

Call `gcc` with `-fopt-info-optimized-missed-optall` for optimization info.

```

1 #define _GLIBCXX_DEBUG           1 //for debug mode
# define _GLIBCXX_SANITIZE_VECTOR 1 //for asan on vectors
3 #pragma GCC optimize("O3", "unroll-loops")
# pragma GCC optimize("fast-math")
5 #pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: `lscpu` 
// before a loop
7 #pragma GCC unroll 16 // 0 or 1 -> no unrolling
# pragma GCC ivdep

```

1.2.2. Fast I/O

```

1 struct scanner {
3   static constexpr size_t LEN = 32 << 20;
5   char *buf, *buf_ptr, *buf_end;
scanner()
7     : buf(new char[LEN]), buf_ptr(buf + LEN),
      buf_end(buf + LEN) {}
~scanner() { delete[] buf; }
char getc() {
9     if (buf_ptr == buf_end) [[unlikely]]
11       buf_end = buf + fread_unlocked(buf, 1, LEN, stdin),
13         buf_ptr = buf;
15       return *(buf_ptr++);
}
char seek(char del) {
17   char c;
19   while ((c = getc()) < del) {}
21   return c;
}
void read(int &t) {
23   bool neg = false;
25   char c = seek('-');
if (c == '-') neg = true, t = 0;
else t = c ^ '0';
while ((c = getc()) >= '0') t = t * 10 + (c ^ '0');
if (neg) t = -t;
}

```

```

27 };
28 struct printer {
29     static constexpr size_t CPI = 21, LEN = 32 << 20;
30     char *buf, *buf_ptr, *buf_end, *tbuf;
31     char *int_buf, *int_buf_end;
32     printer()
33         : buf(new char[LEN]), buf_ptr(buf),
34             buf_end(buf + LEN), int_buf(new char[CPI + 1][0]),
35             int_buf_end(int_buf + CPI - 1) {}
36     ~printer() {
37         flush();
38         delete[] buf, delete[] int_buf;
39     }
40     void flush() {
41         fwrite_unlocked(buf, 1, buf_ptr - buf, stdout);
42         buf_ptr = buf;
43     }
44     void write_(const char &c) {
45         *buf_ptr = c;
46         if (++buf_ptr == buf_end) [[unlikely]]
47             flush();
48     }
49     void write_(const char *s) {
50         for (; *s != '\0'; ++s) write_(*s);
51     }
52     void write(int x) {
53         if (x < 0) write_('-'), x = -x;

```

```

55     if (x == 0) [[unlikely]]
56         return write_('0');
57     for (tbuf = int_buf_end; x != 0; --tbuf, x /= 10)
58         *tbuf = '0' + char(x % 10);
59         write_(++tbuf);
60     }
61 };

```

Kotlin

```

1 import java.io.*
2 import java.util.*
3
4     @JvmField val cin = System.`in`.bufferedReader()
5     @JvmField val cout = PrintWriter(System.out, false)
6     @JvmField var tokenizer: StringTokenizer = StringTokenizer("")
7     fun nextLine() = cin.readLine()!!
8     fun read(): String {
9         while (!tokenizer.hasMoreTokens())
10             tokenizer = StringTokenizer(nextLine())
11         return tokenizer.nextToken()
12     }
13
14     // example
15     fun main() {
16         val n = read().toInt()
17         val a = DoubleArray(n) { read().toDouble() }

```

```
19 cout.println("omg hi")
  cout.flush()
}
```

1.2.3. constexpr

Some default limits in gcc (7.x - trunk):

- constexpr recursion depth: 512
- constexpr loop iteration per function: 262 144
- constexpr operation count per function: 33 554 432
- template recursion depth: 900 (gcc *might* segfault first)

```
1 constexpr array<int, 10> fibonacci[] {
  array<int, 10> a{};
  a[0] = a[1] = 1;
  for (int i = 2; i < 10; i++) a[i] = a[i - 1] + a[i - 2];
  return a;
}();
7 static_assert(fibonacci[9] == 55, "CE");

9 template <typename F, typename INT, INT... S>
constexpr void for_constexpr(integer_sequence<INT, S...>,
11           F &&func) {
  int _[] = {(func(integral_constant<INT, S>{}), 0)...};
13 }
// example
15 template <typename... T> void print_tuple(tuple<T...> t) {
  for_constexpr(make_index_sequence<sizeof...(T)>{},
17    [&](auto i) { cout << get<i>(t) << '\n'; });
}
```

```
}
```

1.2.4. Bump Allocator

```
1
3
5 // global bump allocator
7 char mem[256 << 20]; // 256 MB
9 size_t rsp = sizeof mem;
11 void*operator new(size_t s) {
  assert(s < rsp); // MLE
  return (void*)&mem[rsp -= s];
}
13 // bump allocator for STL / pbds containers
15 char mem[256 << 20];
17 size_t rsp = sizeof mem;
19 template <typename T> struct bump {
21   typedef T value_type;
23   bump() {}
25   template <typename U> bump(U, ...) {
27     T *allocate(size_t n) {
29       rsp -= n * sizeof(T);
31       rsp &= 0 - alignof(T);
33       return (T*)(mem + rsp);
35   }}
```

```
25 void deallocate(T * , size_t n) {}  
};
```

1.3. Tools

1.3.1. Floating Point Binary Search

```
1 union di {  
2     double d;  
3     ull i;  
4 };  
5 bool check(double);  
// binary search in [L, R) with relative error 2^-eps  
7 double binary_search(double L, double R, int eps) {  
8     di l = {L}, r = {R}, m;  
9     while (r.i - l.i > 1LL << (52 - eps)) {  
10        m.i = (l.i + r.i) >> 1;  
11        if (check(m.d)) r = m;  
12        else l = m;  
13    }  
14    return l.d;  
15 }
```

1.3.2. SplitMix64

```
1 using ull = unsigned long long;  
2 inline ull splitmix64(ull x) {  
3     // change to `static ull x = SEED;` for DRBG  
4     ull z = (x += 0x9E3779B97F4A7C15);  
5     z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
```

```
7     z = (z ^ (z >> 27)) * 0x94D049BB133111EB;  
8     return z ^ (z >> 31);  
9 }
```

1.3.3. <random>

```
1 #ifdef __unix__  
2 random_device rd;  
3 mt19937_64 RNG(rd());  
4 #else  
5 const auto SEED = chrono::high_resolution_clock::now()  
6         .time_since_epoch()  
7         .count();  
8 mt19937_64 RNG(SEED);  
9 #endif  
10 // random uint_fast64_t: RNG();  
11 // uniform random of type T (int, double, ...) in [l, r]:  
12 // uniform_int_distribution<T> dist(l, r); dist(RNG);
```

1.3.4. x86 Stack Hack

```
1 constexpr size_t size = 200 << 20; // 200MiB  
2 int main() {  
3     register long rsp asm("rsp");  
4     char *buf = new char[size];  
5     asm("movq %0, %%rsp\n" :: "r"(buf + size));  
6     // do stuff  
7     asm("movq %0, %%rsp\n" :: "r"(rsp));  
8     delete[] buf;  
9 }
```

1.4. Algorithms

1.4.1. Bit Hacks

```

1 // next permutation of x as a bit sequence
ull next_bits_permutation(ull x) {
3 ull c = __builtin_ctzll(x), r = x + (1ULL << c);
    return (r ^ x) >> (c + 2) | r;
5 }
// iterate over all (proper) subsets of bitset s
7 void subsets(ull s) {
9     for (ull x = s; x;) { --x &= s; /* do stuff */ }

```

1.4.2. Infinite Grid Knight Distance

```

1 ll get_dist(ll dx, ll dy) {
2     if (++(dx = abs(dx)) > ++(dy = abs(dy))) swap(dx, dy);
3     if (dx == 1 && dy == 2) return 3;
4     if (dx == 3 && dy == 3) return 4;
5     ll lb = max(dy / 2, (dx + dy) / 3);
6     return ((dx ^ dy ^ lb) & 1) ? ++lb : lb;
7 }

```

1.4.3. Longest Increasing Subsequence

```

1
3 template <class I> vi lis(const vector<I> &S) {
4     if (S.empty()) return {};
5     vi prev(sz(S));

```

```

7     typedef pair<I, intauto it = lower_bound(all(res), p{S[i], 0});
15         if (it == res.end())
17             res.emplace_back(), it = res.end() - 1;
19         *it = {S[i], i};
21         prev[i] = it == res.begin() ? 0 : (it - 1)->second;
23     }
25     int L = sz(res), cur = res.back().second;
27     vi ans(L);
29     while (L--) ans[L] = cur, cur = prev[cur];
31     return ans;
33 }

```

1.4.4. Mo's Algorithm on Tree

```

1 void MoAlgoOnTree() {
2     Dfs(0, -1);
3     vector<int> euler(tk);
4     for (int i = 0; i < n; ++i) {
5         euler[tin[i]] = i;
6         euler[tout[i]] = i;
7     }
8     vector<int> l(q), r(q), qr(q), sp(q, -1);
9     for (int i = 0; i < q; ++i) {
10         if (tin[u[i]] > tin[v[i]]) swap(u[i], v[i]);
11     }
12 }

```

```
11 | int z = GetLCA(u[i], v[i]);
  | sp[i] = z[i];
13 | if (z == u) l[i] = tin[u[i]], r[i] = tin[v[i]];
  | else l[i] = tout[u[i]], r[i] = tin[v[i]];
15 | qr[i] = i;
  |
17 | sort(qr.begin(), qr.end(), [&](int i, int j) {
  |   if (l[i] / kB == l[j] / kB) return r[i] < r[j];
19 |   return l[i] / kB < l[j] / kB;
  | });
21 | vector<bool> used(n);
// Add(v): add/remove v to/from the path based on used[v]
23 | for (int i = 0, tl = 0, tr = -1; i < q; ++i) {
  |   while (tl < l[qr[i]]) Add(euler[tl++]);
25 |   while (tl > l[qr[i]]) Add(euler[--tl]);
  |   while (tr > r[qr[i]]) Add(euler[tr--]);
27 |   while (tr < r[qr[i]]) Add(euler[++tr]);
// add/remove LCA(u, v) if necessary
29 | }
```

2. Data Structures

2.1. GNU PBDS

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/priority_queue.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace __gnu_pbds;
5
6 // most std::map + order_of_key, find_by_order, split, join
7 template <typename T, typename U = null_type>
8 using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
9                 tree_order_statistics_node_update>;
10 // useful tags: rb_tree_tag, splay_tree_tag
11
12 template <typename T> struct myhash {
13     size_t operator()(T x) const; // splitmix, bswap(x*R), ...
14 };
15 // most of std::unordered_map, but faster (needs good hash)
16 template <typename T, typename U = null_type>
17 using hash_table = gp_hash_table<T, U, myhash<T>>;
18
19 // most std::priority_queue + modify, erase, split, join
20 using heap = priority_queue<int, std::less<>>;
21 // useful tags: pairing_heap_tag, binary_heap_tag,
22 //               (rc_)?binomial_heap_tag, thin_heap_tag

```

2.2. 2D Partial Sums

```
1 using vvi = vector<vector<int>>;
```

```

3 using vvll = vector<vector<ll>>;
4 using vll = vector<ll>;
5
6 struct PrefixSum2D {
7     vvll pref;           // 0-based 2-D prefix sum
8     void build(const vvll &v) { // creates a copy
9         int n = v.size(), m = v[0].size();
10        pref.assign(n, vll(m, 0));
11        pref[0][0] = v[0][0];
12        for (int i = 1; i < n; i++) {
13            for (int j = 1; j < m; j++) {
14                pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +
15                           (j ? pref[i][j - 1] : 0) -
16                           (i && j ? pref[i - 1][j - 1] : 0);
17            }
18        }
19    }
20
21    ll query(int ulx, int uly, int brx, int bry) const {
22        ll ans = pref[brx][bry];
23        if (ulx) ans -= pref[ulx - 1][bry];
24        if (uly) ans -= pref[brx][uly - 1];
25        if (ulx && uly) ans += pref[ulx - 1][uly - 1];
26        return ans;
27    }
28
29    ll query(int ulx, int uly, int size) const {
30        return query(ulx, uly, ulx + size - 1, uly + size - 1);
31    }
32};
```

```

29 struct PartialSum2D : PrefixSum2D {
30     vvl diff; // 0 based
31     int n, m;
32     PartialSum2D(int _n, int _m) : n(_n), m(_m) {
33         diff.assign(n + 1, vll(m + 1, 0));
34     }
35     // add c from [ulx,uly] to [brx,bry]
36     void update(int ulx, int uly, int brx, int bry, ll c) {
37         diff[ulx][uly] += c;
38         diff[ulx][bry + 1] -= c;
39         diff[brx + 1][uly] -= c;
40         diff[brx + 1][bry + 1] += c;
41     }
42     void update(int ulx, int uly, int size, ll c) {
43         int brx = ulx + size - 1;
44         int bry = uly + size - 1;
45         update(ulx, uly, brx, bry, c);
46     }
47     // process the grid using prefix sum
48     void process() { this->build(diff); }
49 };
50 // usage
51 PrefixSum2D pref;
52 pref.build(v); // takes 2d 0-based vector as input
53 pref.query(x1, y1, x2, y2); // sum of region
54
55 PartialSum2D part(n, m); // dimension of grid 0 based

```

```

56     part.update(x1, y1, x2, y2, 1); // add 1 in region
57     // must run after all updates
58     part.process(); // prefix sum on diff array
59     // only exists after processing
60     vvl &grid = part.pref; // processed diff array
61     part.query(x1, y1, x2, y2); // gives sum of region

```

2.3. Heavy-Light Decomposition

```

1
2
3 template <bool VALS_EDGES> struct HLD {
4     int N, tim = 0;
5     vector<vi> adj;
6     vi par, siz, depth, rt, pos;
7     Node *tree;
8     HLD(vector<vi> adj_)
9         : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1),
10           depth(N), rt(N), pos(N), tree(new Node(0, N)) {
11         dfsSz(0);
12         dfsHld(0);
13     }
14     void dfsSz(int v) {
15         if (par[v] != -1)
16             adj[v].erase(find(all(adj[v]), par[v]));
17         for (int &u : adj[v]) {
18             par[u] = v, depth[u] = depth[v] + 1;
19             dfsSz(u);
20         }
21     }
22     void dfsHld(int v) {
23         if (par[v] == -1)
24             return;
25         if (siz[par[v]] < siz[v])
26             swap(par[v], v);
27         if (depth[par[v]] < depth[v])
28             swap(par[v], v);
29         if (rt[par[v]] == -1)
30             rt[par[v]] = v;
31         if (siz[par[v]] <= 1)
32             return;
33         dfsHld(v);
34         dfsHld(par[v]);
35     }
36 }
37
38 Node Node::operator=(const Node& n) {
39     val = n.val;
40     sz = n.sz;
41     depth = n.depth;
42     rt = n.rt;
43     pos = n.pos;
44     adj = n.adj;
45     par = n.par;
46     return *this;
47 }
48
49 Node Node::operator-(<operator>) {
50     val = -val;
51     sz = -sz;
52     depth = -depth;
53     rt = -rt;
54     pos = -pos;
55     adj = -adj;
56     par = -par;
57     return *this;
58 }
59
60 Node Node::operator+=(const Node& n) {
61     val += n.val;
62     sz += n.sz;
63     depth += n.depth;
64     rt = n.rt;
65     pos = n.pos;
66     adj = n.adj;
67     par = n.par;
68     return *this;
69 }
70
71 Node Node::operator-=(const Node& n) {
72     val -= n.val;
73     sz -= n.sz;
74     depth -= n.depth;
75     rt = -n.rt;
76     pos = -n.pos;
77     adj = -n.adj;
78     par = -n.par;
79     return *this;
80 }
81
82 Node Node::operator*=(const Node& n) {
83     val *= n.val;
84     sz *= n.sz;
85     depth *= n.depth;
86     rt = n.rt;
87     pos = n.pos;
88     adj = n.adj;
89     par = n.par;
90     return *this;
91 }
92
93 Node Node::operator/=(const Node& n) {
94     val /= n.val;
95     sz /= n.sz;
96     depth /= n.depth;
97     rt = -n.rt;
98     pos = -n.pos;
99     adj = -n.adj;
100    par = -n.par;
101    return *this;
102 }
103
104 Node Node::operator%=(const Node& n) {
105     val %= n.val;
106     sz %= n.sz;
107     depth %= n.depth;
108     rt = -n.rt;
109     pos = -n.pos;
110     adj = -n.adj;
111     par = -n.par;
112     return *this;
113 }
114
115 Node Node::operator^=(const Node& n) {
116     val ^= n.val;
117     sz ^= n.sz;
118     depth ^= n.depth;
119     rt = -n.rt;
120     pos = -n.pos;
121     adj = -n.adj;
122     par = -n.par;
123     return *this;
124 }
125
126 Node Node::operator|=(const Node& n) {
127     val |= n.val;
128     sz |= n.sz;
129     depth |= n.depth;
130     rt = -n.rt;
131     pos = -n.pos;
132     adj = -n.adj;
133     par = -n.par;
134     return *this;
135 }
136
137 Node Node::operator&=(const Node& n) {
138     val &= n.val;
139     sz &= n.sz;
140     depth &= n.depth;
141     rt = -n.rt;
142     pos = -n.pos;
143     adj = -n.adj;
144     par = -n.par;
145     return *this;
146 }
147
148 Node Node::operator~() {
149     val = ~val;
150     sz = -sz;
151     depth = -depth;
152     rt = -rt;
153     pos = -pos;
154     adj = -adj;
155     par = -par;
156     return *this;
157 }
158
159 Node Node::operator<operator> {
160     val < operator>;
161     sz < operator>;
162     depth < operator>;
163     rt < operator>;
164     pos < operator>;
165     adj < operator>;
166     par < operator>;
167     return *this;
168 }
169
170 Node Node::operator>>(const Node& n) {
171     val >> n.val;
172     sz >> n.sz;
173     depth >> n.depth;
174     rt >= n.rt;
175     pos >= n.pos;
176     adj >= n.adj;
177     par >= n.par;
178     return *this;
179 }
180
181 Node Node::operator<<(const Node& n) {
182     val << n.val;
183     sz << n.sz;
184     depth << n.depth;
185     rt <= n.rt;
186     pos <= n.pos;
187     adj <= n.adj;
188     par <= n.par;
189     return *this;
190 }
191
192 Node Node::operator<operator> {
193     val < operator>;
194     sz < operator>;
195     depth < operator>;
196     rt < operator>;
197     pos < operator>;
198     adj < operator>;
199     par < operator>;
200     return *this;
201 }
202
203 Node Node::operator<operator> {
204     val < operator>;
205     sz < operator>;
206     depth < operator>;
207     rt < operator>;
208     pos < operator>;
209     adj < operator>;
210     par < operator>;
211     return *this;
212 }
213
214 Node Node::operator<operator> {
215     val < operator>;
216     sz < operator>;
217     depth < operator>;
218     rt < operator>;
219     pos < operator>;
220     adj < operator>;
221     par < operator>;
222     return *this;
223 }
224
225 Node Node::operator<operator> {
226     val < operator>;
227     sz < operator>;
228     depth < operator>;
229     rt < operator>;
230     pos < operator>;
231     adj < operator>;
232     par < operator>;
233     return *this;
234 }
235
236 Node Node::operator<operator> {
237     val < operator>;
238     sz < operator>;
239     depth < operator>;
240     rt < operator>;
241     pos < operator>;
242     adj < operator>;
243     par < operator>;
244     return *this;
245 }
246
247 Node Node::operator<operator> {
248     val < operator>;
249     sz < operator>;
250     depth < operator>;
251     rt < operator>;
252     pos < operator>;
253     adj < operator>;
254     par < operator>;
255     return *this;
256 }
257
258 Node Node::operator<operator> {
259     val < operator>;
260     sz < operator>;
261     depth < operator>;
262     rt < operator>;
263     pos < operator>;
264     adj < operator>;
265     par < operator>;
266     return *this;
267 }
268
269 Node Node::operator<operator> {
270     val < operator>;
271     sz < operator>;
272     depth < operator>;
273     rt < operator>;
274     pos < operator>;
275     adj < operator>;
276     par < operator>;
277     return *this;
278 }
279
280 Node Node::operator<operator> {
281     val < operator>;
282     sz < operator>;
283     depth < operator>;
284     rt < operator>;
285     pos < operator>;
286     adj < operator>;
287     par < operator>;
288     return *this;
289 }
290
291 Node Node::operator<operator> {
292     val < operator>;
293     sz < operator>;
294     depth < operator>;
295     rt < operator>;
296     pos < operator>;
297     adj < operator>;
298     par < operator>;
299     return *this;
300 }
301
302 Node Node::operator<operator> {
303     val < operator>;
304     sz < operator>;
305     depth < operator>;
306     rt < operator>;
307     pos < operator>;
308     adj < operator>;
309     par < operator>;
310     return *this;
311 }
312
313 Node Node::operator<operator> {
314     val < operator>;
315     sz < operator>;
316     depth < operator>;
317     rt < operator>;
318     pos < operator>;
319     adj < operator>;
320     par < operator>;
321     return *this;
322 }
323
324 Node Node::operator<operator> {
325     val < operator>;
326     sz < operator>;
327     depth < operator>;
328     rt < operator>;
329     pos < operator>;
330     adj < operator>;
331     par < operator>;
332     return *this;
333 }
334
335 Node Node::operator<operator> {
336     val < operator>;
337     sz < operator>;
338     depth < operator>;
339     rt < operator>;
340     pos < operator>;
341     adj < operator>;
342     par < operator>;
343     return *this;
344 }
345
346 Node Node::operator<operator> {
347     val < operator>;
348     sz < operator>;
349     depth < operator>;
350     rt < operator>;
351     pos < operator>;
352     adj < operator>;
353     par < operator>;
354     return *this;
355 }
356
357 Node Node::operator<operator> {
358     val < operator>;
359     sz < operator>;
360     depth < operator>;
361     rt < operator>;
362     pos < operator>;
363     adj < operator>;
364     par < operator>;
365     return *this;
366 }
367
368 Node Node::operator<operator> {
369     val < operator>;
370     sz < operator>;
371     depth < operator>;
372     rt < operator>;
373     pos < operator>;
374     adj < operator>;
375     par < operator>;
376     return *this;
377 }
378
379 Node Node::operator<operator> {
380     val < operator>;
381     sz < operator>;
382     depth < operator>;
383     rt < operator>;
384     pos < operator>;
385     adj < operator>;
386     par < operator>;
387     return *this;
388 }
389
390 Node Node::operator<operator> {
391     val < operator>;
392     sz < operator>;
393     depth < operator>;
394     rt < operator>;
395     pos < operator>;
396     adj < operator>;
397     par < operator>;
398     return *this;
399 }
400
401 Node Node::operator<operator> {
402     val < operator>;
403     sz < operator>;
404     depth < operator>;
405     rt < operator>;
406     pos < operator>;
407     adj < operator>;
408     par < operator>;
409     return *this;
410 }
411
412 Node Node::operator<operator> {
413     val < operator>;
414     sz < operator>;
415     depth < operator>;
416     rt < operator>;
417     pos < operator>;
418     adj < operator>;
419     par < operator>;
420     return *this;
421 }
422
423 Node Node::operator<operator> {
424     val < operator>;
425     sz < operator>;
426     depth < operator>;
427     rt < operator>;
428     pos < operator>;
429     adj < operator>;
430     par < operator>;
431     return *this;
432 }
433
434 Node Node::operator<operator> {
435     val < operator>;
436     sz < operator>;
437     depth < operator>;
438     rt < operator>;
439     pos < operator>;
440     adj < operator>;
441     par < operator>;
442     return *this;
443 }
444
445 Node Node::operator<operator> {
446     val < operator>;
447     sz < operator>;
448     depth < operator>;
449     rt < operator>;
450     pos < operator>;
451     adj < operator>;
452     par < operator>;
453     return *this;
454 }
455
456 Node Node::operator<operator> {
457     val < operator>;
458     sz < operator>;
459     depth < operator>;
460     rt < operator>;
461     pos < operator>;
462     adj < operator>;
463     par < operator>;
464     return *this;
465 }
466
467 Node Node::operator<operator> {
468     val < operator>;
469     sz < operator>;
470     depth < operator>;
471     rt < operator>;
472     pos < operator>;
473     adj < operator>;
474     par < operator>;
475     return *this;
476 }
477
478 Node Node::operator<operator> {
479     val < operator>;
480     sz < operator>;
481     depth < operator>;
482     rt < operator>;
483     pos < operator>;
484     adj < operator>;
485     par < operator>;
486     return *this;
487 }
488
489 Node Node::operator<operator> {
490     val < operator>;
491     sz < operator>;
492     depth < operator>;
493     rt < operator>;
494     pos < operator>;
495     adj < operator>;
496     par < operator>;
497     return *this;
498 }
499
500 Node Node::operator<operator> {
501     val < operator>;
502     sz < operator>;
503     depth < operator>;
504     rt < operator>;
505     pos < operator>;
506     adj < operator>;
507     par < operator>;
508     return *this;
509 }
510
511 Node Node::operator<operator> {
512     val < operator>;
513     sz < operator>;
514     depth < operator>;
515     rt < operator>;
516     pos < operator>;
517     adj < operator>;
518     par < operator>;
519     return *this;
520 }
521
522 Node Node::operator<operator> {
523     val < operator>;
524     sz < operator>;
525     depth < operator>;
526     rt < operator>;
527     pos < operator>;
528     adj < operator>;
529     par < operator>;
530     return *this;
531 }
532
533 Node Node::operator<operator> {
534     val < operator>;
535     sz < operator>;
536     depth < operator>;
537     rt < operator>;
538     pos < operator>;
539     adj < operator>;
540     par < operator>;
541     return *this;
542 }
543
544 Node Node::operator<operator> {
545     val < operator>;
546     sz < operator>;
547     depth < operator>;
548     rt < operator>;
549     pos < operator>;
550     adj < operator>;
551     par < operator>;
552     return *this;
553 }
554
555 Node Node::operator<operator> {
556     val < operator>;
557     sz < operator>;
558     depth < operator>;
559     rt < operator>;
560     pos < operator>;
561     adj < operator>;
562     par < operator>;
563     return *this;
564 }
565
566 Node Node::operator<operator> {
567     val < operator>;
568     sz < operator>;
569     depth < operator>;
570     rt < operator>;
571     pos < operator>;
572     adj < operator>;
573     par < operator>;
574     return *this;
575 }
576
577 Node Node::operator<operator> {
578     val < operator>;
579     sz < operator>;
580     depth < operator>;
581     rt < operator>;
582     pos < operator>;
583     adj < operator>;
584     par < operator>;
585     return *this;
586 }
587
588 Node Node::operator<operator> {
589     val < operator>;
590     sz < operator>;
591     depth < operator>;
592     rt < operator>;
593     pos < operator>;
594     adj < operator>;
595     par < operator>;
596     return *this;
597 }
598
599 Node Node::operator<operator> {
600     val < operator>;
601     sz < operator>;
602     depth < operator>;
603     rt < operator>;
604     pos < operator>;
605     adj < operator>;
606     par < operator>;
607     return *this;
608 }
609
610 Node Node::operator<operator> {
611     val < operator>;
612     sz < operator>;
613     depth < operator>;
614     rt < operator>;
615     pos < operator>;
616     adj < operator>;
617     par < operator>;
618     return *this;
619 }
620
621 Node Node::operator<operator> {
622     val < operator>;
623     sz < operator>;
624     depth < operator>;
625     rt < operator>;
626     pos < operator>;
627     adj < operator>;
628     par < operator>;
629     return *this;
630 }
631
632 Node Node::operator<operator> {
633     val < operator>;
634     sz < operator>;
635     depth < operator>;
636     rt < operator>;
637     pos < operator>;
638     adj < operator>;
639     par < operator>;
640     return *this;
641 }
642
643 Node Node::operator<operator> {
644     val < operator>;
645     sz < operator>;
646     depth < operator>;
647     rt < operator>;
648     pos < operator>;
649     adj < operator>;
650     par < operator>;
651     return *this;
652 }
653
654 Node Node::operator<operator> {
655     val < operator>;
656     sz < operator>;
657     depth < operator>;
658     rt < operator>;
659     pos < operator>;
660     adj < operator>;
661     par < operator>;
662     return *this;
663 }
664
665 Node Node::operator<operator> {
666     val < operator>;
667     sz < operator>;
668     depth < operator>;
669     rt < operator>;
670     pos < operator>;
671     adj < operator>;
672     par < operator>;
673     return *this;
674 }
675
676 Node Node::operator<operator> {
677     val < operator>;
678     sz < operator>;
679     depth < operator>;
680     rt < operator>;
681     pos < operator>;
682     adj < operator>;
683     par < operator>;
684     return *this;
685 }
686
687 Node Node::operator<operator> {
688     val < operator>;
689     sz < operator>;
690     depth < operator>;
691     rt < operator>;
692     pos < operator>;
693     adj < operator>;
694     par < operator>;
695     return *this;
696 }
697
698 Node Node::operator<operator> {
699     val < operator>;
700     sz < operator>;
701     depth < operator>;
702     rt < operator>;
703     pos < operator>;
704     adj < operator>;
705     par < operator>;
706     return *this;
707 }
708
709 Node Node::operator<operator> {
710     val < operator>;
711     sz < operator>;
712     depth < operator>;
713     rt < operator>;
714     pos < operator>;
715     adj < operator>;
716     par < operator>;
717     return *this;
718 }
719
720 Node Node::operator<operator> {
721     val < operator>;
722     sz < operator>;
723     depth < operator>;
724     rt < operator>;
725     pos < operator>;
726     adj < operator>;
727     par < operator>;
728     return *this;
729 }
730
731 Node Node::operator<operator> {
732     val < operator>;
733     sz < operator>;
734     depth < operator>;
735     rt < operator>;
736     pos < operator>;
737     adj < operator>;
738     par < operator>;
739     return *this;
740 }
741
742 Node Node::operator<operator> {
743     val < operator>;
744     sz < operator>;
745     depth < operator>;
746     rt < operator>;
747     pos < operator>;
748     adj < operator>;
749     par < operator>;
750     return *this;
751 }
752
753 Node Node::operator<operator> {
754     val < operator>;
755     sz < operator>;
756     depth < operator>;
757     rt < operator>;
758     pos < operator>;
759     adj < operator>;
760     par < operator>;
761     return *this;
762 }
763
764 Node Node::operator<operator> {
765     val < operator>;
766     sz < operator>;
767     depth < operator>;
768     rt < operator>;
769     pos < operator>;
770     adj < operator>;
771     par < operator>;
772     return *this;
773 }
774
775 Node Node::operator<operator> {
776     val < operator>;
777     sz < operator>;
778     depth < operator>;
779     rt < operator>;
780     pos < operator>;
781     adj < operator>;
782     par < operator>;
783     return *this;
784 }
785
786 Node Node::operator<operator> {
787     val < operator>;
788     sz < operator>;
789     depth < operator>;
790     rt < operator>;
791     pos < operator>;
792     adj < operator>;
793     par < operator>;
794     return *this;
795 }
796
797 Node Node::operator<operator> {
798     val < operator>;
799     sz < operator>;
800     depth < operator>;
801     rt < operator>;
802     pos < operator>;
803     adj < operator>;
804     par < operator>;
805     return *this;
806 }
807
808 Node Node::operator<operator> {
809     val < operator>;
810     sz < operator>;
811     depth < operator>;
812     rt < operator>;
813     pos < operator>;
814     adj < operator>;
815     par < operator>;
816     return *this;
817 }
818
819 Node Node::operator<operator> {
820     val < operator>;
821     sz < operator>;
822     depth < operator>;
823     rt < operator>;
824     pos < operator>;
825     adj < operator>;
826     par < operator>;
827     return *this;
828 }
829
830 Node Node::operator<operator> {
831     val < operator>;
832     sz < operator>;
833     depth < operator>;
834     rt < operator>;
835     pos < operator>;
836     adj < operator>;
837     par < operator>;
838     return *this;
839 }
840
841 Node Node::operator<operator> {
842     val < operator>;
843     sz < operator>;
844     depth < operator>;
845     rt < operator>;
846     pos < operator>;
847     adj < operator>;
848     par < operator>;
849     return *this;
850 }
851
852 Node Node::operator<operator> {
853     val < operator>;
854     sz < operator>;
855     depth < operator>;
856     rt < operator>;
857     pos < operator>;
858     adj < operator>;
859     par < operator>;
860     return *this;
861 }
862
863 Node Node::operator<operator> {
864     val < operator>;
865     sz < operator>;
866     depth < operator>;
867     rt < operator>;
868     pos < operator>;
869     adj < operator>;
870     par < operator>;
871     return *this;
872 }
873
874 Node Node::operator<operator> {
875     val < operator>;
876     sz < operator>;
877     depth < operator>;
878     rt < operator>;
879     pos < operator>;
880     adj < operator>;
881     par < operator>;
882     return *this;
883 }
884
885 Node Node::operator<operator> {
886     val < operator>;
887     sz < operator>;
888     depth < operator>;
889     rt < operator>;
890     pos < operator>;
891     adj < operator>;
892     par < operator>;
893     return *this;
894 }
895
896 Node Node::operator<operator> {
897     val < operator>;
898     sz < operator>;
899     depth < operator>;
900     rt < operator>;
901     pos < operator>;
902     adj < operator>;
903     par < operator>;
904     return *this;
905 }
906
907 Node Node::operator<operator> {
908     val < operator>;
909     sz < operator>;
910     depth < operator>;
911     rt < operator>;
912     pos < operator>;
913     adj < operator>;
914     par < operator>;
915     return *this;
916 }
917
918 Node Node::operator<operator> {
919     val < operator>;
920     sz < operator>;
921     depth < operator>;
922     rt < operator>;
923     pos < operator>;
924     adj < operator>;
925     par < operator>;
926     return *this;
927 }
928
929 Node Node::operator<operator> {
930     val < operator>;
931     sz < operator>;
932     depth < operator>;
933     rt < operator>;
934     pos < operator>;
935     adj < operator>;
936     par < operator>;
937     return *this;
938 }
939
940 Node Node::operator<operator> {
941     val < operator>;
942     sz < operator>;
943     depth < operator>;
944     rt < operator>;
945     pos < operator>;
946     adj < operator>;
947     par < operator>;
948     return *this;
949 }
950
951 Node Node::operator<operator> {
952     val < operator>;
953     sz < operator>;
954     depth < operator>;
955     rt < operator>;
956     pos < operator>;
957     adj < operator>;
958     par < operator>;
959     return *this;
960 }
961
962 Node Node::operator<operator> {
963     val < operator>;
964     sz < operator>;
965     depth < operator>;
966     rt < operator>;
967     pos < operator>;
968     adj < operator>;
969     par < operator>;
970     return *this;
971 }
972
973 Node Node::operator<operator> {
974     val < operator>;
975     sz < operator>;
976     depth < operator>;
977     rt < operator>;
978     pos < operator>;
979     adj < operator>;
980     par < operator>;
981     return *this;
982 }
983
984 Node Node::operator<operator> {
985     val < operator>;
986     sz < operator>;
987     depth < operator>;
988     rt < operator>;
989     pos < operator>;
990     adj < operator>;
991     par < operator>;
992     return *this;
993 }
994
995 Node Node::operator<operator> {
996     val < operator>;
997     sz < operator>;
998     depth < operator>;
999     rt < operator>;
1000    pos < operator>;
1001    adj < operator>;
1002    par < operator>;
1003    return *this;
1004 }
1005
1006 Node Node::operator<operator> {
1007     val < operator>;
1008     sz < operator>;
1009     depth < operator>;
1010     rt < operator>;
1011     pos < operator>;
1012     adj < operator>;
1013     par < operator>;
1014
```

```
21     siz[v] += siz[u];
22     if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
23   }
24
25   void dfsHld(int v) {
26     pos[v] = tim++;
27     for (int u : adj[v]) {
28       rt[u] = (u == adj[v][0] ? rt[v] : u);
29       dfsHld(u);
30     }
31   }
32
33   template <class B> void process(int u, int v, B op) {
34     for (; rt[u] != rt[v]; v = par[rt[v]]) {
35       if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
36       op(pos[rt[v]], pos[v] + 1);
37     }
38     if (depth[u] > depth[v]) swap(u, v);
39     op(pos[u] + VALS_EDGES, pos[v] + 1);
40   }
41
42   void modifyPath(int u, int v, int val) {
43     process(u, v,
44             [&](int l, int r) { tree->add(l, r, val); });
45   }
46
47   int queryPath(int u,
48                 int v) { // Modify depending on problem
49     int res = -1e9;
50     process(u, v, [&](int l, int r) {
51       res = max(res, tree->query(l, r));
52     });
53     return res;
54   }
55
56   int querySubtree(int v) { // modifySubtree is similar
57     return tree->query(pos[v] + VALS_EDGES,
58                         pos[v] + siz[v]);
59   }
60 }
```

3. Graph

3.1. Matching/Flows

3.1.1. Kuhn-Munkres algorithm

```

1 // Maximum Weight Perfect Bipartite Matching
// Detect non-perfect-matching:
3 // 1. set all edge[i][j] as INF
// 2. if solve() >= INF, it is not perfect matching.
5
6 typedef long long ll;
7 struct KM {
8     static const int MAXN = 1050;
9     static const ll INF = 1LL << 60;
10    int n, match[MAXN], vx[MAXN], vy[MAXN];
11    ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
12    void init(int _n) {
13        n = _n;
14        for (int i = 0; i < n; i++)
15            for (int j = 0; j < n; j++) edge[i][j] = 0;
16    }
17    void add_edge(int x, int y, ll w) { edge[x][y] = w; }
18    bool DFS(int x) {
19        vx[x] = 1;
20        for (int y = 0; y < n; y++) {
21            if (vy[y]) continue;
22            if (lx[x] + ly[y] > edge[x][y]) {
23                slack[y] =

```

```

25                min(slack[y], lx[x] + ly[y] - edge[x][y]);
26            } else {
27                vy[y] = 1;
28                if (match[y] == -1 || DFS(match[y])) {
29                    match[y] = x;
30                    return true;
31                }
32            }
33        }
34        return false;
35    }
36    ll solve() {
37        fill(match, match + n, -1);
38        fill(lx, lx + n, -INF);
39        fill(ly, ly + n, 0);
40        for (int i = 0; i < n; i++)
41            for (int j = 0; j < n; j++)
42                lx[i] = max(lx[i], edge[i][j]);
43        for (int i = 0; i < n; i++) {
44            fill(slack, slack + n, INF);
45            while (true) {
46                fill(vx, vx + n, 0);
47                fill(vy, vy + n, 0);
48                if (DFS(i)) break;
49                ll d = INF;
50                for (int j = 0; j < n; j++)
51                    if (!vy[j]) d = min(d, slack[j]);
52            }
53        }
54    }

```

```

51 |     for (int j = 0; j < n; j++) {
52 |         if (vx[j]) lx[j] -= d;
53 |         if (vy[j]) ly[j] += d;
54 |         else slack[j] -= d;
55 |     }
56 | }
57 | }
58 | ll res = 0;
59 | for (int i = 0; i < n; i++) {
60 |     res += edge[match[i]][i];
61 | }
62 | return res;
63 | }
64 | } graph;

```

```

13 |     for (pii i : v[a])
14 |         if (!inneg[i.F]) dfs(i.F);
15 |     }
16 |     bool solve(int n, int s) { // true if have neg-cycle
17 |         for (int i = 0; i <= n; i++) dis[i] = INF;
18 |         dis[s] = 0, q.push(s);
19 |         for (int i = 0; i < n; i++) {
20 |             inq.reset();
21 |             int now;
22 |             while (!q.empty()) {
23 |                 now = q.front(), q.pop();
24 |                 for (pii &i : v[now]) {
25 |                     if (dis[i.F] > dis[now] + i.S) {
26 |                         dis[i.F] = dis[now] + i.S;
27 |                         if (!inq[i.F]) tq.push(i.F), inq[i.F] = 1;
28 |                     }
29 |                 }
30 |                 q.swap(tq);
31 |             }
32 |             bool re = !q.empty();
33 |             inneg.reset();
34 |             while (!q.empty()) {
35 |                 if (!inneg[q.front()]) dfs(q.front());
36 |                 q.pop();
37 |             }
38 |             return re;

```

3.2. Shortest Path Faster Algorithm

```

1 | struct SPFA {
2 |     static const int maxn = 1010, INF = 1e9;
3 |     int dis[maxn];
4 |     bitset<maxn> inq, inneg;
5 |     queue<int> q, tq;
6 |     vector<pii> v[maxn];
7 |     void make_edge(int s, int t, int w) {
8 |         v[s].emplace_back(t, w);
9 |     }
10 |     void dfs(int a) {
11 |         inneg[a] = 1;

```

```

39  }
40  void reset(int n) {
41    for (int i = 0; i <= n; i++) v[i].clear();
42  }
43 };

```

3.3. Strongly Connected Components

```

1 struct TarjanScc {
2   int n, step;
3   vector<int> time, low, instk, stk;
4   vector<vector<int>> e, scc;
5   TarjanScc(int n_) : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
6   void add_edge(int u, int v) { e[u].push_back(v); }
7   void dfs(int x) {
8     time[x] = low[x] = ++step;
9     stk.push_back(x);
10    instk[x] = 1;
11    for (int y : e[x])
12      if (!time[y]) {
13        dfs(y);
14        low[x] = min(low[x], low[y]);
15      } else if (instk[y]) {
16        low[x] = min(low[x], time[y]);
17      }
18    if (time[x] == low[x]) {
19      scc.emplace_back();

```

```

21   for (int y = -1; y != x;) {
22     y = stk.back();
23     stk.pop_back();
24     instk[y] = 0;
25     scc.back().push_back(y);
26   }
27 }
28
29 void solve() {
30   for (int i = 0; i < n; i++)
31     if (!time[i]) dfs(i);
32   reverse(scc.begin(), scc.end());
33   // scc in topological order
34 }
35 };

```

3.4. Biconnected Components

3.4.1. Articulation Points

```

1 void dfs(int x, int p) {
2   tin[x] = low[x] = ++t;
3   int ch = 0;
4   for (auto u : g[x])
5     if (u.first != p) {
6       if (!ins[u.second])
7         st.push(u.second), ins[u.second] = true;
8       if (tin[u.first])
9         low[x] = min(low[x], tin[u.first]);

```

```

11     continue;
12 }
13     ++ch;
14     dfs(u.first, x);
15     low[x] = min(low[x], low[u.first]);
16     if (low[u.first] >= tin[x]) {
17         cut[x] = true;
18         ++sz;
19         while (true) {
20             int e = st.top();
21             st.pop();
22             bcc[e] = sz;
23             if (e == u.second) break;
24         }
25     }
26     if (ch == 1 && p == -1) cut[x] = false;
27 }

```

```

9     low[x] = min(low[x], tin[u.first]);
10    continue;
11 }
12     dfs(u.first, x);
13     low[x] = min(low[x], low[u.first]);
14     if (low[u.first] == tin[u.first]) br[u.second] = true;
15 }
16 if (tin[x] == low[x]) {
17     ++sz;
18     while (st.size()) {
19         int u = st.top();
20         st.pop();
21         bcc[u] = sz;
22         if (u == x) break;
23     }
24 }

```

3.4.2. Bridges

```

1 // if there are multi-edges, then they are not bridges
2 void dfs(int x, int p) {
3     tin[x] = low[x] = ++t;
4     st.push(x);
5     for (auto u : g[x])
6         if (u.first != p) {
7             if (tin[u.first]) {

```

3.5. Centroid Decomposition

```

1 void get_center(int now) {
2     v[now] = true;
3     vtx.push_back(now);
4     sz[now] = 1;
5     mx[now] = 0;
6     for (int u : G[now])
7         if (!v[u]) {
8             get_center(u);

```

```
9    mx[now] = max(mx[now], sz[u]);
10   sz[now] += sz[u];
11 }
12
13 void get_dis(int now, int d, int len) {
14   dis[d][now] = cnt;
15   v[now] = true;
16   for (auto u : G[now])
17     if (!v[u.first]) { get_dis(u, d, len + u.second); }
18 }
19
20 void dfs(int now, int fa, int d) {
21   get_center(now);
22   int c = -1;
23   for (int i : vtx) {
24     if (max(mx[i], (int)vtx.size() - sz[i]) <=
25         (int)vtx.size() / 2)
26       c = i;
27     v[i] = false;
28   }
29   get_dis(c, d, 0);
30   for (int i : vtx) v[i] = false;
31   v[c] = true;
32   vtx.clear();
33   dep[c] = d;
34   p[c] = fa;
35   for (auto u : G[c])
36     if (u.first != fa && !v[u.first]) {  
37     dfs(u.first, c, d + 1);  
38   }
```

4. Math

4.1. Number Theory

4.1.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467
 910927547, 919012223, 947326223, 990669467, 1007939579, 1019126697
 929760389146037459, 975500632317046523, 989312547895528379

NTT prime p	$p - 1$	primitive root	
65537	$1 \ll 16$	3	21
998244353	$119 \ll 23$	3	23
2748779069441	$5 \ll 39$	3	25
1945555039024054273	$27 \ll 56$	5	

Requires: Extended GCD²⁷

```

1
3 template <typename T> struct M {
4     static T MOD; // change to constexpr if already known
5     T v;
6     M(T x = 0) {
7         v = (-MOD <= x && x < MOD) ? x : x % MOD;
8         if (v < 0) v += MOD;
9     }
10    explicit operator T() const { return v; }
11    bool operator==(const M &b) const { return v == b.v; }
```

```

13    bool operator!=(const M &b) const { return v != b.v; }
14    M operator-() { return M(-v); }
15    M operator+(M b) { return M(v + b.v); }
16    M operator-(M b) { return M(v - b.v); }
17    M operator*(M b) { return M((__int128)v * b.v % MOD); }
18    M operator/(M b) { return *this * (b ^ (MOD - 2)); }
19    // change above implementation to this if MOD is not prime
20    M inv() {
21        auto [p, _, g] = extgcd(v, MOD);
22        return assert(g == 1), p;
23    }
24    friend M operator^(M a, ll b) {
25        M ans(1);
26        for (; b; b >>= 1, a *= a)
27            if (b & 1) ans *= a;
28        return ans;
29    }
30    friend M &operator+=(M &a, M b) { return a = a + b; }
31    friend M &operator-=(M &a, M b) { return a = a - b; }
32    friend M &operator*=(M &a, M b) { return a = a * b; }
33    friend M &operator/=(M &a, M b) { return a = a / b; }
34 };
35 using Mod = M<int>;
36 template <> int Mod::MOD = 1'000'000'007;
37 int &MOD = Mod::MOD;
```

4.1.2. Miller-Rabin

Requires: Mod Struct

```

1
3 // checks if Mod::MOD is prime
4 bool is_prime() {
5   if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
6   Mod A[] = {2, 7, 61}; //for int values (< 2^31)
7   // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
8   int s = __builtin_ctzll(MOD - 1), i;
9   for (Mod a : A) {
10     Mod x = a ^ (MOD >> s);
11     for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
12     if (i && x != -1) return 0;
13   }
14   return 1;
15 }
```

4.1.3. Linear Sieve

```

1 constexpr ll MAXN = 1000000;
2 bitset<MAXN> is_prime;
3 vector<ll> primes;
4 ll mpf[MAXN], phi[MAXN], mu[MAXN];
5
6 void sieve() {
7   is_prime.set();
8   is_prime[1] = 0;
9   mu[1] = phi[1] = 1;
```

```

11   for (ll i = 2; i < MAXN; i++) {
12     if (is_prime[i]) {
13       mpf[i] = i;
14       primes.push_back(i);
15       phi[i] = i - 1;
16       mu[i] = -1;
17     }
18     for (ll p : primes) {
19       if (p > mpf[i] || i * p >= MAXN) break;
20       is_prime[i * p] = 0;
21       mpf[i * p] = p;
22       mu[i * p] = -mu[i];
23       if (i % p == 0)
24         phi[i * p] = phi[i] * p, mu[i * p] = 0;
25       else phi[i * p] = phi[i] * (p - 1);
26     }
27   }
```

4.1.4. Get Factors

Requires: Linear Sieve

```

1
3 vector<ll> all_factors(ll n) {
4   vector<ll> fac = {1};
5   while (n > 1) {
6     const ll p = mpf[n];
```

```

7 vector<ll> cur = {1};
9 while (n % p == 0) {
11     n /= p;
13     cur.push_back(cur.back() * p);
15 }
17 return fac;
}

```

4.1.5. Binary GCD

```

1 // returns the gcd of non-negative a, b
3 ull bin_gcd(ull a, ull b) {
5     if (!a || !b) return a + b;
7     int s = __builtin_ctzll(a | b);
9     a >>= __builtin_ctzll(a);
11    while (b) {
13        if ((b >>= __builtin_ctzll(b)) < a) swap(a, b);
15        b -= a;
17    }
19    return a << s;
}

```

4.1.6. Extended GCD

```

1 // returns (p, q, g): p * a + q * b == g == gcd(a, b)

```

```

3 // g is not guaranteed to be positive when a < 0 or b < 0
5 tuple<ll, ll, ll> extgcd(ll a, ll b) {
7     ll s = 1, t = 0, u = 0, v = 1;
9     while (b) {
11         ll q = a / b;
13         swap(a -= q * b, b);
15         swap(s -= q * t, t);
17         swap(u -= q * v, v);
19     }
21     return {s, u, a};
}

```

4.1.7. Chinese Remainder Theorem

Requires: Extended GCD

```

1 // for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
3 // such that x % m == a and x % n == b
5 ll crt(ll a, ll m, ll b, ll n) {
7     if (n > m) swap(a, b), swap(m, n);
9     auto [x, y, g] = extgcd(m, n);
11    assert((a - b) % g == 0); // no solution
13    x = ((b - a) / g * y) % (n / g) * m + a;
15    return x < 0 ? x + m / g * n : x;
}

```

4.1.8. Pollard's Rho

```

1 ll f(ll x, ll mod) { return (x * x + 1) % mod; }
// n should be composite

```

```

3 ll pollard_rho(ll n) {
4     if (!(n & 1)) return 2;
5     while (1) {
6         ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
7         for (int sz = 2; res == 1; sz *= 2) {
8             for (int i = 0; i < sz && res <= 1; i++) {
9                 x = f(x, n);
10                res = __gcd(abs(x - y), n);
11            }
12            y = x;
13        }
14        if (res != 0 && res != n) return res;
15    }
16}

```

4.1.9. Rational Number Binary Search

```

1 struct QQ {
2     ll p, q;
3     QQ go(QQ b, ll d) { return {p + b.p * d, q + b.q * d}; }
4 };
5     bool pred(QQ);
6     // returns smallest p/q in [lo, hi] such that
7     // pred(p/q) is true, and 0 <= p,q <= N
8     QQ frac_bs(ll N) {
9         QQ lo{0, 1}, hi{1, 0};
10        if (pred(lo)) return lo;
11        assert(pred(hi));

```

```

13     bool dir = 1, L = 1, H = 1;
14     for (; L || H; dir = !dir) {
15         ll len = 0, step = 1;
16         for (int t = 0; t < 2 && (t ? step /= 2 : step *= 2);)
17             if (QQ mid = hi.go(lo, len + step));
18                 mid.p > N || mid.q > N || dir ^ pred(mid))
19                     t++;
20             else len += step;
21             swap(lo, hi = hi.go(lo, len));
22             (dir ? L : H) = !!len;
23         }
24         return dir ? hi : lo;
25     }

```

4.1.10. De Bruijn Sequence

```

1 int res[kN], aux[kN], a[kN], sz;
2 void Rec(int t, int p, int n, int k) {
3     if (t > n) {
4         if (n % p == 0)
5             for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
6         } else {
7             aux[t] = aux[t - p];
8             Rec(t + 1, p, n, k);
9             for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
10                 Rec(t + 1, t, n, k);
11     }
12 }

```

```
13 int DeBruijn(int k, int n) {  
    // return cyclic string of length  $k^n$  such that every  
15 // string of length n using k character appears as a  
    // substring.  
17 if (k == 1) return res[0] = 0, 1;  
    fill(aux, aux + k * n, 0);  
19 return sz = 0, Rec(1, 1, n, k), sz;  
}
```

4.1.11. Long Long Multiplication

```
1 using ull = unsigned long long;  
2 using ll = long long;  
3 using ld = long double;  
// returns  $a * b \% M$  where  $a, b < M < 2^{**63}$   
5 ull mult(ull a, ull b, ull M) {  
    ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));  
7    return ret + M * (ret < 0) - M * (ret >= (ll)M);  
}
```

5. Geometry

5.1. Point

```

1 template <typename T> struct P {
2     T x, y;
3     P(T x = 0, T y = 0) : x(x), y(y) {}
4     bool operator<(const P &p) const {
5         return tie(x, y) < tie(p.x, p.y);
6     }
7     bool operator==(const P &p) const {
8         return tie(x, y) == tie(p.x, p.y);
9     }
10    P operator-() const { return {-x, -y}; }
11    P operator+(P p) const { return {x + p.x, y + p.y}; }
12    P operator-(P p) const { return {x - p.x, y - p.y}; }
13    P operator*(T d) const { return {x * d, y * d}; }
14    P operator/(T d) const { return {x / d, y / d}; }
15    T dist2() const { return x * x + y * y; }
16    double len() const { return sqrt(dist2()); }
17    P unit() const { return *this / len(); }
18    friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
19    friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
20    friend T cross(P a, P b, P o) {
21        return cross(a - o, b - o);
22    }
23};

using pt = P<ll>;

```

5.1.1. Quaternion

```

1 constexpr double PI = 3.141592653589793;
2 constexpr double EPS = 1e-7;
3 struct Q {
4     using T = double;
5     T x, y, z, r;
6     Q(T r = 0) : x(0), y(0), z(0), r(r) {}
7     Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
8     friend bool operator==(const Q &a, const Q &b) {
9         return (a - b).abs2() <= EPS;
10    }
11    friend bool operator!=(const Q &a, const Q &b) {
12        return !(a == b);
13    }
14    Q operator-() { return Q(-x, -y, -z, -r); }
15    Q operator+(const Q &b) const {
16        return Q(x + b.x, y + b.y, z + b.z, r + b.r);
17    }
18    Q operator-(const Q &b) const {
19        return Q(x - b.x, y - b.y, z - b.z, r - b.r);
20    }
21    Q operator*(const T &t) const {
22        return Q(x * t, y * t, z * t, r * t);
23    }
24    Q operator*(const Q &b) const {
25        return Q(r * b.x + x * b.r + y * b.z - z * b.y,
26                  r * b.y - x * b.z + y * b.r + z * b.x,
27                  r * b.z - y * b.r + z * b.y, r * b.r - x * b.y - y * b.x);
28    }
29}
```

```

27     r * b.z + x * b.y - y * b.x + z * b.r,
28     r * b.r - x * b.x - y * b.y - z * b.z);
29 }
30 Q operator/(const Q &b) const { return *this * b.inv(); }
31 T abs2() const { return r * r + x * x + y * y + z * z; }
32 T len() const { return sqrt(abs2()); }
33 Q conj() const { return Q(-x, -y, -z, r); }
34 Q unit() const { return *this * (1.0 / len()); }
35 Q inv() const { return conj() * (1.0 / abs2()); }
36 friend T dot(Q a, Q b) {
37     return a.x * b.x + a.y * b.y + a.z * b.z;
38 }
39 friend Q cross(Q a, Q b) {
40     return Q(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
41             a.x * b.y - a.y * b.x);
42 }
43 friend Q rotation_around(Q axis, T angle) {
44     return axis.unit() * sin(angle / 2) + cos(angle / 2);
45 }
46 Q rotated_around(Q axis, T angle) {
47     Q u = rotation_around(axis, angle);
48     return u * *this / u;
49 }
50 friend Q rotation_between(Q a, Q b) {
51     a = a.unit(), b = b.unit();
52     if (a == -b) {
53         // degenerate case

```

```

55     Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
56                               : cross(a, Q(0, 1, 0));
57     return rotation_around(ortho, PI);
58 }
59 return (a * (a + b)).conj();
60 };

```

5.1.2. Spherical Coordinates

```

1 struct car_p {
2     double x, y, z;
3 };
4 struct sph_p {
5     double r, theta, phi;
6 };
7 sph_p conv(car_p p) {
8     double r = sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
9     double theta = asin(p.y / r);
10    double phi = atan2(p.y, p.x);
11    return {r, theta, phi};
12 }
13 car_p conv(sph_p p) {
14     double x = p.r * cos(p.theta) * sin(p.phi);
15     double y = p.r * cos(p.theta) * cos(p.phi);
16     double z = p.r * sin(p.theta);
17     return {x, y, z};

```

19| }

5.2. Segments

```

1 //for non-collinear ABCD, if segments AB and CD intersect
2 bool intersects(pt a, pt b, pt c, pt d) {
3     if (cross(b, c, a) * cross(b, d, a) > 0) return false;
4     if (cross(d, a, c) * cross(d, b, c) > 0) return false;
5     return true;
6 }
7 // the intersection point of lines AB and CD
8 pt intersect(pt a, pt b, pt c, pt d) {
9     auto x = cross(b, c, a), y = cross(b, d, a);
10    if (x == y) {
11        // if(abs(x, y) < 1e-8) {
12        // is parallel
13    } else {
14        return d * (x / (x - y)) - c * (y / (x - y));
15    }
16 }
```

5.3. Convex Hull

```

1 // returns a convex hull in counterclockwise order
2 // for a non-strict one, change cross >= to >
3 vector<pt> convex_hull(vector<pt> p) {
4     sort(ALL(p));
5     if (p[0] == p.back()) return {p[0]};
6     int n = p.size(), t = 0;
```

```

7     vector<pt> h(n + 1);
8     for (int _ = 2, s = 0; _--; s = --t, reverse(ALL(p)))
9         for (pt i : p) {
10             while (t > s + 1 && cross(i, h[t - 1], h[t - 2]) >= 0)
11                 t--;
12             h[t++] = i;
13         }
14     return h.resize(t), h;
15 }
```

5.3.1. 3D Hull

```

1
2
3 typedef Point3D<double> P3;
4
5 struct PR {
6     void ins(int x) { (a == -1 ? a : b) = x; }
7     void rem(int x) { (a == x ? a : b) = -1; }
8     int cnt() { return (a != -1) + (b != -1); }
9     int a, b;
10 };
11
12 struct F {
13     P3 q;
14     int a, b, c;
15 };
```

```

17| vector<F> hull3d(const vector<P3> &A) {
18|   assert(sz(A) >= 4);
19|   vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x, y) E[f.x][f.y]
20|   vector<F> FS;
21|   auto mf = [&](int i, int j, int k, int l) {
22|     P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
23|     if (q.dot(A[l]) > q.dot(A[i])) q = q * -1;
24|     F f{q, i, j, k};
25|     E(a, b).ins(k);
26|     E(a, c).ins(j);
27|     E(b, c).ins(i);
28|     FS.push_back(f);
29|   };
30|   rep(i, 0, 4) rep(j, i + 1, 4) rep(k, j + 1, 4)
31|     mf(i, j, k, 6 - i - j - k);
32|
33|   rep(i, 4, sz(A)) {
34|     rep(j, 0, sz(FS)) {
35|       F f = FS[j];
36|       if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
37|         E(a, b).rem(f.c);
38|         E(a, c).rem(f.b);
39|         E(b, c).rem(f.a);
40|         swap(FS[j--], FS.back());
41|         FS.pop_back();
42|       }
43|     }
44|   }
45|   }
46|   int nw = sz(FS);
47|   rep(j, 0, nw) {
48|     F f = FS[j];
#define C(a, b, c)
49|     if (E(a, b).cnt() != 2) mff(f.a, f.b, i, f.c);
50|     C(a, b, c);
51|     C(a, c, b);
52|     C(b, c, a);
53|   }
54| }
55| for (F &it : FS)
56|   if ((A[it.b] - A[it.a])
57|     .cross(A[it.c] - A[it.a])
58|     .dot(it.q) <= 0)
59|     swap(it.c, it.b);
60|   return FS;
61| }

```

5.4. Angular Sort

```

1 auto angle_cmp = [](const pt &a, const pt &b) {
2   auto btm = [](const pt &a) {
3     return a.y < 0 || (a.y == 0 && a.x < 0);
4   };
5   return make_tuple(btm(a), a.y * b.x, abs2(a)) <
6     make_tuple(btm(b), a.x * b.y, abs2(b));
7 }

```

```
9 void angular_sort(vector<pt> &p) {
    sort(p.begin(), p.end(), angle_cmp);
}
```

5.5. Convex Polygon Minkowski Sum

```
1 // O(n) convex polygon minkowski sum
// must be sorted and counterclockwise
3 vector<pt> minkowski_sum(vector<pt> p, vector<pt> q) {
    auto diff = [] (vector<pt> &c) {
        auto rcmp = [] (pt a, pt b) {
            return pt{a.y, a.x} < pt{b.y, b.x};
        };
        rotate(c.begin(), min_element(ALL(c), rcmp), c.end());
        c.push_back(c[0]);
        vector<pt> ret;
        for (int i = 1; i < c.size(); i++)
            ret.push_back(c[i] - c[i - 1]);
        return ret;
    };
    auto dp = diff(p), dq = diff(q);
    pt cur = p[0] + q[0];
    vector<pt> d(dp.size() + dq.size()), ret = {cur};
    // include angle_cmp from angular-sort.cpp
    merge(ALL(dp), ALL(dq), d.begin(), angle_cmp);
    // optional: make ret strictly convex (UB if degenerate)
    int now = 0;
    for (int i = 1; i < d.size(); i++) {
```

```
23     if (cross(d[i], d[now]) == 0) d[now] = d[now] + d[i];
25     else d[++now] = d[i];
26 }
27 d.resize(now + 1);
// end optional part
28 for (pt v : d) ret.push_back(cur = cur + v);
29 return ret.pop_back(), ret;
}
```

5.6. Point In Polygon

```
1 bool on_segment(pt a, pt b, pt p) {
2     return cross(a, b, p) == 0 && dot((p - a), (p - b)) <= 0;
3 }
// p can be any polygon, but this is O(n)
5 bool inside(const vector<pt> &p, pt a) {
    int cnt = 0, n = p.size();
    for (int i = 0; i < n; i++) {
        pt l = p[i], r = p[(i + 1) % n];
        // change to return 0; for strict version
        if (on_segment(l, r, a)) return 1;
        cnt ^= ((a.y < l.y) - (a.y < r.y)) * cross(l, r, a) > 0;
    }
    return cnt;
}
```

5.6.1. Convex Version

```
1 // no preprocessing version
```

```

1 // p must be a strict convex hull, counterclockwise
3 // if point is inside or on border
4 bool is_inside(const vector<pt> &c, pt p) {
5     int n = c.size(), l = 1, r = n - 1;
6     if (cross(c[0], c[1], p) < 0) return false;
7     if (cross(c[n - 1], c[0], p) < 0) return false;
8     while (l < r - 1) {
9         int m = (l + r) / 2;
10        T a = cross(c[0], c[m], p);
11        if (a > 0) l = m;
12        else if (a < 0) r = m;
13        else return dot(c[0] - p, c[m] - p) <= 0;
14    }
15    if (l == r) return dot(c[0] - p, c[l] - p) <= 0;
16    else return cross(c[l], c[r], p) >= 0;
17 }

19 // with preprocessing version
20 vector<pt> vecs;
21 pt center;
22 // p must be a strict convex hull, counterclockwise
23 // BEWARE OF OVERFLOWS!!
24 void preprocess(vector<pt> p) {
25     for (auto &v : p) v = v * 3;
26     center = p[0] + p[1] + p[2];
27     center.x /= 3, center.y /= 3;
28     for (auto &v : p) v = v - center;

29     vecs = (angular_sort(p), p);
30 }
31 bool intersect_strict(pt a, pt b, pt c, pt d) {
32     if (cross(b, c, a) * cross(b, d, a) > 0) return false;
33     if (cross(d, a, c) * cross(d, b, c) >= 0) return false;
34     return true;
35 }
36 // if point is inside or on border
37 bool query(pt p) {
38     p = p * 3 - center;
39     auto pr = upper_bound(ALL(vecs), p, angle_cmp);
40     if (pr == vecs.end()) pr = vecs.begin();
41     auto pl = (pr == vecs.begin()) ? vecs.back() : *(pr - 1);
42     return !intersect_strict({0, 0}, p, pl, *pr);
43 }

```

5.6.2. Offline Multiple Points Version

Requires: GNU PBDS, Point

```

1
3
5
6
7     using Double = __float128;
8     using Point = pt<Double, Double>;
9     int n, m;

```

```

11 vector<Point> poly;
12 vector<Point> query;
13 vector<int> ans;
14
15 struct Segment {
16     Point a, b;
17     int id;
18 };
19 vector<Segment> segs;
20
21 Double Xnow;
22 inline Double get_y(const Segment &u, Double xnow = Xnow) {
23     const Point &a = u.a;
24     const Point &b = u.b;
25     return (a.y * (b.x - xnow) + b.y * (xnow - a.x)) /
26            (b.x - a.x);
27 }
28 bool operator<(Segment u, Segment v) {
29     Double yu = get_y(u);
30     Double yv = get_y(v);
31     if (yu != yv) return yu < yv;
32     return u.id < v.id;
33 }
34 ordered_map<Segment> st;
35
36 struct Event {
37     int type; // +1 insert seg, -1 remove seg, 0 query
38     Double x, y;
39     int id;
40 };
41 bool operator<(Event a, Event b) {
42     if (a.x != b.x) return a.x < b.x;
43     if (a.type != b.type) return a.type < b.type;
44     return a.y < b.y;
45 }
46 vector<Event> events;
47
48 void solve() {
49     set<Double> xs;
50     set<Point> ps;
51     for (int i = 0; i < n; i++) {
52         xs.insert(poly[i].x);
53         ps.insert(poly[i]);
54     }
55     for (int i = 0; i < n; i++) {
56         Segment s{poly[i], poly[(i + 1) % n], i};
57         if (s.a.x > s.b.x ||
58             (s.a.x == s.b.x && s.a.y > s.b.y)) {
59             swap(s.a, s.b);
60         }
61         segs.push_back(s);
62     }
63     if (s.a.x != s.b.x) {
64         events.push_back({+1, s.a.x + 0.2, s.a.y, i});
65     }
66 }

```

```

65     events.push_back({-1, s.b.x - 0.2, s.b.y, i});
66 }
67 for (int i = 0; i < m; i++) {
68     events.push_back({0, query[i].x, query[i].y, i});
69 }
70 sort(events.begin(), events.end());
71 int cnt = 0;
72 for (Event e : events) {
73     int i = e.id;
74     Xnow = e.x;
75     if (e.type == 0) {
76         Double x = e.x;
77         Double y = e.y;
78         Segment tmp = {{x - 1, y}, {x + 1, y}, -1};
79         auto it = st.lower_bound(tmp);
80
81         if (ps.count(query[i]) > 0) {
82             ans[i] = 0;
83         } else if (xs.count(x) > 0) {
84             ans[i] = -2;
85         } else if (it != st.end() &&
86                     get_y(*it) == get_y(tmp)) {
87             ans[i] = 0;
88         } else if (it != st.begin() &&
89                     get_y(*prev(it)) == get_y(tmp)) {
90             ans[i] = 0;
91         } else {
92             int rk = st.order_of_key(tmp);
93             if (rk % 2 == 1) {
94                 ans[i] = 1;
95             } else {
96                 ans[i] = -1;
97             }
98         } else if (e.type == 1) {
99             st.insert(segs[i]);
100            assert((int)st.size() == ++cnt);
101        } else if (e.type == -1) {
102            st.erase(segs[i]);
103            assert((int)st.size() == --cnt);
104        }
105    }
106 }
107 }
```

5.7. Closest Pair

```

1 vector<pll> p; // sort by x first!
2 bool cmpy(const pll &a, const pll &b) const {
3     return a.y < b.y;
4 }
5 ll sq(ll x) { return x * x; }
6 // returns (minimum dist)^2 in [l, r)
7 ll solve(int l, int r) {
8     if (r - l <= 1) return 1e18;
```

```

9 | int m = (l + r) / 2;
10 | ll mid = p[m].x, d = min(solve(l, m), solve(m, r));
11 | auto pb = p.begin();
12 | inplace_merge(pb + l, pb + m, pb + r, cmpy);
13 | vector<P> s;
14 | for (int i = l; i < r; i++)
15 |   if (sq(p[i].x - mid) < d) s.push_back(p[i]);
16 | for (int i = 0; i < s.size(); i++)
17 |   for (int j = i + 1;
18 |     j < s.size() && sq(s[j].y - s[i].y) < d; j++)
19 |     d = min(d, dis(s[i], s[j]));
20 | return d;
21 |

```

5.8. Minimum Enclosing Circle

```

1 |
2 |
3 typedef Point<double> P;
4 double ccRadius(const P &A, const P &B, const P &C) {
5   return (B - A).dist() * (C - B).dist() * (A - C).dist() /
6     abs((B - A).cross(C - A)) / 2;
7 }
8 P ccCenter(const P &A, const P &B, const P &C) {
9   P b = C - A, c = B - A;
10  return A + (b * c.dist2() - c * b.dist2()).perp() /
11    b.cross(c) / 2;
12 }

```

```

13 | pair<P, double> mec(vector<P> ps) {
14 |   shuffle(all(ps), mt19937(time(0)));
15 |   P o = ps[0];
16 |   double r = 0, EPS = 1 + 1e-8;
17 |   rep(i, 0, sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
18 |     o = ps[i], r = 0;
19 |     rep(j, 0, i) if ((o - ps[j]).dist() > r * EPS) {
20 |       o = (ps[i] + ps[j]) / 2;
21 |       r = (o - ps[i]).dist();
22 |       rep(k, 0, j) if ((o - ps[k]).dist() > r * EPS) {
23 |         o = ccCenter(ps[i], ps[j], ps[k]);
24 |         r = (o - ps[i]).dist();
25 |       }
26 |     }
27 |   }
28 |   return {o, r};
29 |

```

6. Strings

6.1. Knuth-Morris-Pratt Algorithm

```

1 vector<int> pi(const string &s) {
2     vector<int> p(s.size());
3     for (int i = 1; i < s.size(); i++) {
4         int g = p[i - 1];
5         while (g && s[i] != s[g]) g = p[g - 1];
6         p[i] = g + (s[i] == s[g]);
7     }
8     return p;
9 }
10 vector<int> match(const string &s, const string &pat) {
11     vector<int> p = pi(pat + '\0' + s), res;
12     for (int i = p.size() - s.size(); i < p.size(); i++)
13         if (p[i] == pat.size())
14             res.push_back(i - 2 * pat.size());
15     return res;
16 }
```

6.2. Z Value

```

1 int z[n];
2 void zval(string s) {
3     // z[i] => longest common prefix of s and s[i:], i > 0
4     int n = s.size();
5     z[0] = 0;
6     for (int b = 0, i = 1; i < n; i++) {
```

```

7     if (z[b] + b <= i) z[i] = 0;
8     else z[i] = min(z[i - b], z[b] + b - i);
9     while (s[i + z[i]] == s[z[i]]) z[i]++;
10    if (i + z[i] > b + z[b]) b = i;
11 }
```

6.3. Manacher's Algorithm

```

1 int z[n];
2 void manacher(string s) {
3     // z[i] => longest odd palindrome centered at i is
4     //      s[i - z[i] ... i + z[i]]
5     // to get all palindromes (including even length),
6     // insert a '#' between each s[i] and s[i + 1]
7     int n = s.size();
8     z[0] = 0;
9     for (int b = 0, i = 1; i < n; i++) {
10        if (z[b] + b >= i)
11            z[i] = min(z[2 * b - i], b + z[b] - i);
12        else z[i] = 0;
13        while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
14              s[i + z[i] + 1] == s[i - z[i] - 1])
15            z[i]++;
16        if (z[i] + i > z[b] + b) b = i;
17    }
```

6.4. Minimum Rotation

```

1 int min_rotation(string s) {
2     int a = 0, n = s.size();
3     s += s;
4     for (int b = 0; b < n; b++) {
5         for (int k = 0; k < n; k++) {
6             if (a + k == b || s[a + k] < s[b + k]) {
7                 b += max(0, k - 1);
8                 break;
9             }
10            if (s[a + k] > s[b + k]) {
11                a = b;
12                break;
13            }
14        }
15    }
16    return a;
17 }
```

6.5. Aho-Corasick Automaton

```

1 struct Aho_Corasick {
2     static const int maxc = 26, maxn = 4e5;
3     struct NODES {
4         int Next[maxc], fail, ans;
5     };
6     NODES T[maxn];
7     int top, qtop, q[maxn];
8     int get_node(const int &fail) {
9         fill_n(T[top].Next, maxc, 0);
10        T[top].fail = fail;
11        T[top].ans = 0;
12        return top++;
13    }
14    int insert(const string &s) {
15        int ptr = 1;
16        for (char c : s) { // change char id
17            c -= 'a';
18            if (!T[ptr].Next[c]) T[ptr].Next[c] = get_node(ptr);
19            ptr = T[ptr].Next[c];
20        }
21        return ptr;
22    } // return ans_last_place
23    void build_fail(int ptr) {
24        int tmp;
25        for (int i = 0; i < maxc; i++) {
26            if (T[ptr].Next[i]) {
```

```

27    tmp = T[ptr].fail;
28    while (tmp != 1 && !T[tmp].Next[i])
29        tmp = T[tmp].fail;
30        if (T[tmp].Next[i] != T[ptr].Next[i])
31            if (T[tmp].Next[i]) tmp = T[tmp].Next[i];
32            T[T[ptr].Next[i]].fail = tmp;
33            q[qtop++] = T[ptr].Next[i];
34        }
35    }
36
37    void AC_auto(const string &s) {
38        int ptr = 1;
39        for (char c : s) {
40            while (ptr != 1 && !T[ptr].Next[c]) ptr = T[ptr].fail;
41            if (T[ptr].Next[c]) {
42                ptr = T[ptr].Next[c];
43                T[ptr].ans++;
44            }
45        }
46
47        void Solve(string &s) {
48            for (char &c : s) // change char id
49                c -= 'a';
50            for (int i = 0; i < qtop; i++) build_fail(q[i]);
51            AC_auto(s);
52            for (int i = qtop - 1; i > -1; i--)
53                T[T[q[i]].fail].ans += T[q[i]].ans;
54        }

```

```

55    void reset() {
56        qtop = top = q[0] = 1;
57        get_node(1);
58    }
59 } AC;
60 // usage example
61 string s, S;
62 int n, t, ans_place[50000];
63 int main() {
64     Tie cin >> t;
65     while (t--) {
66         AC.reset();
67         cin >> S >> n;
68         for (int i = 0; i < n; i++) {
69             cin >> s;
70             ans_place[i] = AC.insert(s);
71         }
72         AC.Solve(S);
73         for (int i = 0; i < n; i++)
74             cout << AC.T[ans_place[i]].ans << '\n';
75     }

```

7. Debug List

- 1 - Pre-submit:
 - Did you make a typo when copying a template?
 - Test more cases if unsure.
 - Write a naive solution and check small cases.
 - Submit the correct file.

- 7 - General Debugging:
 - Read the whole problem again.
 - Have a teammate read the problem.
 - Have a teammate read your code.
 - Explain your solution to them (or a rubber duck).
 - Print the code and its output / debug output.
 - Go to the toilet.

- 15 - Wrong Answer:
 - Any possible overflows?
 - > `__int128` ?
 - Try `"-ftrapv"` or `#pragma GCC optimize("trapv")`
 - Floating point errors?
 - > `long double` ?
 - turn off math optimizations
 - check for `==`, `>=`, `acos(1.000000001)`, etc.
 - Did you forget to sort or unique?
 - Generate large and worst "corner" cases.
 - Check your `m` / `n`, `i` / `j` and `x` / `y`.
 - Are everything initialized or reset properly?

- | | |
|----|---|
| 27 | <ul style="list-style-type: none"> - Are you sure about the STL thing you are using? - Read cppreference (should be available). - Print everything and run it on pen and paper. |
| 29 | <ul style="list-style-type: none"> - Time Limit Exceeded: <ul style="list-style-type: none"> - Calculate your time complexity again. - Does the program actually end? <ul style="list-style-type: none"> - Check for `while(q.size())` etc. - Test the largest cases locally. - Did you do unnecessary stuff? <ul style="list-style-type: none"> - e.g. pass vectors by value - e.g. `memset` for every test case - Is your constant factor reasonable? |
| 31 | <ul style="list-style-type: none"> - Runtime Error: <ul style="list-style-type: none"> - Check memory usage. <ul style="list-style-type: none"> - Forget to clear or destroy stuff? <ul style="list-style-type: none"> -> `vector::shrink_to_fit()` - Stack overflow? - Bad pointer / array access? <ul style="list-style-type: none"> - Try `"-fsanitize=address` - Division by zero? NaN's? |
| 33 | <ul style="list-style-type: none"> - |
| 35 | <ul style="list-style-type: none"> - |
| 37 | <ul style="list-style-type: none"> - |
| 39 | <ul style="list-style-type: none"> - |
| 41 | <ul style="list-style-type: none"> - |
| 43 | <ul style="list-style-type: none"> - |
| 45 | <ul style="list-style-type: none"> - |
| 47 | <ul style="list-style-type: none"> - |