

University of Tartu ICPC Team Notebook

(2017-2018) October 10, 2018

Contents

- 1 Setup
- 2 crc.sh
- 3 gcc ordered set
- 4 Numerical integration with Simpson's rule
- 5 Triangle centers
- 6 2D line segment
- 7 Convex polygon algorithms
- 8 Aho Corasick $\mathcal{O}(|\alpha| \sum \text{len})$
- 9 Suffix automaton $\mathcal{O}((n+q) \log(|\alpha|))$
- 10 Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$
- 11 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$
- 12 DMST $\mathcal{O}(E \log V)$
- 13 Bridges $\mathcal{O}(n)$
- 14 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$
- 15 Generic persistent compressed lazy segment tree
- 16 Templated HLD $\mathcal{O}(M(n) \log n)$ per query
- 17 Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query
- 18 Treap $\mathcal{O}(\log n)$ per query
- 19 FFT 5M length/sec
- 20 Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$

1 Setup

```
1 set smartindent cindent
2 set ts=4 sw=4 expandtab
3 syntax enable
4 set clipboard=unnamedplus
5 "colorscheme elflord
6 "setxkbmap -option caps:escape
7 "setxkbmap -option
8 "valgrind --vgdb-error=0 ./a <inp &
9 "gdb a
10 "target remote | vgdb
```

2 crc.sh

```
1 #!/bin/envbash
2 starts=( $(sed '/^\s*$/d' $1 | grep -n "/\!/start" | cut -f1 -d:) )
3 finishes=( $(sed '/^\s*$/d' $1 | grep -n "/\!/finish" | cut -f1 -d:) )
4 for ((i=0; i<${#starts[@]}; i++)); do
5     for j in `seq 10 10 ${finishes[i]-starts[i]+8}`; do
6         sed '/^\s*$/d' $1 | head -${finishes[i]-1} | tail
        ↪ -${finishes[i]-starts[i]-1} | \
7         head -$j | tr -d '[:space:]' | cksum | cut -f1 -d ' ' | tail -c
        ↪ 4
8     done #whistespaces don't matter
9     echo #there shouldn't be any comments in the checked range
10 done #check last number in each block
```

3 gcc ordered set

```
1 #include <bits/stdc++.h>
2 typedef long long ll;
3 using namespace std;
4 #include <ext/pb_ds/assoc_container.hpp>
5 #include <ext/pb_ds/tree_policy.hpp>
6 using namespace __gnu_pbds;
7 template <typename T>
8 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
9     ↪ tree_order_statistics_node_update>;
10 int main(){
11     ordered_set<int> cur;
12     cur.insert(1);
13     cur.insert(3);
14     cout << cur.order_of_key(2) << endl; // the number of elements in the
15     ↪ set less than 2
16     cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in
17     ↪ the set(0-based)
18     cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in
19     ↪ the set(0-based)
20 } %574
```

4 Numerical integration with Simpson's rule

```
1 //computing power = how many times function integrate gets called
2 template<typename T>
3 double simps(T f, double a, double b) {
4     return (f(a) + 4*f((a+b)/2) + f(b))*(b-a)/6;
5 }
6 template<typename T>
```

```

7 double integrate(T f, double a, double b, double computing_power){
8   double m = (a+b)/2;
9   double l = simps(f,a,m), r = simps(f,m,b), tot=simps(f,a,b);
10  if (computing_power < 1) return tot;
11  return integrate(f, a, m, computing_power/2) + integrate(f, m, b,
12    ↪ computing_power/2);                                     #430
                                                                %360

```

5 Triangle centers

```

1 const double min_delta = 1e-13;
2 const double coord_max = 1e6;
3 typedef complex < double > point;
4 point A, B, C; // vertices of the triangle
5 bool collinear(){
6   double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7   if(min_diff < coord_max * min_delta)
8     return true;
9   point sp = (B - A) / (C - A);
10  double ang = M_PI/2-abs(abs(arg(sp))-M_PI/2); //positive angle with
11    ↪ the real line                                     #623
12  return ang < min_delta;                                     %446
13 }
14 point circum_center(){
15   if(collinear())
16     return point(NAN,NAN);
17   //squared lengths of sides
18   double a2, b2, c2;
19   a2 = norm(B - C);
20   b2 = norm(A - C);
21   c2 = norm(A - B);
22   //barycentric coordinates of the circumcenter
23   double c_A, c_B, c_C;
24   c_A = a2 * (b2 + c2 - a2); //sin(2 * alpha) may be used as well
25   c_B = b2 * (a2 + c2 - b2);                                     #385
26   c_C = c2 * (a2 + b2 - c2);
27   double sum = c_A + c_B + c_C;
28   c_A /= sum;
29   c_B /= sum;
30   c_C /= sum;
31   // cartesian coordinates of the circumcenter
32   return c_A * A + c_B * B + c_C * C;                                     %742
33 }
34 point centroid(){ //center of mass
35   return (A + B + C) / 3.0;
36 }
37 point ortho_center(){ //euler line
38   point O = circum_center();
39   return O + 3.0 * (centroid() - O);
40 }
41 point nine_point_circle_center(){ //euler line
42   point O = circum_center();
43   return O + 1.5 * (centroid() - O);                                     #193
44 }
45 point in_center(){

```

```

45 if(collinear())
46   return point(NAN,NAN);
47 double a, b, c; //side lengths
48 a = abs(B - C);
49 b = abs(A - C);
50 c = abs(A - B);
51 //trilinear coordinates are (1,1,1)
52 //barycentric coordinates
53 double c_A = a, c_B = b, c_C = c;
54 double sum = c_A + c_B + c_C;
55 c_A /= sum;
56 c_B /= sum;
57 c_C /= sum;
58 // cartesian coordinates of the incenter
59 return c_A * A + c_B * B + c_C * C;                                     #157
60 }
                                                                %980

```

6 2D line segment

```

1 const long double PI = acos(-1.0L);
2 struct Vec {
3   long double x, y;
4   Vec& operator--=(Vec r) {
5     x -= r.x, y -= r.y;
6     return *this;
7   }
8   Vec operator-(Vec r) {return Vec(*this) -= r;}
9   Vec& operator+=(Vec r) {
10     x += r.x, y += r.y;
11     return *this;
12   }
13   Vec operator+(Vec r) {return Vec(*this) += r;}
14   Vec operator-() {return {-x, -y};}
15   Vec& operator*=(long double r) {
16     x *= r, y *= r;
17     return *this;
18   }
19   Vec operator*(long double r) {return Vec(*this) *= r;}
20   Vec& operator/=(long double r) {
21     x /= r, y /= r;
22     return *this;
23   }
24   Vec operator/(long double r) {return Vec(*this) /= r;}
25   long double operator*(Vec r) {
26     return x * r.x + y * r.y;
27   }
28 };
29 ostream& operator<<(ostream& l, Vec r) {
30   return l << '(' << r.x << ", " << r.y << ')';
31 }
32 long double len(Vec a) {
33   return hypot(a.x, a.y);
34 }
35 long double cross(Vec l, Vec r) {
36   return l.x * r.y - l.y * r.x;

```

```

37 }
38 long double angle(Vec a) {
39     return fmod(atan2(a.y, a.x)+2*PI, 2*PI);
40 } #872
41 Vec normal(Vec a) {
42     return Vec({-a.y, a.x}) / len(a);
43 } %654

```

```

1 struct Segment {
2     Vec a, b;
3     Vec d() {
4         return b-a;
5     }
6 };
7 ostream& operator<<(ostream& l, Segment r) {
8     return l << r.a << '-' << r.b;
9 }
10 Vec intersection(Segment l, Segment r) { #355
11     Vec dl = l.d(), dr = r.d();
12     if(cross(dl, dr) == 0)
13         return {nanl(""), nanl("")};
14     long double h = cross(dr, l.a-r.a) / len(dr);
15     long double dh = cross(dr, dl) / len(dr);
16     return l.a + dl * (h / -dh);
17 }
18 //Returns the area bounded by halfplanes
19 long double getArea(vector<Segment> lines) {
20     long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
21     vector<Segment> linesBySide[2];
22     for(auto line : lines) {
23         if(line.b.y == line.a.y) {
24             if(line.a.x < line.b.x) {
25                 lowerbound = max(lowerbound, line.a.y);
26             } else {
27                 upperbound = min(upperbound, line.a.y);
28             }
29         } else if(line.a.y < line.b.y) {
30             linesBySide[1].push_back(line);
31         } else {
32             linesBySide[0].push_back({line.b, line.a});
33         }
34     }
35     sort(linesBySide[0].begin(), linesBySide[0].end(), [](Segment l,
36         ↪ Segment r) {
37         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a >
38             ↪ normal(r.d())*r.a;
39         return cross(l.d(), r.d()) < 0;
40     });
41     sort(linesBySide[1].begin(), linesBySide[1].end(), [](Segment l,
42         ↪ Segment r) {
43         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a <
44             ↪ normal(r.d())*r.a;
45         return cross(l.d(), r.d()) > 0;
46     });

```

#434

```

43 //Now find the application area of the lines and clean up redundant
44 ↪ ones
45 vector<long double> applyStart[2];
46 for(int side = 0; side < 2; side++) {
47     vector<long double> &apply = applyStart[side];
48     vector<Segment> curLines;
49     for(auto line : linesBySide[side]) {
50         while(curLines.size() > 0) {
51             Segment other = curLines.back();
52             if(cross(line.d(), other.d()) != 0) {
53                 long double start = intersection(line, other).y; #501
54                 if(start > apply.back()) break;
55             }
56             curLines.pop_back();
57             apply.pop_back();
58         }
59         if(curLines.size() == 0) {
60             apply.push_back(-HUGE_VALL);
61         } else {
62             apply.push_back(intersection(line, curLines.back()).y);
63         }
64         curLines.push_back(line);
65     }
66     linesBySide[side] = curLines;
67 }
68 applyStart[0].push_back(HUGE_VALL);
69 applyStart[1].push_back(HUGE_VALL);
70 long double result = 0;
71 {
72     long double lb = -HUGE_VALL, ub;
73     for(int i=0, j=0; i < (int)linesBySide[0].size() && j <
74         ↪ (int)linesBySide[1].size(); lb = ub) { #349
75         ub = min(applyStart[0][i+1], applyStart[1][j+1]);
76         long double alb = lb, aub = ub;
77         Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];
78         if(cross(l1.d(), l0.d()) > 0) {
79             alb = max(alb, intersection(l0, l1).y);
80         } else if(cross(l1.d(), l0.d()) < 0) {
81             aub = min(aub, intersection(l0, l1).y);
82         }
83         alb = max(alb, lowerbound);
84         aub = min(aub, upperbound); #419
85         result += (aub - alb) * (x1 + x2) / 2;
86     }
87 }
88 {
89     long double x1 = 11.a.x + (alb - 11.a.y) / 11.d().y * 11.d().x;
90     long double x2 = 11.a.x + (aub - 11.a.y) / 11.d().y * 11.d().x;
91     result += (aub - alb) * (x1 + x2) / 2; #228
92 }
93 }

```

```

94     if(applyStart[0][i+1] < applyStart[1][j+1]) {
95         i++;
96     } else {
97         j++;
98     }
99 }
100 }
101 return result;
102 }

```

%011

7 Convex polygon algorithms

```

1 ll dot(const pair< int, int > &v1, const pair< int, int > &v2) {
2     return (ll)v1.first * v2.first + (ll)v1.second * v2.second;
3 }
4 ll cross(const pair< int, int > &v1, const pair< int, int > &v2) {
5     return (ll)v1.first * v2.second - (ll)v2.first * v1.second;
6 }
7 ll dist_sq(const pair< int, int > &p1, const pair< int, int > &p2) {
8     return (ll)(p2.first - p1.first) * (p2.first - p1.first) +
9         (ll)(p2.second - p1.second) * (p2.second - p1.second);
10 }
11 struct Hull {
12     vector< pair< pair< int, int >, pair< int, int > > > hull;
13     vector< pair< pair< int, int >, pair< int, int > > >::iterator
14     ↪ upper_begin;
15     template < typename Iterator >
16     void extend_hull(Iterator begin, Iterator end) { // 0(n)
17         vector< pair< int, int > > res;
18         for (auto it = begin; it != end; ++it) {
19             if (res.empty() || *it != res.back()) {
20                 while (res.size() >= 2) {
21                     auto v1 = make_pair(res[res.size() - 1].first -
22                                         ↪ res[res.size() - 2].first,
23                                         ↪ res[res.size() - 1].second -
24                                         ↪ res[res.size() - 2].second);
25                     auto v2 = make_pair(it->first - res[res.size() - 2].first,
26                                         ↪ it->second - res[res.size() - 2].second);
27                     if (cross(v1, v2) > 0)
28                         break;
29                     res.pop_back();
30                 }
31                 res.push_back(*it);
32             }
33         }
34         for (int i = 0; i < res.size() - 1; ++i)
35             hull.emplace_back(res[i], res[i + 1]);
36     }
37     Hull(vector< pair< int, int > > &vert) { // at least 2 distinct
38         ↪ points
39         sort(vert.begin(), vert.end()); // 0(n log(n))
40         extend_hull(vert.begin(), vert.end());
41         int diff = hull.size();
42         extend_hull(vert.rbegin(), vert.rend());
43         upper_begin = hull.begin() + diff;

```

#048

#901

```

40 }
41 bool contains(pair< int, int > p) { // 0(log(n))
42     if (p < hull.front().first || p > upper_begin->first) return false;
43 }
44     auto it_low = lower_bound(hull.begin(), upper_begin,
45                               ↪ make_pair(make_pair(p.first,
46                               ↪ (int)-2e9), make_pair(0, 0)));
47     if (it_low != hull.begin())
48         --it_low;
49     auto v1 = make_pair(it_low->second.first - it_low->first.first,
50                               ↪ it_low->second.second -
51                               ↪ it_low->first.second);
52     auto v2 = make_pair(p.first - it_low->first.first, p.second -
53                               ↪ it_low->first.second);
54     if (cross(v1, v2) < 0) // < 0 is inclusive, <= 0 is exclusive
55         return false;
56 }
57     auto it_up = lower_bound(hull.rbegin(), hull.rbegin() +
58                               ↪ (hull.end() - upper_begin),
59                               ↪ make_pair(make_pair(p.first, (int)2e9),
60                               ↪ make_pair(0, 0)));
61     if (it_up - hull.rbegin() == hull.end() - upper_begin)
62         --it_up;
63     auto v1 = make_pair(it_up->first.first - it_up->second.first,
64                               ↪ it_up->first.second - it_up->second.second);
65     auto v2 = make_pair(p.first - it_up->second.first, p.second -
66                               ↪ it_up->second.second);
67     if (cross(v1, v2) > 0) // > 0 is inclusive, >= 0 is exclusive
68         return false;
69 }
70     return true;
71 }
72 template < typename T > // The function can have only one local min
73     ↪ and max and may be constant
74     // only at min and max.
75     vector< pair< pair< int, int >, pair< int, int > > >::iterator max(
76     ↪ function< T(const pair< int, int >, pair< int, int > > &) >
77     ↪ f) { // 0(log(n))
78     auto l = hull.begin();
79     auto r = hull.end();
80     vector< pair< pair< int, int >, pair< int, int > > >::iterator best
81     ↪ = hull.end();
82     T best_val;
83     while (r - l > 2) {
84         auto mid = l + (r - l) / 2;
85         T l_val = f(*l);
86         T l_nxt_val = f(*(l + 1));
87         T mid_val = f(*mid);
88         T mid_nxt_val = f(*(mid + 1));
89         if (best == hull.end() ||

```

%873

#094

%092

#242

```

82     l_val > best_val) { // If max is at l we may remove it from
83         ↪ the range.
84     best = l;
85     best_val = l_val;
86 }
87 if (l_nxt_val > l_val) {
88     if (mid_val < l_val) {
89         r = mid;
90     } else {
91         if (mid_nxt_val > mid_val) {
92             l = mid + 1;
93         } else {
94             r = mid + 1;
95         }
96     }
97 } else {
98     if (mid_val < l_val) {
99         l = mid + 1;
100     } else {
101         if (mid_nxt_val > mid_val) {
102             l = mid + 1;
103         } else {
104             r = mid + 1;
105         }
106     }
107 }
108 T l_val = f(*l);
109 if (best == hull.end() || l_val > best_val) {
110     best = l;
111     best_val = l_val;
112 }
113 if (r - l > 1) {
114     T l_nxt_val = f(*(l + 1));
115     if (best == hull.end() || l_nxt_val > best_val) {
116         best = l + 1;
117         best_val = l_nxt_val;
118     }
119 }
120 return best;
121 }
122 vector< pair< pair< int, int >, pair< int, int > >::iterator
123     ↪ closest(
124     pair< int, int >
125     p) { // p can't be internal(can be on border), hull must
126         ↪ have atleast 3 points
127     const pair< pair< int, int >, pair< int, int > > &ref_p =
128         ↪ hull.front(); // O(log(n))
129     return max(function< double(const pair< pair< int, int >, pair<
130         ↪ int, int > > &) >(
131         [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
132             &seg) { // accuracy of used type should be
133                 coord^2

```

#012

#373

#332

#930

%331

```

129     if (p == seg.first) return 10 - M_PI;
130     auto v1 =
131         make_pair(seg.second.first - seg.first.first,
132             ↪ seg.second.second - seg.first.second);
133     auto v2 = make_pair(p.first - seg.first.first, p.second -
134         ↪ seg.first.second);
135     ll cross_prod = cross(v1, v2);
136     if (cross_prod > 0) { // order the backside by angle
137         auto v1 = make_pair(ref_p.first.first - p.first,
138             ↪ ref_p.first.second - p.second);
139         auto v2 = make_pair(seg.first.first - p.first,
140             ↪ seg.first.second - p.second);
141         ll dot_prod = dot(v1, v2);
142         ll cross_prod = cross(v2, v1);
143         return atan2(cross_prod, dot_prod) / 2;
144     }
145     ll dot_prod = dot(v1, v2);
146     double res = atan2(dot_prod, cross_prod);
147     if (dot_prod <= 0 && res > 0) res = -M_PI;
148     if (res > 0) {
149         res += 20;
150     } else {
151         res = 10 - res;
152     }
153     return res;
154 }
155 }
156 pair< int, int > forw_tan(pair< int, int > p) { // can't be internal
157     ↪ or on border
158     const pair< pair< int, int >, pair< int, int > > &ref_p =
159         ↪ hull.front(); // O(log(n))
160     auto best_seg = max(function< double(const pair< pair< int, int >,
161         ↪ pair< int, int > > &) >(
162         [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
163             &seg) { // accuracy of used type should be
164                 coord^2
165         auto v1 = make_pair(ref_p.first.first - p.first,
166             ↪ ref_p.first.second - p.second);
167         auto v2 = make_pair(seg.first.first - p.first,
168             ↪ seg.first.second - p.second);
169         ll dot_prod = dot(v1, v2);
170         ll cross_prod = cross(v2, v1); // cross(v1, v2) for
171             ↪ back_tan!!!
172         return atan2(cross_prod, dot_prod); // order by signed
173             ↪ angle
174     }));
175     return best_seg->first;
176 }
177 vector< pair< pair< int, int >, pair< int, int > >::iterator
178     ↪ max_in_dir(
179     pair< int, int > v) { // first is the ans. O(log(n))
180     return max(function< ll(const pair< pair< int, int >, pair< int,
181         ↪ int > > &) >(

```

#685

#395

%483

#291

%850


```

168     [&v](const pair< pair< int, int >, pair< int, int > > &seg) {
169         ↪ return dot(v, seg.first); }));
170 }
171 pair< vector< pair< pair< int, int >, pair< int, int > >::iterator,
172     vector< pair< pair< int, int >, pair< int, int > >::iterator
173     ↪ > %013
174 intersections(pair< pair< int, int >, pair< int, int > > line) { //
175     ↪ 0(log(n))
176     int x = line.second.first - line.first.first;
177     int y = line.second.second - line.first.second;
178     auto dir = make_pair(-y, x);
179     auto it_max = max_in_dir(dir);
180     auto it_min = max_in_dir(make_pair(y, -x));
181     ll opt_val = dot(dir, line.first);
182     if (dot(dir, it_max->first) < opt_val || dot(dir, it_min->first) >
183     ↪ opt_val)
184     return make_pair(hull.end(), hull.end());
185     vector< pair< pair< int, int >, pair< int, int > >::iterator
186     ↪ it_r1, it_r2; #785
187     function< bool(const pair< pair< int, int >, pair< int, int > > &,
188     ↪ const pair< pair< int, int >, pair< int, int > > &)
189     ↪ >
190     inc_comp([&dir](const pair< pair< int, int >, pair< int, int >
191     ↪ > &lft,
192     ↪ const pair< pair< int, int >, pair< int, int >
193     ↪ > &rgt) {
194     ↪ return dot(dir, lft.first) < dot(dir, rgt.first);
195     });
196     function< bool(const pair< pair< int, int >, pair< int, int > > &,
197     ↪ const pair< pair< int, int >, pair< int, int > > &)
198     ↪ >
199     dec_comp([&dir](const pair< pair< int, int >, pair< int, int >
200     ↪ > &lft,
201     ↪ const pair< pair< int, int >, pair< int, int >
202     ↪ > &rgt) { #979
203     ↪ return dot(dir, lft.first) > dot(dir, rgt.first);
204     });
205     if (it_min <= it_max) {
206     it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
207     if (dot(dir, hull.front().first) >= opt_val) {
208     it_r2 = upper_bound(hull.begin(), it_min + 1, line, dec_comp) -
209     ↪ 1;
210     } else {
211     it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
212     }
213     } else { #684
214     it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
215     if (dot(dir, hull.front().first) <= opt_val) {
216     it_r2 = upper_bound(hull.begin(), it_max + 1, line, inc_comp) -
217     ↪ 1;
218     } else {
219     it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
220     }
221     }

```

```

208     }
209     return make_pair(it_r1, it_r2);
210 }
211 pair< pair< int, int >, pair< int, int > > diameter() { // 0(n) %000
212     pair< pair< int, int >, pair< int, int > > res;
213     ll dia_sq = 0;
214     auto it1 = hull.begin();
215     auto it2 = upper_begin;
216     auto v1 = make_pair(hull.back().second.first -
217     ↪ hull.back().first.first,
218     ↪ hull.back().second.second -
219     ↪ hull.back().first.second);
220     while (it2 != hull.begin()) {
221     auto v2 = make_pair((it2 - 1)->second.first - (it2 -
222     ↪ 1)->first.first,
223     ↪ (it2 - 1)->second.second - (it2 -
224     ↪ 1)->first.second); #671
225     ll decider = cross(v1, v2);
226     if (decider > 0) break;
227     --it2;
228     }
229     while (it2 != hull.end()) { // check all antipodal pairs
230     if (dist_sq(it1->first, it2->first) > dia_sq) {
231     res = make_pair(it1->first, it2->first);
232     dia_sq = dist_sq(res.first, res.second);
233     }
234     auto v1 = #674
235     ↪ make_pair(it1->second.first - it1->first.first,
236     ↪ it1->second.second - it1->first.second);
237     auto v2 =
238     ↪ make_pair(it2->second.first - it2->first.first,
239     ↪ it2->second.second - it2->first.second);
240     ll decider = cross(v1, v2);
241     if (decider == 0) { // report cross pairs at parallel lines.
242     if (dist_sq(it1->second, it2->first) > dia_sq) {
243     res = make_pair(it1->second, it2->first);
244     dia_sq = dist_sq(res.first, res.second);
245     }
246     if (dist_sq(it1->first, it2->second) > dia_sq) { #466
247     res = make_pair(it1->first, it2->second);
248     dia_sq = dist_sq(res.first, res.second);
249     }
250     ++it1;
251     ++it2;
252     } else if (decider < 0) {
253     ++it1;
254     } else {
255     ++it2;
256     }
257     }
258     return res;
259 }
260 };

```

%000

#671

#674

#466

#502

%215

8 Aho Corasick $\mathcal{O}(|\alpha| \sum \text{len})$

```

1 const int alpha_size=26;
2 struct node{
3     node *nxt[alpha_size]; //May use other structures to move in trie
4     node *suffix;
5     node(){
6         memset(nxt, 0, alpha_size*sizeof(node *));
7     }
8     int cnt=0;
9 };
10 node *aho_corasick(vector<vector<char> > &dict){ #480
11     node *root= new node;
12     root->suffix = 0;
13     vector<pair<vector<char> *, node *> > cur_state;
14     for(vector<char> &s : dict)
15         cur_state.emplace_back(&s, root);
16     for(int i=0; !cur_state.empty(); ++i){
17         vector<pair<vector<char> *, node *> > nxt_state;
18         for(auto &cur : cur_state){
19             node *nxt=cur.second->nxt[(cur.first)[i]];
20             if(nxt){ #888
21                 cur.second=nxt;
22             }else{
23                 nxt = new node;
24                 cur.second->nxt[(cur.first)[i]] = nxt;
25                 node *suf = cur.second->suffix;
26                 cur.second = nxt;
27                 nxt->suffix = root; //set correct suffix link
28                 while(suf){
29                     if(suf->nxt[(cur.first)[i]]){
30                         nxt->suffix = suf->nxt[(cur.first)[i]]; #786
31                         break;
32                     }
33                     suf=suf->suffix;
34                 }
35             }
36             if(cur.first->size() > i+1)
37                 nxt_state.push_back(cur);
38         }
39         cur_state=nxt_state; #940
40     }
41     return root; #064
42 }
43 //auxiliary functions for searching and counting
44 node *walk(node *cur, char c){ //longest prefix in dict that is suffix
45     ↪ of walked string.
46     while(true){
47         if(cur->nxt[c])
48             return cur->nxt[c];
49         if(!cur->suffix)
50             return cur;
51         cur = cur->suffix;
52     }
53 }

```

```

52 } %127
53 void cnt_matches(node *root, vector<char> &match_in){
54     node *cur = root;
55     for(char c : match_in){
56         cur = walk(cur, c);
57         ++cur->cnt;
58     }
59 } %286
60 void add_cnt(node *root){ //After counting matches propagete ONCE to
61     ↪ suffixes for final counts
62     vector<node *> to_visit = {root};
63     for(int i=0; i<to_visit.size(); ++i){
64         node *cur = to_visit[i];
65         for(int j=0; j<alpha_size; ++j){
66             if(cur->nxt[j])
67                 to_visit.push_back(cur->nxt[j]);
68         }
69     }
70     for(int i=to_visit.size()-1; i>0; --i) #865
71         to_visit[i]->suffix->cnt += to_visit[i]->cnt; %313
72 int main(){
73     ↪ //http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
74     int n, len;
75     scanf("%d %d", &n, &len);
76     vector<char> a(len+1);
77     scanf("%s", a.data());
78     a.pop_back();
79     for(char &c : a)
80         c -= 'a';
81     vector<vector<char> > dict(n);
82     for(int i=0; i<n; ++i){
83         scanf("%d", &len);
84         dict[i].resize(len+1);
85         scanf("%s", dict[i].data());
86         dict[i].pop_back();
87         for(char &c : dict[i])
88             c -= 'a';
89     }
90     node *root = aho_corasick(dict);
91     cnt_matches(root, a);
92     add_cnt(root);
93     for(int i=0; i<n; ++i){
94         node *cur = root;
95         for(char c : dict[i])
96             cur = walk(cur, c);
97         printf("%d\n", cur->cnt);
98     }
99 }

```

9 Suffix automaton $\mathcal{O}((n+q) \log(|\alpha|))$

```

1 class AutoNode {
2 private:

```

```

3  map< char, AutoNode * > nxt_char; // Map is faster than hashtable
   ↳ and unsorted arrays
4  public:
5  int len; //Length of longest suffix in equivalence class.
6  AutoNode *suf;
7  bool has_nxt(char c) const {
8      return nxt_char.count(c);
9  }
10 AutoNode *nxt(char c) { #486
11     if (!has_nxt(c))
12         return NULL;
13     return nxt_char[c];
14 }
15 void set_nxt(char c, AutoNode *node) {
16     nxt_char[c] = node;
17 }
18 AutoNode *split(int new_len, char c) {
19     AutoNode *new_n = new AutoNode;
20     new_n->nxt_char = nxt_char;
21     new_n->len = new_len;
22     new_n->suf = suf;
23     suf = new_n;
24     return new_n;
25 }
26 // Extra functions for matching and counting
27 AutoNode *lower_depth(int depth) { //move to longest suffix of
   ↳ current with a maximum length of depth.
28     if (suf->len >= depth)
29         return suf->lower_depth(depth);
30     return this;
31 }
32 AutoNode *walk(char c, int depth, int &match_len) { //move to longest
   ↳ suffix of walked path that is a substring
33     match_len = min(match_len, len); //includes depth limit(needed for
   ↳ finding matches)
34     if (has_nxt(c)) { //as suffixes are in classes match_len must be
   ↳ tracked externally
35         ++match_len;
36         return nxt(c)->lower_depth(depth);
37     }
38     if (suf)
39         return suf->walk(c, depth, match_len);
40     return this;
41 }
42 int paths_to_end = 0;
43 void set_as_end() { //All suffixes of current node are marked as
   ↳ ending nodes.
44     paths_to_end = 1;
45     if (suf) suf->set_as_end();
46 }
47 bool vis = false;
48 void calc_paths_to_end() { //Call ONCE from ROOT. For each node
   ↳ calculates number of ways to reach an end node.

```

```

49     if (!vis) { //paths_to_end is ocurence count for any strings in
   ↳ current suffix equivalence class.
50         vis = true;
51         for (auto cur : nxt_char) { #738
52             cur.second->calc_paths_to_end();
53             paths_to_end += cur.second->paths_to_end;
54         }
55     }
56 }
57 };
58 struct SufAutomaton {
59     AutoNode *last;
60     AutoNode *root;
61     void extend(char new_c) { #885
62         AutoNode *new_end = new AutoNode;
63         new_end->len = last->len + 1;
64         AutoNode *suf_w_nxt = last;
65         while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {
66             suf_w_nxt->set_nxt(new_c, new_end);
67             suf_w_nxt = suf_w_nxt->suf;
68         }
69         if (!suf_w_nxt) {
70             new_end->suf = root;
71         } else { #873
72             AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);
73             if (suf_w_nxt->len + 1 == max_sbstr->len) {
74                 new_end->suf = max_sbstr;
75             } else {
76                 AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1,
77                     ↳ new_c);
78                 new_end->suf = eq_sbstr;
79                 AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
80                 while (w_edge_to_eq_sbstr != 0 &&
81                     ↳ w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
82                     w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
83                     w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
84                 }
85             }
86         }
87         last = new_end;
88     }
89     SufAutomaton(string to_suffix) {
90         root = new AutoNode;
91         root->len = 0;
92         root->suf = NULL;
93         last = root;
94         for (char c : to_suffix) extend(c);
95     }
96 };
97
98 #include <bits/stdc++.h>
99 using namespace std;
100 typedef long long ll; #section{Dinic}

```



```

5 struct MaxFlow{
6     typedef long long ll;
7     const ll INF = 1e18;
8     struct Edge{
9         int u,v;
10        ll c,rc;
11        shared_ptr<ll> flow;
12        Edge(int _u, int _v, ll _c, ll _rc = 0):u(_u),v(_v),c(_c),rc(_rc){
13        }
14    };
15    struct FlowTracker{
16        shared_ptr<ll> flow;
17        ll cap, rcap;
18        bool dir;
19        FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
20        ↪ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
21        ll rem() const {
22            if(dir == 0){
23                return cap-*flow;
24            }
25            else{
26                return rcap+*flow;
27            }
28        }
29        void add_flow(ll f){
30            if(dir == 0)
31                *flow += f;
32            else
33                *flow -= f;
34            assert(*flow <= cap);
35            assert(-*flow <= rcap);
36        }
37        operator ll() const { return rem(); }
38        void operator--=(ll x){ add_flow(x); }
39        void operator+=(ll x){ add_flow(-x); }
40    };
41    int source,sink;
42    vector<vector<int>> adj;
43    vector<vector<FlowTracker>> cap;
44    vector<Edge> edges;
45    MaxFlow(int _source, int _sink):source(_source),sink(_sink){
46        assert(source != sink);
47    }
48    int add_edge(int u, int v, ll c, ll rc = 0){
49        edges.push_back(Edge(u,v,c,rc));
50        return edges.size()-1;
51    }
52    vector<int> now,lv1;
53    void prep(){
54        int max_id = max(source, sink);
55        for(auto edge : edges)
56            max_id = max(max_id, max(edge.u, edge.v));
57        adj.resize(max_id+1);
58        cap.resize(max_id+1);

```

#787

#844

#287

#080

#328

```

58    now.resize(max_id+1);
59    lv1.resize(max_id+1);
60    for(auto &edge : edges){
61        auto flow = make_shared<ll>(0);
62        adj[edge.u].push_back(edge.v);
63        cap[edge.u].push_back(FlowTracker(edge.c, edge.rc, flow, 0));
64        if(edge.u != edge.v){
65            adj[edge.v].push_back(edge.u);
66            cap[edge.v].push_back(FlowTracker(edge.c, edge.rc, flow, 1));
67        }
68        assert(cap[edge.u].back() == edge.c);
69        edge.flow = flow;
70    }
71 }
72 bool dinic_bfs(){
73     fill(now.begin(),now.end(),0);
74     fill(lv1.begin(),lv1.end(),0);
75     lv1[source] = 1;
76     vector<int> bfs(1,source);
77     for(int i = 0; i < bfs.size(); ++i){
78         int u = bfs[i];
79         for(int j = 0; j < adj[u].size(); ++j){
80             int v = adj[u][j];
81             if(cap[u][j] > 0 && lv1[v] == 0){
82                 lv1[v] = lv1[u]+1;
83                 bfs.push_back(v);
84             }
85         }
86     }
87     return lv1[sink] > 0;
88 }
89 ll dinic_dfs(int u, ll flow){
90     if(u == sink)
91         return flow;
92     while(now[u] < adj[u].size()){
93         int v = adj[u][now[u]];
94         if(lv1[v] == lv1[u] + 1 && cap[u][now[u]] != 0){
95             ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
96             if(res > 0){
97                 cap[u][now[u]] -= res;
98                 return res;
99             }
100         }
101         ++now[u];
102     }
103     return 0;
104 }
105 ll calc_max_flow(){
106     prep();
107     ll ans = 0;
108     while(dinic_bfs()){
109         ll cur = 0;
110         do{

```

#717

#038

#010

#014

#197

```

111     cur = dinic_dfs(source, INF);
112     ans += cur;
113 }while(cur > 0);
114 }
115 return ans;
116 }
117 ll flow_on_edge(int edge_index){
118     assert(edge_index < edges.size());
119     return *edges[edge_index].flow;
120 }
121 };
122 int main(){
123     int n,m;
124     cin >> n >> m;
125     vector<pair<int, pair<int, int> > > graph(m);
126     for(int i=0; i<m; ++i){
127         cin>>graph[i].second.first>>graph[i].second.second>>graph[i].first;
128     }
129     ll res=0;
130     for(auto cur : graph){
131         auto mf = MaxFlow(cur.second.first, cur.second.second); // arguments
132         ↪ source and sink, memory usage O(largest node index + input
133         ↪ size), sink doesn't need to be last index
134         for(int i = 0; i < m; ++i){
135             if(graph[i].first > cur.first){
136                 mf.add_edge(graph[i].second.first, graph[i].second.second, 1, 1);
137                 ↪ // store edge index if care about flow value
138             }
139         }
140     }
141     res += mf.calc_max_flow();
142     cout<<res<<endl;
143 }

```

#817

%583

10 Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$

```

1 const int nmax=1055;
2 const ll inf=1e14;
3 int t, n, v; //0 is source, v-1 sink
4 ll rem_flow[nmax][nmax]; //set [x][y] for directed capacity from x to
5 ↪ y.
6 ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO
7 ↪ inf IF NOT USED
8 ll min_dist[nmax];
9 int prev_node[nmax];
10 ll node_flow[nmax];
11 bool visited[nmax];
12 ll tot_cost, tot_flow; //output
13 void min_cost_max_flow(){
14     tot_cost=0;
15     tot_flow=0;
16     ll sink_pot=0;
17     min_dist[0] = 0;
18     for(int i=1; i<=v; ++i){ //incase of no negative edges Bellman-Ford
19         ↪ can be removed.

```

%576

%927

```

17     min_dist[i]=inf;
18 }
19 for(int i=0; i<v-1; ++i){
20     for(int j=0; j<v; ++j){
21         for(int k=0; k<v; ++k){
22             if(rem_flow[j][k] > 0 && min_dist[j]+cost[j][k] < min_dist[k])
23                 min_dist[k] = min_dist[j]+cost[j][k];
24         }
25     }
26 }
27 for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
28     for(int j=0; j<v; ++j){
29         if(cost[i][j]!=inf){
30             cost[i][j]+=min_dist[i];
31             cost[i][j]-=min_dist[j];
32         }
33     }
34 }
35 sink_pot+=min_dist[v-1]; //Bellman-Ford end.
36 while(true){
37     for(int i=0; i<=v; ++i){ //node after sink is used as start value
38         ↪ for Dijkstra.
39         min_dist[i]=inf;
40         visited[i]=false;
41     }
42     min_dist[0]=0;
43     node_flow[0]=inf;
44     int min_node;
45     while(true){ //Use Dijkstra to calculate potentials
46         int min_node=v;
47         for(int i=0; i<v; ++i){
48             if(!visited[i] && min_dist[i]<min_dist[min_node])
49                 min_node=i;
50         }
51         if(min_node==v) break
52         visited[min_node]=true;
53         for(int i=0; i<v; ++i){
54             if(!visited[i] && min_dist[min_node]+cost[min_node][i] <
55                 ↪ min_dist[i]){
56                 min_dist[i]=min_dist[min_node]+cost[min_node][i];
57                 prev_node[i]=min_node;
58                 node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
59             }
60         }
61     }
62     if(min_dist[v-1]==inf) break
63     for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
64         for(int j=0; j<v; ++j){ //Found path from source to sink becomes
65             ↪ 0 cost.
66             if(cost[i][j]!=inf){
67                 cost[i][j]+=min_dist[i];
68                 cost[i][j]-=min_dist[j];

```

#599

%849

#782

#881

#083

```

67     }
68 }
69 sink_pot+=min_dist[v-1];
70 tot_flow+=node_flow[v-1];
71 tot_cost+=sink_pot*node_flow[v-1];
72 int cur=v-1;
73 while(cur!=0){ //Backtrack along found path that now has 0 cost.
74     rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
75     rem_flow[cur][prev_node[cur]]+=node_flow[v-1];           #582
76     cost[cur][prev_node[cur]]=0;
77     if(rem_flow[prev_node[cur]][cur]==0)
78         cost[prev_node[cur]][cur]=inf;
79     cur=prev_node[cur];
80 }
81 }
82 }                                     %803
83 int main(){//http://www.spoj.com/problems/GREED/
84     cin>>t;
85     for(int i=0; i<t; ++i){
86         cin>>n;
87         for(int j=0; j<nmax; ++j){
88             for(int k=0; k<nmax; ++k){
89                 cost[j][k]=inf;
90                 rem_flow[j][k]=0;
91             }
92         }
93         for(int j=1; j<=n; ++j){
94             cost[j][2*n+1]=0;
95             rem_flow[j][2*n+1]=1;
96         }
97         for(int j=1; j<=n; ++j){
98             int card;
99             cin>>card;
100             ++rem_flow[0][card];
101             cost[0][card]=0;
102         }
103         int ex_c;
104         cin>>ex_c;
105         for(int j=0; j<ex_c; ++j){
106             int a, b;
107             cin>>a>>b;
108             if(b<a) swap(a,b);
109             cost[a][b]=1;
110             rem_flow[a][b]=nmax;
111             cost[b][n+b]=0;
112             rem_flow[b][n+b]=nmax;
113             cost[n+b][a]=1;
114             rem_flow[n+b][a]=nmax;
115         }
116         v=2*n+2;
117         min_cost_max_flow();
118         cout<<tot_cost<<'\n';
119     }
120 }

```

11 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$

```

1 struct Network {
2     struct Node;
3     struct Edge {
4         Node *u, *v;
5         int f, c, cost;
6         Node* from(Node* pos) {
7             if(pos == u)
8                 return v;
9             return u;
10        }
11        int getCap(Node* pos) {
12            if(pos == u)
13                return c-f;
14            return f;
15        }
16        int addFlow(Node* pos, int toAdd) {
17            if(pos == u) {
18                f += toAdd;
19                return toAdd * cost;
20            } else {
21                f -= toAdd;
22                return -toAdd * cost;
23            }
24        }
25    };
26    struct Node {
27        vector<Edge*> conn;
28        int index;
29    };
30    deque<Node> nodes;
31    deque<Edge> edges;
32    Node* addNode() {
33        nodes.push_back(Node());
34        nodes.back().index = nodes.size()-1;
35        return &nodes.back();
36    }
37    Edge* addEdge(Node* u, Node* v, int f, int c, int cost) {
38        edges.push_back({u, v, f, c, cost});
39        u->conn.push_back(&edges.back());
40        v->conn.push_back(&edges.back());
41        return &edges.back();
42    }
43    //Assumes all needed flow has already been added
44    int minCostMaxFlow() {
45        int n = nodes.size();
46        int result = 0;
47        struct State {
48            int p;
49            Edge* used;
50        };
51        while(1) {

```

#042

#965

#534

#507

#877

```

52 vector<vector<State> > state(1, vector<State>(n, {0, 0}));
53 for(int lev = 0; lev < n; lev++) {
54     state.push_back(state[lev]);
55     for(int i=0; i<n; i++){
56         if(lev == 0 || state[lev][i].p < state[lev-1][i].p) {
57             for(Edge* edge : nodes[i].conn){
58                 if(edge->getCap(&nodes[i]) > 0) {
59                     int np = state[lev][i].p + (edge->u == &nodes[i] ?
60                         edge->cost : -edge->cost);
61                     int ni = edge->from(&nodes[i])->index;
62                     if(np < state[lev+1][ni].p) {
63                         state[lev+1][ni].p = np;
64                         state[lev+1][ni].used = edge;
65                     }
66                 }
67             }
68         }
69     }
70     //Now look at the last level
71     bool valid = false;
72     for(int i=0; i<n; i++)
73         if(state[n-1][i].p > state[n][i].p) {
74             valid = true;
75             vector<Edge*> path;
76             int cap = 1000000000;
77             Node* cur = &nodes[i];
78             int clev = n;
79             vector<bool> explr(n, false);
80             while(!explr[cur->index]) {
81                 explr[cur->index] = true;
82                 State cstate = state[clev][cur->index];
83                 cur = cstate.used->from(cur);
84                 path.push_back(cstate.used);
85             }
86             reverse(path.begin(), path.end());
87             {
88                 int i=0;
89                 Node* cur2 = cur;
90                 do {
91                     cur2 = path[i]->from(cur2);
92                     i++;
93                 } while(cur2 != cur);
94                 path.resize(i);
95             }
96             for(auto edge : path) {
97                 cap = min(cap, edge->getCap(cur));
98                 cur = edge->from(cur);
99             }
100             for(auto edge : path) {
101                 result += edge->addFlow(cur, cap);
102                 cur = edge->from(cur);
103             }
104         }

```

#281

#283

#954

#990

#599

```

105         if(!valid) break;
106     }
107     return result;
108 }
109 };

```

%900

12 DMST $\mathcal{O}(E \log V)$

```

1 struct EdgeDesc{
2     int from, to, w;
3 };
4 struct DMST{
5     struct Node;
6     struct Edge{
7         Node *from;
8         Node *tar;
9         int w;
10        bool inc;
11    };
12    struct Circle{
13        bool vis = false;
14        vector<Edge*> contents;
15        void clean(int idx);
16    };
17    const static greater<pair<ll, Edge*> > comp; //Can use inline static
18    // since C++17
19    static vector<Circle> to_process;
20    static bool no_dmst;
21    static Node *root;
22    struct Node{
23        Node *par = NULL;
24        vector<pair<int, int> > out_cands; //Circ, edge idx
25        vector<pair<ll, Edge*> > con;
26        bool in_use = false;
27        ll w = 0; //extra to add to edges in con
28        Node *anc(){
29            if(!par)
30                return this;
31            while(par->par)
32                par = par->par;
33            return par;
34        }
35        void clean(){
36            if(!no_dmst){
37                in_use = false;
38                for(auto &cur : out_cands)
39                    to_process[cur.first].clean(cur.second);
40            }
41        }
42        Node *con_to_root(){
43            if(anc() == root)
44                return root;
45            in_use = true;
46            Node *super = this; //Will become root or the first Node
47            // encountered in a loop.

```

#186

#536

#425

#561

```

46 while(super == this){
47     while(!con.empty() && con.front().second->tar->anc() == anc()){
48         pop_heap(con.begin(), con.end(), comp);
49         con.pop_back();
50     }
51     if(con.empty()){
52         no_dmst = true;
53         return root;
54     }
55     pop_heap(con.begin(), con.end(), comp);
56     auto nxt = con.back();
57     con.pop_back();
58     w = -nxt.first;
59     if(nxt.second->tar->in_use){ //anc() wouldn't change anything
60         super = nxt.second->tar->anc();
61         to_process.resize(to_process.size()+1);
62     } else {
63         super = nxt.second->tar->con_to_root();
64     }
65     if(super != root){
66         to_process.back().contents.push_back(nxt.second);
67         out_cands.emplace_back(to_process.size()-1,
68             ↳ to_process.back().contents.size()-1);
69     } else { //Clean circles
70         nxt.second->inc = true;
71         nxt.second->from->clean();
72     }
73     if(super != root){ //we are some loops non first Node.
74         if(con.size() > super->con.size()){
75             swap(con, super->con); //Largest con in loop should not be
76             ↳ copied.
77             swap(w, super->w);
78         }
79         for(auto cur : con){
80             super->con.emplace_back(cur.first - super->w + w,
81                 ↳ cur.second);
82             push_heap(super->con.begin(), super->con.end(), comp);
83         }
84         par = super; //root or anc() of first Node encountered in a loop
85         return super;
86     };
87     Node *cur_root;
88     vector<Node> graph;
89     vector<Edge> edges;
90     DMST(int n, vector<EdgeDesc> &desc, int r){ //Self loops and multiple
91         ↳ edges are okay.
92         graph.resize(n);
93         cur_root = &graph[r];
94         for(auto &cur : desc) //Edges are reversed internally
95             edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
96         for(int i=0; i<desc.size(); ++i)

```

#522

#174

#629

#375

#076

```

96     graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
97     for(int i=0; i<n; ++i)
98         make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
99 }
100 bool find(){
101     root = cur_root;
102     no_dmst = false;
103     for(auto &cur : graph){
104         cur.con_to_root();
105         to_process.clear();
106         if(no_dmst) return false;
107     }
108     return true;
109 }
110 ll weight(){
111     ll res = 0;
112     for(auto &cur : edges){
113         if(cur.inc)
114             res += cur.w;
115     }
116     return res;
117 }
118 };
119 void DMST::Circle::clean(int idx){
120     if(!vis){
121         vis = true;
122         for(int i=0; i<contents.size(); ++i){
123             if(i != idx){
124                 contents[i]->inc = true;
125                 contents[i]->from->clean();
126             }
127         }
128     }
129 }
130 const greater<pair<ll, DMST::Edge *> > DMST::comp;
131 vector<DMST::Circle> DMST::to_process;
132 bool DMST::no_dmst;
133 DMST::Node *DMST::root;

```

#469

%732

%477

#711

%771

13 Bridges $\mathcal{O}(n)$

```

1 struct vert;
2 struct edge{
3     bool exists = true;
4     vert *dest;
5     edge *rev;
6     edge(vert *_dest) : dest(_dest){
7         rev = NULL;
8     }
9     vert &operator*(){
10         return *dest;
11     }
12     vert *operator->(){
13         return dest;

```

#955

14 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$

```

14 }
15 bool is_bridge();
16 };
17 struct vert{
18     deque<edge> con;
19     int val = 0;
20     int seen;
21     int dfs(int upd, edge *ban){ //handles multiple edges
22         if(!val){
23             val = upd;
24             seen = val;
25             for(edge &nxt : con){
26                 if(nxt.exists && (&nxt) != ban)
27                     seen = min(seen, nxt->dfs(upd+1, nxt.rev));
28             }
29         }
30         return seen;
31     }
32     void remove_adj_bridges(){
33         for(edge &nxt : con){
34             if(nxt.is_bridge())
35                 nxt.exists = false;
36         }
37     }
38     int cnt_adj_bridges(){
39         int res = 0;
40         for(edge &nxt : con)
41             res += nxt.is_bridge();
42         return res;
43     }
44 };
45 bool edge::is_bridge(){
46     return exists && (dest->seen > rev->dest->val || dest->val <
47         ↪ rev->dest->seen);
48 }
49 vert graph[nmax];
50 int main(){ //Mechanics Practice BRIDGES
51     int n, m;
52     cin>>n>>m;
53     for(int i=0; i<m; ++i){
54         int u, v;
55         scanf("%d %d", &u, &v);
56         graph[u].con.emplace_back(graph+v);
57         graph[v].con.emplace_back(graph+u);
58         graph[u].con.back().rev = &graph[v].con.back();
59         graph[v].con.back().rev = &graph[u].con.back();
60     }
61     graph[1].dfs(1, NULL);
62     int res = 0;
63     for(int i=1; i<=n; ++i)
64         res += graph[i].cnt_adj_bridges();
65     cout<<res/2<<endl;

```

#336

#673
%624

%106

%056

%223

```

1 struct Graph {
2     int n;
3     vector<vector<int>> > conn;
4     Graph(int nsize) {
5         n = nsize;
6         conn.resize(n);
7     }
8     void add_edge(int u, int v) {
9         conn[u].push_back(v);
10    }
11    void _topsort_dfs(int pos, vector<int> &result, vector<bool>
12        ↪ &explr, vector<vector<int>> > &revconn) {
13        if(explr[pos])
14            return;
15        explr[pos] = true;
16        for(auto next : revconn[pos])
17            _topsort_dfs(next, result, explr, revconn);
18        result.push_back(pos);
19    }
20    vector<int> topsort() {
21        vector<vector<int>> > revconn(n);
22        for(int u = 0; u < n; u++) {
23            for(auto v : conn[u])
24                revconn[v].push_back(u);
25        }
26        vector<int> result;
27        vector<bool> explr(n, false);
28        for(int i=0; i < n; i++)
29            _topsort_dfs(i, result, explr, revconn);
30        reverse(result.begin(), result.end());
31        return result;
32    }
33    void dfs(int pos, vector<int> &result, vector<bool> &explr) {
34        if(explr[pos])
35            return;
36        explr[pos] = true;
37        for(auto next : conn[pos])
38            dfs(next, result, explr);
39        result.push_back(pos);
40    }
41    vector<vector<int>> > scc(){ // tested on
42        ↪ https://www.hackerearth.com/practice/algorithms/graphs/strongly-connect
43        vector<int> order = topsort();
44        reverse(order.begin(), order.end());
45        vector<bool> explr(n, false);
46        vector<vector<int>> > results;
47        for(auto it = order.rbegin(); it != order.rend(); ++it){
48            vector<int> component;
49            _topsort_dfs(*it, component, explr, conn);
50            sort(component.begin(), component.end());
51            results.push_back(component);

```

#078

#346

#991

%603

#741

```

51     sort(results.begin(), results.end());
52     return results;
53 }
54 };
55 //Solution for:
56 ↪ http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
57 int main() {
58     int n, m;
59     cin >> n >> m;
60     Graph g(2*m);
61     for(int i=0; i<n; i++) {
62         int a, sa, b, sb;
63         cin >> a >> sa >> b >> sb;
64         a--, b--;
65         g.add_edge(2*a + 1 - sa, 2*b + sb);
66         g.add_edge(2*b + 1 - sb, 2*a + sa);
67     }
68     vector<int> state(2*m, 0);
69     {
70         vector<int> order = g.topsort();
71         vector<bool> explr(2*m, false);
72         for(auto u : order) {
73             vector<int> traversed;
74             g.dfs(u, traversed, explr);
75             if(traversed.size() > 0 && !state[traversed[0]^1]) {
76                 for(auto c : traversed)
77                     state[c] = 1;
78             }
79         }
80         for(int i=0; i < m; i++) {
81             if(state[2*i] == state[2*i+1]) {
82                 cout << "IMPOSSIBLE\n";
83                 return 0;
84             }
85         }
86         for(int i=0; i < m; i++) {
87             cout << state[2*i+1] << '\n';
88         }
89         return 0;
90 }

```

%983

15 Generic persistent compressed lazy segment tree

```

1 struct Seg{
2     ll sum=0;
3     void recalc(const Seg &lhs_seg, int lhs_len, const Seg &rhs_seg, int
4         ↪ rhs_len){
5         sum = lhs_seg.sum + rhs_seg.sum;
6     }
7 } __attribute__((packed));
8 struct Lazy{
9     ll add;
10    ll assign_val; //LLONG_MIN if no assign;
11    void init(){

```

#883

```

11    add = 0;
12    assign_val = LLONG_MIN;
13 }
14 Lazy(){ init(); }
15 void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len){
16     lhs_lazy = *this;
17     rhs_lazy = *this;
18     init();
19 }
20 void merge(Lazy &oth, int len){
21     if(oth.assign_val != LLONG_MIN){
22         add = 0;
23         assign_val = oth.assign_val;
24     }
25     add += oth.add;
26 }
27 void apply_to_seg(Seg &cur, int len) const{
28     if(assign_val != LLONG_MIN){
29         cur.sum = len * assign_val;
30     }
31     cur.sum += len * add;
32 }
33 } __attribute__((packed));
34 struct Node{ //Following code should not need to be modified
35     int ver;
36     bool is_lazy = false;
37     Seg seg;
38     Lazy lazy;
39     Node *lc=NULL, *rc=NULL;
40     void init(){
41         if(!lc){
42             lc = new Node {ver};
43             rc = new Node {ver};
44         }
45     }
46     Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver){
47         if(ver != tar_ver){
48             Node *rep = new Node(*this);
49             rep->ver = tar_ver;
50             return rep->upd(L, R, l, r, val, tar_ver);
51         }
52         if(L >= l && R <= r){
53             val.apply_to_seg(seg, R-L);
54             lazy.merge(val, R-L);
55             is_lazy = true;
56         } else {
57             init();
58             int M = (L+R)/2;
59             if(is_lazy){
60                 Lazy l_val, r_val;
61                 lazy.split(l_val, r_val, R-L);
62                 lc = lc->upd(L, M, L, M, l_val, ver);
63                 rc = rc->upd(M, R, M, R, r_val, ver);

```

#470

#216

%625

#313

#138

#104

```

64     is_lazy = false;
65 }
66 Lazy l_val , r_val;
67 val.split(l_val, r_val, R-L);
68 if(l < M)
69     lc = lc->upd(L, M, l, r, l_val, ver);
70 if(M < r)
71     rc = rc->upd(M, R, l, r, r_val, ver);
72 seg.recalc(lc->seg, M-L, rc->seg, R-M);
73 }
74 return this;
75 }
76 void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp, bool
    ↪ last_ver){
77     if(L >= l && R <= r){
78         tmp->recalc(*lft_res, L-l, seg, R-L);
79         swap(lft_res, tmp);
80     } else {
81         init();
82         int M = (L+R)/2;
83         if(is_lazy){
84             Lazy l_val , r_val;
85             lazy.split(l_val, r_val, R-L);
86             lc = lc->upd(L, M, L, M, l_val, ver+last_ver);
87             lc->ver = ver;
88             rc = rc->upd(M, R, M, R, r_val, ver+last_ver);
89             rc->ver = ver;
90             is_lazy = false;
91         }
92         if(l < M)
93             lc->get(L, M, l, r, lft_res, tmp, last_ver);
94         if(M < r)
95             rc->get(M, R, l, r, lft_res, tmp, last_ver);
96     }
97 }
98 } __attribute__((packed));
99 struct SegTree{ //indexes start from 0, ranges are [beg, end)
100     vector<Node *> roots; //versions start from 0
101     int len;
102     SegTree(int _len) : len(_len){
103         roots.push_back(new Node {0});
104     }
105     int upd(int l, int r, Lazy &val, bool new_ver = false){
106         Node *cur_root = roots.back()->upd(0, len, l, r, val,
    ↪ roots.size()-1, new_ver);
107         if(cur_root != roots.back())
108             roots.push_back(cur_root);
109         return roots.size()-1;
110     }
111     Seg get(int l, int r, int ver = -1){
112         if(ver == -1)
113             ver = roots.size()-1;
114         Seg seg1, seg2;
115         Seg *pres = &seg1, *ptmp = &seg2;

```

#245

#726

#696

#295

#977

```

116     roots[ver]->get(0, len, l, r, pres, ptmp, roots.size()-1);
117     return *pres;
118 }
119 };
120 int main(){
121     int n, m; //solves Mechanics Practice LAZY
122     cin>>n>>m;
123     SegTree seg_tree(1<<17);
124     for(int i=0; i<n; ++i){
125         Lazy tmp;
126         scanf("%lld", &tmp.assign_val);
127         seg_tree.upd(i, i+1, tmp);
128     }
129     for(int i=0; i<m; ++i){
130         int o;
131         int l, r;
132         scanf("%d %d %d", &o, &l, &r);
133         --l;
134         if(o==1){
135             Lazy tmp;
136             scanf("%lld", &tmp.add);
137             seg_tree.upd(l, r, tmp);
138         } else if(o==2){
139             Lazy tmp;
140             scanf("%lld", &tmp.assign_val);
141             seg_tree.upd(l, r, tmp);
142         } else {
143             Seg res = seg_tree.get(l, r);
144             printf("%lld\n", res.sum);
145         }
146     }
147 }

```

%542

16 Templated HLD $\mathcal{O}(M(n) \log n)$ per query

```

1 class dummy {
2 public:
3     dummy () {}
4     dummy (int, int) {}
5     void set (int, int) {}
6     int query (int left, int right) {
7         cout << this << ' ' << left << ' ' << right << endl;
8     }
9 };
10 /* T should be the type of the data stored in each vertex;
11  * DS should be the underlying data structure that is used to perform
    ↪ the
12  * group operation. It should have the following methods:
13  * * DS () - empty constructor
14  * * DS (int size, T initial) - constructs the structure with the given
    ↪ size,
15  * * initially filled with initial.
16  * * void set (int index, T value) - set the value at index `index` to
    ↪ `value`

```

%932

```

17   * T query (int left, int right) - return the "sum" of elements
18   ↪ between left and right, inclusive.
19   */
20  template<typename T, class DS>
21  class HLD {
22      int vertexc;
23      vector<int> *adj;
24      vector<int> subtree_size;
25      DS structure;
26      DS aux;
27      void build_sizes (int vertex, int parent) {
28          subtree_size[vertex] = 1;
29          for (int child : adj[vertex]) {
30              if (child != parent) {
31                  build_sizes(child, vertex);
32                  subtree_size[vertex] += subtree_size[child];
33              }
34          }
35          int cur;
36          vector<int> ord;
37          vector<int> chain_root;
38          vector<int> par;
39          void build_hld (int vertex, int parent, int chain_source) {
40              cur++;
41              ord[vertex] = cur;
42              chain_root[vertex] = chain_source;
43              par[vertex] = parent;
44              if (adj[vertex].size() > 1) {
45                  int big_child, big_size = -1;
46                  for (int child : adj[vertex]) {
47                      if ((child != parent) && (subtree_size[child] > big_size)) {
48                          big_child = child;
49                          big_size = subtree_size[child];
50                      }
51                  }
52                  build_hld(big_child, vertex, chain_source);
53                  for (int child : adj[vertex]) {
54                      if ((child != parent) && (child != big_child))
55                          build_hld(child, vertex, child);
56                  }
57              }
58          }
59      public:
60          HLD (int _vertexc) {
61              vertexc = _vertexc;
62              adj = new vector<int> [vertexc + 5];
63          }
64          void add_edge (int u, int v) {
65              adj[u].push_back(v);
66              adj[v].push_back(u);
67          }
68          void build (T initial) {
69              subtree_size = vector<int> (vertexc + 5);

```

#037

#593

#646

#738

#841

```

70      ord = vector<int> (vertexc + 5);
71      chain_root = vector<int> (vertexc + 5);
72      par = vector<int> (vertexc + 5);
73      cur = 0;
74      build_sizes(1, -1);
75      build_hld(1, -1, 1);
76      structure = DS (vertexc + 5, initial);
77      aux = DS (50, initial);
78  }
79  void set (int vertex, int value) {
80      structure.set(ord[vertex], value);
81  }
82  T query_path (int u, int v) { /* returns the "sum" of the path u->v
83      ↪ */
84      int cur_id = 0;
85      while (chain_root[u] != chain_root[v]) {
86          if (ord[u] > ord[v]) {
87              cur_id++;
88              aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
89              u = par[chain_root[u]];
90          } else {
91              cur_id++;
92              aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
93              v = par[chain_root[v]];
94          }
95      }
96      cur_id++;
97      aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u],
98      ↪ ord[v])));
99      return aux.query(1, cur_id);
100  }
101  void print () {
102      for (int i = 1; i <= vertexc; i++)
103          cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' <<
104          ↪ par[i] << endl;
105  }
106  int main () {
107      int vertexc;
108      cin >> vertexc;
109      HLD<int, dummy> hld (vertexc);
110      for (int i = 0; i < vertexc - 1; i++) {
111          int u, v;
112          cin >> u >> v;
113          hld.add_edge(u, v);
114      }
115      hld.build(0);
116      hld.print();
117      int queryc;
118      cin >> queryc;
119      for (int i = 0; i < queryc; i++) {
120          int u, v;
121          cin >> u >> v;

```

#793

#517

%257

```

120     hld.query_path(u, v);
121     cout << endl;
122 }
123 }

```

17 Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates,
  // elements, return value.
2 // Includes coordinate compression.
3 template < typename elem_t, typename coord_t, coord_t n_inf, typename
  ↪ ret_t >
4 class BIT {
5     vector< coord_t > positions;
6     vector< elem_t > elems;
7     bool initiated = false;
8 public:
9     BIT() {
10         positions.push_back(n_inf);
11     }
12     void initiate() {
13         if (initiated) {
14             for (elem_t &c_elem : elems)
15                 c_elem.initiate();
16         } else {
17             initiated = true;
18             sort(positions.begin(), positions.end());
19             positions.resize(unique(positions.begin(), positions.end()) -
20                               ↪ positions.begin());
21             elems.resize(positions.size());
22         }
23     }
24     template < typename... loc_form >
25     void update(coord_t cord, loc_form... args) {
26         if (initiated) {
27             int pos = lower_bound(positions.begin(), positions.end(), cord) -
28                 ↪ positions.begin();
29             for (; pos < positions.size(); pos += pos & -pos)
30                 elems[pos].update(args...);
31         } else {
32             positions.push_back(cord);
33         }
34     }
35     template < typename... loc_form >
36     ret_t query(coord_t cord, loc_form... args) { //sum in open interval
37         ↪ (-inf, cord)
38         ret_t res = 0;
39         int pos = (lower_bound(positions.begin(), positions.end(), cord) -
40                 ↪ positions.begin())-1;
41         for (; pos > 0; pos -= pos & -pos)
42             res += elems[pos].query(args...);
43         return res;
44     }
45 };
46 template < typename internal_type >

```

#448

#036

#154

#895

```

43 struct wrapped {
44     internal_type a = 0;
45     void update(internal_type b) {
46         a += b;
47     }
48     internal_type query() {
49         return a;
50     }
51     // Should never be called, needed for compilation
52     void initiate() {
53         cerr << 'i' << endl;
54     }
55     void update() {
56         cerr << 'u' << endl;
57     }
58 };
59 int main() {
60     // return type should be same as type inside wrapped
61     BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll >
62     ↪ fenwick;
63     int dim = 2;
64     vector< tuple< int, int, ll > > to_insert;
65     to_insert.emplace_back(1, 1, 1);
66     // set up all positions that are to be used for update
67     for (int i = 0; i < dim; ++i) {
68         for (auto &cur : to_insert)
69             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include
70             ↪ value which won't be used
71         fenwick.initiate();
72     }
73     // actual use
74     for (auto &cur : to_insert)
75         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
76     cout << fenwick.query(2, 2)<<'\n';
77 }

```

#560

%714

18 Treap $\mathcal{O}(\log n)$ per query

```

1 mt19937 randgen;
2 struct Treap {
3     struct Node {
4         int key;
5         int value;
6         unsigned int priority;
7         long long total;
8         Node* lch;
9         Node* rch;
10        Node(int new_key, int new_value) {
11            key = new_key;
12            value = new_value;
13            priority = randgen();
14            total = new_value;
15            lch = 0;
16            rch = 0;
17        }

```

#698


```

18     void update() {
19         total = value;
20         if(lch) total += lch->total;
21         if(rch) total += rch->total;
22     }
23 };
24 deque<Node> nodes;
25 Node* root = 0;
26 pair<Node*, Node*> split(int key, Node* cur) {
27     if(cur == 0) return {0, 0};
28     pair<Node*, Node*> result;
29     if(key <= cur->key) {
30         auto ret = split(key, cur->lch);
31         cur->lch = ret.second;
32         result = {ret.first, cur};
33     } else {
34         auto ret = split(key, cur->rch);
35         cur->rch = ret.first;
36         result = {cur, ret.second};
37     }
38     cur->update();
39     return result;
40 }
41 Node* merge(Node* left, Node* right) {
42     if(left == 0) return right;
43     if(right == 0) return left;
44     Node* top;
45     if(left->priority < right->priority) {
46         left->rch = merge(left->rch, right);
47         top = left;
48     } else {
49         right->lch = merge(left, right->lch);
50         top = right;
51     }
52     top->update();
53     return top;
54 }
55 void insert(int key, int value) {
56     nodes.push_back(Node(key, value));
57     Node* cur = &nodes.back();
58     pair<Node*, Node*> ret = split(key, root);
59     cur = merge(ret.first, cur);
60     cur = merge(cur, ret.second);
61     root = cur;
62 }
63 void erase(int key) {
64     Node *left, *mid, *right;
65     tie(left, mid) = split(key, root);
66     tie(mid, right) = split(key+1, mid);
67     root = merge(left, right);
68 }
69 long long sum_upto(int key, Node* cur) {
70     if(cur == 0) return 0;
71     if(key <= cur->key) {

```

#295

#233

#230

#510

#760

#634

```

72         return sum_upto(key, cur->lch);
73     } else {
74         long long result = cur->value + sum_upto(key, cur->rch);
75         if(cur->lch) result += cur->lch->total;
76         return result;
77     }
78 }
79 long long get(int l, int r) {
80     return sum_upto(r+1, root) - sum_upto(l, root);
81 }
82 };
83 //Solution for:
84 ↪ http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP
85 int main() {
86     ios_base::sync_with_stdio(false);
87     cin.tie(0);
88     Treap treap;
89     cin >> m;
90     for(int i=0; i<m; i++) {
91         int type;
92         cin >> type;
93         if(type == 1) {
94             int x, y;
95             cin >> x >> y;
96             treap.insert(x, y);
97         } else if(type == 2) {
98             int x;
99             cin >> x;
100             treap.erase(x);
101         } else {
102             int l, r;
103             cin >> l >> r;
104             cout << treap.get(l, r) << endl;
105         }
106     }
107     return 0;
108 }

```

#509

%959

19 FFT 5M length/sec

integer $c = a * b$ is accurate if $c_i < 2^{49}$

```

1 struct Complex {
2     double a = 0, b = 0;
3     Complex &operator/=(const int &oth) {
4         a /= oth;
5         b /= oth;
6         return *this;
7     }
8 };
9 Complex operator+(const Complex &lft, const Complex &rgt) {
10     return Complex{lft.a + rgt.a, lft.b + rgt.b};
11 }

```

#384

```

12 Complex operator-(const Complex &lft, const Complex &rgt) {
13     return Complex{lft.a - rgt.a, lft.b - rgt.b};
14 }
15 Complex operator*(const Complex &lft, const Complex &rgt) {
16     return Complex{lft.a * rgt.a - lft.b * rgt.b, lft.a * rgt.b + lft.b *
17         ↪ rgt.a};
18 }
19 void fft_rec(Complex *arr, Complex *root_pow, int len) {
20     if (len != 1) {
21         fft_rec(arr, root_pow, len >> 1);
22     }
23     root_pow += len;
24     for (int i = 0; i < len; ++i) {
25         Complex tmp = arr[i] + root_pow[i] * arr[i + len];
26         arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
27         arr[i] = tmp;
28     }
29 }
30 void fft(vector< Complex > &arr, int ord, bool invert) {
31     assert(arr.size() == 1 << ord);
32     static vector< Complex > root_pow(1);
33     static int inc_pow = 1;
34     static bool is_inv = false;
35     if (inc_pow <= ord) {
36         int idx = root_pow.size();
37         root_pow.resize(1 << ord);
38         for (; inc_pow <= ord; ++inc_pow) {
39             for (int idx_p = 0; idx_p < 1 << (ord - 1); idx_p += 1 << (ord -
40                 ↪ inc_pow), ++idx) {
41                 root_pow[idx] =
42                     ↪ Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), sin(-idx_p *
43                     ↪ M_PI / (1 << (ord - 1)))};
44                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
45             }
46         }
47         if (invert != is_inv) {
48             is_inv = invert;
49             for (Complex &cur : root_pow) cur.b = -cur.b;
50         }
51         for (int i = 1, j=0; i < (1 << ord); ++i) {
52             int m = 1 << (ord-1);
53             bool cont = true;
54             while(cont){
55                 cont = j & m;
56                 j ^= m;
57                 m >>= 1;
58             }
59             if (i < j) swap(arr[i], arr[j]);
60         }
61         fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
62         if (invert)
63             for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);

```

#767

#689

#053

#565

#847

```

63 }
64 void mult_poly_mod(vector< int > &a, vector< int > &b, vector< int >
65     ↪ &c) { // c += a*b
66     static vector< Complex > arr[7]; // correct upto 0.5-2M elements(mod
67     ↪ ~1e9)
68     if (c.size() < 400) {
69         for (int i = 0; i < a.size(); ++i)
70             for (int j = 0; j < b.size() && i + j < c.size(); ++j)
71                 c[i + j] = ((ll)a[i] * b[j] + c[i + j]) % mod;
72     } else {
73         int fft_ord = 32 - __builtin_clz(c.size());
74         if (arr[0].size() != 1 << fft_ord)
75             for (int i = 0; i < 7; ++i) arr[i].resize(1 << fft_ord);
76         for (int i = 0; i < 7; ++i) fill(arr[i].begin(), arr[i].end(),
77             ↪ Complex{});
78         for (int &cur : a)
79             if (cur < 0) cur += mod;
80         for (int &cur : b)
81             if (cur < 0) cur += mod;
82         const int shift = 15;
83         const int mask = (1 << shift) - 1;
84         for (int i = 0; i < min(a.size(), c.size()); ++i) {
85             arr[0][i].a = a[i] & mask;
86             arr[1][i].a = a[i] >> shift;
87         }
88         for (int i = 0; i < min(b.size(), c.size()); ++i) {
89             arr[2][i].a = b[i] & mask;
90             arr[3][i].a = b[i] >> shift;
91         }
92         for (int i = 0; i < 4; ++i) fft(arr[i], fft_ord, false);
93         for (int i = 0; i < 2; ++i) {
94             for (int j = 0; j < 2; ++j) {
95                 for (int k = 0; k < (1 << fft_ord); ++k)
96                     arr[tar][k] = arr[tar][k] + arr[i][k] * arr[2 + j][k];
97             }
98         }
99         for (int i = 4; i < 7; ++i) {
100             fft(arr[i], fft_ord, true);
101             for (int k = 0; k < (int)c.size(); ++k)
102                 c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod) << (shift * (i
103                     ↪ - 4)))) % mod;
104         }
105     }
106 }

```

%836

#672

#694

%002

20 Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$

```

1 static vector< char > small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23,
2     ↪ 29, 31, 37};
3 __int128 pow_mod(__int128 a, ll n, __int128 mod) {
4     __int128 res = 1;
5     for (ll i = 1; i <= n; i <= 1) {
6         if (n & i)

```

#850

%047

```

6     res = (res * a) % mod;
7     a = (a * a) % mod;
8 }
9 return res;
10}
11bool is_prime(ll n) { //guaranteed for 64 bit numbers
12    if (n == 2 || n == 3) return true;
13    if (!(n & 1) || n == 1) return false;
14    ll s = __builtin_ctz(n - 1);
15    ll d = (n - 1) >> s;
16    __int128 mod = n;
17    for (__int128 a : small_primes) {
18        if (a >= mod) break;
19        a = pow_mod(a, d, mod);
20        if (a == 1 || a == mod - 1) continue;
21        for (ll r = 1; r < s; ++r) {
22            a = a * a % mod;
23            if (a == 1) return false;
24            if (a == mod - 1) break;
25        }
26        if (a != mod - 1) return false;
27    }
28    return true;
29}
30ll sqp1(ll x, ll n){
31    return ((__int128)x*x+1)%n;
32};
33ll pollard_rho(ll n){
34    while(true){
35        ll lv = rand()%n;
36        ll v = sqp1(lv, n);
37        int tar = 1;
38        for(int i=1; ; ++i){
39            ll g = __gcd(abs(v-lv), n);
40            if(g == n){
41                break;
42            } else if(g != 1){
43                return g;
44            }
45            if(i == tar){
46                lv = v;
47                tar <= 1;
48            }
49            v = sqp1(v, n);
50        }
51    }
52}
53void prime_factor(ll n, map<ll, int> &res){
54    while(!(n&1)){
55        ++res[2];
56        n >>=1;
57    }
58    for(int i=3; n>1 && i<=1000; i += 2){
59        while(!(n%i)){

```

#180

#754

%457

#695

#852

#475

```

60        ++res[i];
61        n /= i;
62    }
63 }
64 if(n != 1){
65     if(is_prime(n)){
66         ++res[n];
67     } else {
68         ll factor = pollard_rho(n);
69         prime_factor(factor, res);
70         prime_factor(n/factor, res);
71     }
72 }
73 }

```


Combinatorics Cheat Sheet

Useful formulas

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ — number of ways to choose k objects out of n

$\binom{n+k-1}{k-1}$ — number of ways to choose k objects out of n with repetitions

$[n]$ — Stirling numbers of the first kind; number of permutations of n elements with k cycles

$$[n+1] = n[n] + [n-1]$$

$$(x)_n = x(x-1)\dots x-n+1 = \sum_{k=0}^n (-1)^{n-k} [n]_k x^k$$

$\left\{ \begin{smallmatrix} n \\ m \end{smallmatrix} \right\}$ — Stirling numbers of the second kind; number of partitions of set $1, \dots, n$ into k disjoint subsets.

$$\left\{ \begin{smallmatrix} n+1 \\ m \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n \\ k-1 \end{smallmatrix} \right\}$$

$$\sum_{k=0}^n \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} (x)_k = x^n$$

$$C_n = \frac{1}{n+1} \binom{2n}{n} — \text{Catalan numbers}$$

$$C(x) = \frac{1-\sqrt{1-4x}}{2x}$$

Binomial transform

If $a_n = \sum_{k=0}^n \binom{n}{k} b_k$, then $b_n = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} a_k$

$$\bullet a = (1, x, x^2, \dots), b = (1, (x+1), (x+1)^2, \dots)$$

$$\bullet a_i = i^k, b_i = \left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\} i!$$

Burnside's lemma

Let G be a group of *action* on set X (Ex.: cyclic shifts of array, rotations and symmetries of $n \times n$ matrix, ...)

Call two objects x and y *equivalent* if there is an action f that transforms x to y : $f(x) = y$.

The number of equivalence classes then can be calculated as follows: $C = \frac{1}{|G|} \sum_{f \in G} |X^f|$, where X^f is the set of *fixed points* of f : $X^f = \{x | f(x) = x\}$

Generating functions

Ordinary generating function (o.g.f.) for sequence $a_0, a_1, \dots, a_n, \dots$ is $A(x) = \sum_{n=0}^{\infty} a_n x^n$

Exponential generating function (e.g.f.) for sequence $a_0, a_1, \dots, a_n, \dots$ is $A(x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}$

$$B(x) = A'(x), b_{n-1} = n \cdot a_n$$

$$c_n = \sum_{k=0}^n a_k b_{n-k} \text{ (o.g.f. convolution)}$$

$$c_n = \sum_{k=0}^n \binom{n}{k} a_k b_{n-k} \text{ (e.g.f. convolution, compute with FFT using } \widetilde{a_n} = \frac{a_n}{n!})$$

General linear recurrences

If $a_n = \sum_{k=1}^n b_k a_{n-k}$, then $A(x) = \frac{a_0}{1-B(x)}$. We also can compute all a_n with Divide-and-Conquer algorithm in $O(n \log^2 n)$.

Inverse polynomial modulo x^l

Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l \cdot Q(x)$ for some $Q(x)$

$$1. \text{ Start with } B_0(x) = \frac{1}{a_0}$$

$$2. \text{ Double the length of } B(x): B_{k+1}(x) = (-B_k(x)^2 A(x) + 2B_k(x)) \bmod x^{2^{k+1}}$$

Fast subset convolution

Given array a_i of size 2^k , calculate $b_i = \sum_{j \& i = i} b_j$

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      a[i + (1 << b)] += a[i]
```

Hadamard transform

Treat array a of size 2^k as k -dimensional array of size $2 \times 2 \times \dots \times 2$, calculate FFT of that array:

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      u = a[i], v = a[i + (1 << b)]
      a[i] = u + v
      a[i + (1 << b)] = u - v
```