

University of Tartu ICPC Team Notebook

(2017-2018) April 13, 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	Dinic
11	Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$
12	Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$
13	DMST $\mathcal{O}(E \log V)$
14	Bridges $\mathcal{O}(n)$
15	2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$
16	Lazy Segment Tree $\mathcal{O}(\log n)$ per query
17	Generic segment tree(lazy, noncommutative)
18	Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query
19	Templated HLD $\mathcal{O}(M(n) \log n)$ per query
20	Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query
21	Treap $\mathcal{O}(\log n)$ per query
22	FFT $\mathcal{O}(n \log(n))$

23 MOD int, extended Euctclidean

22

24 Rabbin Miller prime check

23

1 Setup

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

2 crc.sh

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

3 gcc ordered set

```
12 1 #include <bits/stdc++.h>
2 typedef long long ll;
13 3 using namespace std;
4 #include <ext/pb_ds/assoc_container.hpp>
14 5 #include <ext/pb_ds/tree_policy.hpp>
6 using namespace __gnu_pbds;
15 7 template <typename T>
8 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
        ↪ tree_order_statistics_node_update>;
16 9 int main(){
10   ordered_set<int> cur;
18 11 cur.insert(1);
12 cur.insert(3);
18 13 cout << cur.order_of_key(2) << endl; // the number of elements in the
        ↪ set less than 2
20 14 cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in
        ↪ the set(0-based)
20 15 cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in
        ↪ the set(0-based)
16 }
21
```

#221

%626

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; #300
11    return integrate(f, a, m, computing_power/2) + integrate(f, m, b,
12        ↪ computing_power/2); %821

```

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 #647
12    return ang < min_delta; %029
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; #688
24     c_A = a2 * (b2 + c2 - a2); //sin(2 * alpha) may be used as well
25     c_B = b2 * (a2 + c2 - b2);
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; %561
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();

```

```

38     return O + 3.0 * (centroid() - O);
39 };
40 point nine_point_circle_center(){ //euler line
41     point O = circum_center();
42     return O + 1.5 * (centroid() - O); #530
43 }; %132
44 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; #812
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;
60 } %471

```

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 }; #054

```

#673

```

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 }
41 Vec normal(Vec a) {
42     return Vec(-a.y, a.x) / len(a);
43 }
44
45 struct Segment {
46     Vec a, b;
47     Vec d() {
48         return b-a;
49     }
50 };
51 ostream& operator<<(ostream& l, Segment r) {
52     return l << r.a << '-' << r.b;
53 }
54 Vec intersection(Segment l, Segment r) {
55     Vec dl = l.d(), dr = r.d();
56     if(cross(dl, dr) == 0)
57         return {nanl(""), nanl("")};
58     long double h = cross(dr, l.a-r.a) / len(dr);
59     long double dh = cross(dr, dl) / len(dr);
60     return l.a + dl * (h / -dh);
61 }
62 //Returns the area bounded by halfplanes
63 long double getArea(vector<Segment> lines) {
64     long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
65     vector<Segment> linesBySide[2];
66     for(auto line : lines) {
67         if(line.b.y == line.a.y) {
68             if(line.a.x < line.b.x) {
69                 lowerbound = max(lowerbound, line.a.y);
70             } else {
71                 upperbound = min(upperbound, line.a.y);
72             }
73         } else if(line.a.y < line.b.y) {
74             linesBySide[1].push_back(line);
75         } else {
76             linesBySide[0].push_back({line.b, line.a});
77         }
78     }
79     sort(linesBySide[0].begin(), linesBySide[0].end(), [](Segment l,
80         Segment r) {
81         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a <
82             normal(r.d())*r.a;

```

#724

#872

%654

#355

#009

#597

```

37     return cross(l.d(), r.d()) < 0;
38 });
39 sort(linesBySide[1].begin(), linesBySide[1].end(), [](Segment l,
40     Segment r) {
41     if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a <
42         normal(r.d())*r.a;
43     return cross(l.d(), r.d()) > 0;
44 });
45 //Now find the application area of the lines and clean up redundant
46     ones
47 vector<long double> applyStart[2];
48 for(int side = 0; side < 2; side++) {
49     vector<long double> &apply = applyStart[side];
50     vector<Segment> curLines;
51     for(auto line : linesBySide[side]) {
52         while(curLines.size() > 0) {
53             Segment other = curLines.back();
54             if(cross(line.d(), other.d()) != 0) {
55                 long double start = intersection(line, other).y;
56                 if(start > apply.back()) break;
57             }
58             curLines.pop_back();
59             apply.pop_back();
60         }
61         if(curLines.size() == 0) {
62             apply.push_back(-HUGE_VALL);
63         } else {
64             apply.push_back(intersection(line, curLines.back()).y);
65         }
66         curLines.push_back(line);
67     }
68     linesBySide[side] = curLines;
69 }
70 applyStart[0].push_back(HUGE_VALL);
71 applyStart[1].push_back(HUGE_VALL);
72 long double result = 0;
73 {
74     long double lb = -HUGE_VALL, ub;
75     for(int i=0, j=0; i < (int)linesBySide[0].size() && j <
76         (int)linesBySide[1].size(); lb = ub) {
77         ub = min(applyStart[0][i+1], applyStart[1][j+1]);
78         long double alb = lb, aub = ub;
79         Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];
80         if(cross(l1.d(), l0.d()) > 0) {
81             alb = max(alb, intersection(l0, l1).y);
82         } else if(cross(l1.d(), l0.d()) < 0) {
83             aub = min(aub, intersection(l0, l1).y);
84         }
85         alb = max(alb, lowerbound);
86         aub = min(aub, upperbound);
87         aub = max(aub, alb);
88         {
89             long double x1 = l0.a.x + (alb - l0.a.y) / l0.d().y * l0.d().x;

```

#681

#144

#417

#994

#591

```

86     long double x2 = 10.a.x + (aub - 10.a.y) / 10.d().y * 10.d().x;
87     result -= (aub - alb) * (x1 + x2) / 2;
88 }
89 {
90     long double x1 = 11.a.x + (alb - 11.a.y) / 11.d().y * 11.d().x;
91     ↪ #346
92     long double x2 = 11.a.x + (aub - 11.a.y) / 11.d().y * 11.d().x;
93     result += (aub - alb) * (x1 + x2) / 2;
94 }
95 if(applyStart[0][i+1] < applyStart[1][j+1]) {
96     i++;
97 } else {
98     j++;
99 }
100 } #348
101 return result; %183
102 }

```

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 } %025
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, #423
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     }
35 } #082
36 for (int i = 0; i < res.size() - 1; ++i)

```

```

32     hull.emplace_back(res[i], res[i + 1]);
33 }
34 Hull(vector< pair< int, int > > &vert) { // at least 2 distinct
35     ↪ points
36     sort(vert.begin(), vert.end()); // 0(n log(n))
37     extend_hull(vert.begin(), vert.end());
38     int diff = hull.size();
39     extend_hull(vert.rbegin(), vert.rend());
40     upper_begin = hull.begin() + diff; %572
41 }
42 bool contains(pair< int, int > p) { // 0(log(n))
43     if (p < hull.front().first || p > upper_begin->first) return false;
44     {
45         auto it_low = lower_bound(hull.begin(), upper_begin,
46                                     make_pair(make_pair(p.first,
47                                     ↪ (int)-2e9), make_pair(0, 0)));
48         if (it_low != hull.begin())
49             --it_low;
50         auto v1 = make_pair(it_low->second.first - it_low->first.first,
51                             it_low->second.second -
52                             ↪ it_low->first.second);
53         auto v2 = make_pair(p.first - it_low->first.first, p.second -
54                             ↪ it_low->first.second); #248
55         if (cross(v1, v2) < 0) // < 0 is inclusive, <=0 is exclusive
56             return false;
57     }
58     {
59         auto it_up = lower_bound(hull.rbegin(), hull.rbegin() +
60                                 ↪ (hull.end() - upper_begin),
61                                 make_pair(make_pair(p.first, (int)2e9),
62                                 ↪ make_pair(0, 0)));
63         if (it_up - hull.rbegin() == hull.end() - upper_begin)
64             --it_up;
65         auto v1 = make_pair(it_up->first.first - it_up->second.first,
66                             it_up->first.second - it_up->second.second);
67         ↪ #392
68         auto v2 = make_pair(p.first - it_up->second.first, p.second -
69                             ↪ it_up->second.second);
70         if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
71             return false;
72     }
73     return true; %435
74 }
75 template < typename T > // The function can have only one local min
76     ↪ and max and may be constant
77     // only at min and max.
78     vector< pair< pair< int, int >, pair< int, int > > >::iterator max(
79         function< T(const pair< pair< int, int >, pair< int, int > > &) >
80         ↪ f) { // 0(log(n))
81         auto l = hull.begin();
82         auto r = hull.end();
83         vector< pair< pair< int, int >, pair< int, int > > >::iterator best
84         ↪ = hull.end();

```

```

74 T best_val;
75 while (r - l > 2) {
76     auto mid = l + (r - l) / 2;
77     T l_val = f(*l);
78     T l_nxt_val = f(*(l + 1));
79     T mid_val = f(*mid);
80     T mid_nxt_val = f(*(mid + 1));
81     if (best == hull.end() ||
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

```

#836

#650

#419

#675

#629

%671

```

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

```

#927

#295

%543

#146

```

162     }));
163     return best_seg->first;
164 }
165 vector< pair< pair< int, int >, pair< int, int > > >::iterator
166     ↪ max_in_dir(
167     pair< int, int > v) { // first is the ans. O(log(n))
168     return max(function< ll(const pair< pair< int, int >, pair< int,
169     ↪ int > > &) >(
170     [&v](const pair< pair< int, int >, pair< int, int > > &seg) {
171     ↪ return dot(v, seg.first); }));
172 }
173 pair< vector< pair< pair< int, int >, pair< int, int > > >::iterator,
174 vector< pair< pair< int, int >, pair< int, int > > >::iterator
175     ↪ >
176 intersections(pair< pair< int, int >, pair< int, int > > line) { //
177     ↪ O(log(n))
178     int x = line.second.first - line.first.first;
179     int y = line.second.second - line.first.second;
180     auto dir = make_pair(-y, x);
181     auto it_max = max_in_dir(dir);
182     auto it_min = max_in_dir(make_pair(y, -x));
183     ll opt_val = dot(dir, line.first);
184     if (dot(dir, it_max->first) < opt_val || dot(dir, it_min->first) >
185     ↪ opt_val)
186     return make_pair(hull.end(), hull.end());
187     vector< pair< pair< int, int >, pair< int, int > > >::iterator
188     ↪ it_r1, it_r2;
189     function< bool(const pair< pair< int, int >, pair< int, int > > &,
190     ↪ const pair< pair< int, int >, pair< int, int > > &)
191     ↪ >
192     inc_comp([&dir](const pair< pair< int, int >, pair< int, int >
193     ↪ > &lft,
194     ↪ const pair< pair< int, int >, pair< int, int >
195     ↪ > &rgt) {
196     ↪ return dot(dir, lft.first) < dot(dir, rgt.first);
197     });
198     function< bool(const pair< pair< int, int >, pair< int, int > > &,
199     ↪ const pair< pair< int, int >, pair< int, int > > &)
200     ↪ >
201     dec_comp([&dir](const pair< pair< int, int >, pair< int, int >
202     ↪ > &lft,
203     ↪ const pair< pair< int, int >, pair< int, int >
204     ↪ > &rgt) {
205     ↪ return dot(dir, lft.first) > dot(dir, rgt.first);
206     });
207     if (it_min <= it_max) {
208     it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
209     if (dot(dir, hull.front().first) >= opt_val) {
210     it_r2 = upper_bound(hull.begin(), it_min + 1, line, dec_comp) -
211     ↪ 1;
212     } else {
213     it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
214     }
215     } else {
216     it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
217     }
218     return make_pair(it_r1, it_r2);
219 }
220 pair< pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
221     pair< pair< int, int >, pair< int, int > > res;
222     ll dia_sq = 0;
223     auto it1 = hull.begin();
224     auto it2 = upper_begin();
225     auto v1 = make_pair(hull.back().second.first -
226     ↪ hull.back().first.first,
227     ↪ hull.back().second.second -
228     ↪ hull.back().first.second);
229     while (it2 != hull.begin()) {
230     auto v2 = make_pair((it2 - 1)->second.first - (it2 -
231     ↪ 1)->first.first,
232     ↪ (it2 - 1)->second.second - (it2 -
233     ↪ 1)->first.second);
234     ll decider = cross(v1, v2);
235     if (decider > 0) break;
236     --it2;
237 }
238 while (it2 != hull.end()) { // check all antipodal pairs
239     if (dist_sq(it1->first, it2->first) > dia_sq) {
240     res = make_pair(it1->first, it2->first);
241     dia_sq = dist_sq(res.first, res.second);
242     }
243     auto v1 =
244     ↪ make_pair(it1->second.first - it1->first.first,
245     ↪ it1->second.second - it1->first.second);
246     auto v2 =
247     ↪ make_pair(it2->second.first - it2->first.first,
248     ↪ it2->second.second - it2->first.second);
249     ll decider = cross(v1, v2);
250     if (decider == 0) { // report cross pairs at parallel lines.
251     if (dist_sq(it1->second, it2->first) > dia_sq) {
252     res = make_pair(it1->second, it2->first);
253     dia_sq = dist_sq(res.first, res.second);
254     }
255     if (dist_sq(it1->first, it2->second) > dia_sq) {
256     res = make_pair(it1->first, it2->second);
257     dia_sq = dist_sq(res.first, res.second);
258     }
259     ++it1;
260     ++it2;
261     } else if (decider < 0) {

```

```

201     } else {
202     it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
203     if (dot(dir, hull.front().first) <= opt_val) {
204     it_r2 = upper_bound(hull.begin(), it_max + 1, line, inc_comp) -
205     ↪ 1;
206     } else {
207     it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
208     }
209     }
210     return make_pair(it_r1, it_r2);
211 }
212 pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
213     pair< pair< int, int >, pair< int, int > > res;
214     ll dia_sq = 0;
215     auto it1 = hull.begin();
216     auto it2 = upper_begin();
217     auto v1 = make_pair(hull.back().second.first -
218     ↪ hull.back().first.first,
219     ↪ hull.back().second.second -
220     ↪ hull.back().first.second);
221     while (it2 != hull.begin()) {
222     auto v2 = make_pair((it2 - 1)->second.first - (it2 -
223     ↪ 1)->first.first,
224     ↪ (it2 - 1)->second.second - (it2 -
225     ↪ 1)->first.second);
226     ll decider = cross(v1, v2);
227     if (decider > 0) break;
228     --it2;
229 }
230 while (it2 != hull.end()) { // check all antipodal pairs
231     if (dist_sq(it1->first, it2->first) > dia_sq) {
232     res = make_pair(it1->first, it2->first);
233     dia_sq = dist_sq(res.first, res.second);
234     }
235     auto v1 =
236     ↪ make_pair(it1->second.first - it1->first.first,
237     ↪ it1->second.second - it1->first.second);
238     auto v2 =
239     ↪ make_pair(it2->second.first - it2->first.first,
240     ↪ it2->second.second - it2->first.second);
241     ll decider = cross(v1, v2);
242     if (decider == 0) { // report cross pairs at parallel lines.
243     if (dist_sq(it1->second, it2->first) > dia_sq) {
244     res = make_pair(it1->second, it2->first);
245     dia_sq = dist_sq(res.first, res.second);
246     }
247     if (dist_sq(it1->first, it2->second) > dia_sq) {
248     res = make_pair(it1->first, it2->second);
249     dia_sq = dist_sq(res.first, res.second);
250     }
251     ++it1;
252     ++it2;
253     } else if (decider < 0) {

```

#762

%112

#083

#107

#456

```

247     ++it1;
248 } else {
249     ++it2;
250 }
251 }
252 return res;
253 }
254 };
#543

8 Aho Corasick  $\mathcal{O}(|\alpha| \sum \text{len})$ 
%204

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){
#666
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){
#251
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]];
#697
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;
40     }
#791
41     return root;
42 }
%670
43 //auxiliary functions for searching and counting
44 node *walk(node *cur, char c){ //longest prefix in dict that is suffix
    of walked string.

```

```

45 while(true){
46     if(cur->nxt[c])
47         return cur->nxt[c];
48     if(!cur->suffix)
49         return cur;
50     cur = cur->suffix;
51 }
52 }
%570

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
    suffixes for final counts
61     vector<node *> to_visit = {root};
62     for(int i=0; i<to_visit.size(); ++i){
63         node *cur = to_visit[i];
64         for(int j=0; j<alpha_size; ++j){
65             if(cur->nxt[j])
66                 to_visit.push_back(cur->nxt[j]);
67         }
68     }
69     for(int i=to_visit.size()-1; i>0; --i)
#462
70         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
%657
71 }
72 int main(){
    //http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
73     int n, len;
74     scanf("%d %d", &n, &len);
75     vector<char> a(len+1);
76     scanf("%s", a.data());
77     a.pop_back();
78     for(char &c : a)
79         c -= 'a';
80     vector<vector<char> > dict(n);
81     for(int i=0; i<n; ++i){
82         scanf("%d", &len);
83         dict[i].resize(len+1);
84         scanf("%s", dict[i].data());
85         dict[i].pop_back();
86         for(char &c : dict[i])
87             c -= 'a';
88     }
89     node *root = aho_corasick(dict);
90     cnt_matches(root, a);
91     add_cnt(root);
92     for(int i=0; i<n; ++i){
93         node *cur = root;
94         for(char c : dict[i])

```

```

95     cur = walk(cur, c);
96     printf("%d\n", cur->cnt);
97 }
98 }



---


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

```

#388

#163

#239

#252

```

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
49     ↪ calculates number of ways to reach an end node.
50     if (!vis) { //paths_to_end is ocurence count for any strings in
51     ↪ current suffix equivalence class.
52         vis = true;
53         for (auto cur : nxt_char) {
54             cur.second->calc_paths_to_end();
55             paths_to_end += cur.second->paths_to_end;
56         }
57     }
58 };
59 struct SufAutomaton {
60     AutoNode *last;
61     AutoNode *root;
62     void extend(char new_c) {
63         AutoNode *new_end = new AutoNode;
64         new_end->len = last->len + 1;
65         AutoNode *suf_w_nxt = last;
66         while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {
67             suf_w_nxt->set_nxt(new_c, new_end);
68             suf_w_nxt = suf_w_nxt->suf;
69         }
70         if (!suf_w_nxt) {
71             new_end->suf = root;
72         } else {
73             AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);
74             if (suf_w_nxt->len + 1 == max_sbstr->len) {
75                 new_end->suf = max_sbstr;
76             } else {
77                 AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1,
78                 ↪ new_c);
79                 new_end->suf = eq_sbstr
80                 AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
81                 while (w_edge_to_eq_sbstr != 0 &&
82                 ↪ w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
83                     w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
84                     w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
85                 }
86             }
87         }
88         last = new_end;
89     }
90     SufAutomaton(string to_suffix) {
91         root = new AutoNode;
92         root->len = 0;
93         root->suf = NULL;
94         last = root;

```

#257

#914

#458

#550

#193

```

92     for (char c : to_suffix) extend(c);
93 }
94 };
                                     %227

10 Dinic
-----
1 struct MaxFlow{
2     typedef long long ll;
3     const ll INF = 1e18;
4     struct Edge{
5         int u,v;
6         ll c,rc;
7         shared_ptr<ll> flow;
8         Edge(int _u, int _v, ll _c, ll _rc =
9             ↪ 0):u(_u),v(_v),c(_c),rc(_rc){
10         }
11 };
                                     #787
12 struct FlowTracker{
13     shared_ptr<ll> flow;
14     ll cap, rcap;
15     bool dir;
16     FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
17         ↪ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
18     ll rem() const {
19         if(dir == 0){
20             return cap-*flow;
21         }
22         else{
23             return rcap+*flow;
24         }
25     }
26     void add_flow(ll f){
27         if(dir == 0)
28             *flow += f;
29         else
30             *flow -= f;
31         assert(*flow <= cap);
32         assert(-*flow <= rcap);
33     }
34     operator ll() const { return rem(); }
35     void operator--(ll x){ add_flow(x); }
36     void operator+=(ll x){ add_flow(-x); }
37 };
38 int source,sink;
39 vector<vector<int>> > adj;
40 vector<vector<FlowTracker>> > cap;
41 vector<Edge> edges;
42 MaxFlow(int _source, int _sink):source(_source),sink(_sink){ #080
43     assert(source != sink);
44 }
45 int add_edge(int u, int v, ll c, ll rc = 0){
46     edges.push_back(Edge(u,v,c,rc));
47     return edges.size()-1;
48 }
49 vector<int> now,lv1;

```

```

48 void prep(){
49     int max_id = max(source, sink);
50     for(auto edge : edges)
51         max_id = max(max_id, max(edge.u, edge.v));
52     adj.resize(max_id+1);
53     cap.resize(max_id+1);
54     now.resize(max_id+1);
55     lv1.resize(max_id+1);
56     for(auto &edge : edges){
57         auto flow = make_shared<ll>(0);
58         adj[edge.u].push_back(edge.v);
59         cap[edge.u].push_back(FlowTracker(edge.c, edge.rc, flow,
60             ↪ 0));
61         if(edge.u != edge.v){
62             adj[edge.v].push_back(edge.u);
63             cap[edge.v].push_back(FlowTracker(edge.c, edge.rc,
64                 ↪ flow, 1));
65         }
66         assert(cap[edge.u].back() == edge.c);
67         edge.flow = flow;
68     }
69 }
70 bool dinic_bfs(){
71     fill(now.begin(),now.end(),0);
72     fill(lv1.begin(),lv1.end(),0);
73     lv1[source] = 1;
74     vector<int> bfs(1,source);
75     for(int i = 0; i < bfs.size(); ++i){
76         int u = bfs[i];
77         for(int j = 0; j < adj[u].size(); ++j){
78             int v = adj[u][j];
79             if(cap[u][j] > 0 && lv1[v] == 0){
80                 lv1[v] = lv1[u]+1;
81                 bfs.push_back(v);
82             }
83         }
84     }
85     return lv1[sink] > 0;
86 }
87 ll dinic_dfs(int u, ll flow){
88     if(u == sink)
89         return flow;
90     while(now[u] < adj[u].size()){
91         int v = adj[u][now[u]];
92         if(lv1[v] == lv1[u] + 1 && cap[u][now[u]] != 0){ #014
93             ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
94             if(res > 0){
95                 cap[u][now[u]] -= res;
96                 return res;
97             }
98         }
99         ++now[u];
100     }
101 }

```

```

99     return 0;
100 }
101 ll calc_max_flow(){
102     prep();
103     ll ans = 0;
104     while(dinic_bfs()){
105         ll cur = 0;
106         do{
107             cur = dinic_dfs(source,INF);
108             ans += cur;
109         }while(cur > 0);
110     }
111     return ans;
112 }
113 ll flow_on_edge(int edge_index){
114     assert(edge_index < edges.size());
115     return *edges[edge_index].flow;
116 }
117 };
118 int main(){
119     int n,m;
120     cin >> n >> m;
121     auto mf = MaxFlow(1,n); // arguments source and sink, memory usage
122     // 0(largest node index + input size), sink doesn't need to be
123     // last index
124     int edge_index;
125     for(int i = 0; i < m; ++i){
126         int a,b,c;
127         cin >> a >> b >> c;
128         //mf.add_edge(a,b,c); // for directed edges
129         edge_index = mf.add_edge(a,b,c,c); // store edge index if care
130         // about flow value
131     }
132     cout << mf.calc_max_flow() << '\n';
133     //cout << mf.flow_on_edge(edge_index) << endl; // return flow on
134     // this edge
135 }

```

#197

#817

%583

11 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;

```

%230

```

14 ll sink_pot=0;
15 min_dist[0] = 0;
16 for(int i=1; i<=v; ++i){ //incase of no negative edges Bellman-Ford
17     // can be removed.
18     min_dist[i]=inf;
19 }
20 for(int i=0; i<v-1; ++i){
21     for(int j=0; j<v; ++j){
22         for(int k=0; k<v; ++k){
23             if(rem_flow[j][k] > 0 && min_dist[j]+cost[j][k] < min_dist[k])
24                 min_dist[k] = min_dist[j]+cost[j][k];
25         }
26     }
27 }
28 for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
29     for(int j=0; j<v; ++j){
30         if(cost[i][j]!=inf){
31             cost[i][j]+=min_dist[i];
32             cost[i][j]-=min_dist[j];
33         }
34     }
35 }
36 sink_pot+=min_dist[v-1]; //Bellman-Ford end.
37 while(true){
38     for(int i=0; i<=v; ++i){ //node after sink is used as start value
39         // for Dijkstra.
40         min_dist[i]=inf;
41         visited[i]=false;
42     }
43     min_dist[0]=0;
44     node_flow[0]=inf;
45     int min_node;
46     while(true){ //Use Dijkstra to calculate potentials
47         int min_node=v;
48         for(int i=0; i<v; ++i){
49             if(!visited[i] && min_dist[i]<min_dist[min_node])
50                 min_node=i;
51         }
52         if(min_node==v) break
53         visited[min_node]=true;
54         for(int i=0; i<v; ++i){
55             if(!visited[i] && min_dist[min_node]+cost[min_node][i] <
56                 min_dist[i]){
57                 min_dist[i]=min_dist[min_node]+cost[min_node][i];
58                 prev_node[i]=min_node;
59                 node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
60             }
61         }
62     }
63     if(min_dist[v-1]==inf) break
64     for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
65         for(int j=0; j<v; ++j){ //Found path from source to sink becomes
66             // 0 cost.

```

%655

#988

%412

#948

#413

```

63     if(cost[i][j] != inf){
64         cost[i][j] += min_dist[i];
65         cost[i][j] -= min_dist[j];
66     }
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];
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 }
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;

```

#323

#533

%265

```

117     min_cost_max_flow();
118     cout << tot_cost << '\n';
119 }
120 }

```

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

```

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 {

```

#042

#965

#534

#507

```

48     int p;
49     Edge* used;
50 };
51 while(1) {
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) {

```

#834

#554

#916

#455

#881

#554

```

101         result += edge->addFlow(cur, cap);
102         cur = edge->from(cur);
103     }
104 }
105     if(!valid) break;
106 }
107     return result;
108 }
109 };

```

%455

13 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            Node *con_to_root(){

```

#186

#119

#025

#644

```

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

```

#041

#646

#576

#594

#940

```

91 graph.resize(n);
92 cur_root = &graph[r];
93 for(auto &cur : desc) //Edges are reversed internally
94     edges.push_back(Edge(&graph[cur.to], &graph[cur.from], cur.w));
95 for(int i=0; i<desc.size(); ++i)
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;

```

#362

%600

%477

#711

%771

14 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;
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 int main(){ //Mechanics Practice BRIDGES
50     int n, m;
51     cin>>n>>m;
52     for(int i=0; i<m; ++i){
53         int u, v;
54         scanf("%d %d", &u, &v);
55         graph[u].con.emplace_back(graph+v);
56         graph[v].con.emplace_back(graph+u);
57         graph[u].con.back().rev = &graph[v].con.back();
58         graph[v].con.back().rev = &graph[u].con.back();
59     }
60     graph[1].dfs(1, NULL);

```

#955

#336

#232
%273

%106

%056

%223

```

61 int res = 0;
62 for(int i=1; i<=n; ++i)
63     res += graph[i].cnt_adj_bridges();
64 cout<<res/2<<endl;
65 }

```

15 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$

```

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){

```

#078

#346

#991

%603

```

46     vector<int> component;
47     _topsort_dfs(*it, component, explr, conn);
48     sort(component.begin(), component.end());
49     results.push_back(component);                                #522
50 }
51 sort(results.begin(), results.end());
52 return results;
53 }
54 };                                                                %362
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 }

```

16 Lazy Segment Tree $\mathcal{O}(\log n)$ per query

```

1 struct SegmentTree {
2     struct Node {
3         long long value = 0;
4         int size = 1;
5         int lazy_add = 0;
6         bool lazy_set = false;

```

```

7         int lazy_to_set = 0;
8         void set(int to_set) {
9             lazy_set = true;
10            lazy_to_set = to_set;
11            lazy_add = 0;
12        }
13    };
14    int n;
15    vector<Node> nodes;
16    void propagate(int pos) {
17        Node& cur = nodes[pos];
18        if(cur.lazy_set) {
19            if(pos < n) {
20                nodes[pos*2].set(cur.lazy_to_set);
21                nodes[pos*2+1].set(cur.lazy_to_set);
22            }
23            cur.value = 1LL * cur.size * cur.lazy_to_set;
24            cur.lazy_set = false;
25        }
26        if(cur.lazy_add != 0) {
27            if(pos < n) {
28                nodes[pos*2].lazy_add += cur.lazy_add;
29                nodes[pos*2+1].lazy_add += cur.lazy_add;
30            }
31            cur.value += 1LL * cur.size * cur.lazy_add;
32            cur.lazy_add = 0;
33        }
34    }
35    long long get_value(int pos) {
36        propagate(pos);
37        return nodes[pos].value;
38    }
39    SegmentTree(int nsize) {
40        n = 1;
41        while(n < nsize) n*=2;
42        nodes.resize(2*n);
43        for(int i=n-1; i>0; i--)
44            nodes[i].size = nodes[2*i].size * 2;
45    }
46    void set(int l, int r, int to_set, int pos = 1, int lb = 0, int rb
47    ↪ = -1) {
48        propagate(pos);
49        if(rb == -1) rb = n;
50        if(l <= lb && rb <= r) {
51            nodes[pos].set(to_set);
52            return;
53        }
54        int mid = (lb + rb) / 2;
55        if(l < mid)
56            set(l, r, to_set, pos*2, lb, mid);
57        if(mid < r)
58            set(l, r, to_set, pos*2+1, mid, rb);
59        nodes[pos].value = get_value(pos*2) + get_value(pos*2+1);

```

#173

#388

#114

#759

#567

```

59 }
60 void add(int l, int r, int to_add, int pos = 1, int lb = 0, int rb
    ↪ = -1) { #168
61     propagate(pos);
62     if(rb == -1) rb = n;
63     if(l <= lb && rb <= r) {
64         nodes[pos].lazy_add += to_add;
65         return;
66     }
67     int mid = (lb + rb) / 2;
68     if(l < mid)
69         add(l, r, to_add, pos*2, lb, mid);
70     if(mid < r) #620
71         add(l, r, to_add, pos*2+1, mid, rb);
72     nodes[pos].value = get_value(pos*2) + get_value(pos*2+1);
73 }
74 long long get(int l, int r, int pos = 1, int lb = 0, int rb = -1) {
75     propagate(pos);
76     if(rb == -1) rb = n;
77     if(l <= lb && rb <= r) return get_value(pos);
78     int mid = (lb + rb) / 2;
79     long long result = 0;
80     if(l < mid) #133
81         result += get(l, r, pos*2, lb, mid);
82     if(mid < r)
83         result += get(l, r, pos*2+1, mid, rb);
84     return result;
85 }
86 }; #280
87 //Solution for:
88 ↪ http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/LAZY
89 int main() {
90     int n, m;
91     cin >> n >> m;
92     SegmentTree stree(n);
93     for(int i=0; i<n; i++) {
94         int a;
95         cin >> a;
96         stree.set(i, i+1, a);
97     }
98     for(int i=0; i<m; i++) {
99         int type;
100         cin >> type;
101         if(type == 1) {
102             int l, r, d;
103             cin >> l >> r >> d;
104             stree.add(l-1, r, d);
105         } else if(type == 2) {
106             int l, r, x;
107             cin >> l >> r >> x;
108             stree.set(l-1, r, x);
109         } else {
110             int l, r;
111             cin >> l >> r;

```

```

111         cout << stree.get(l-1, r) << '\n';
112     }
113 }
114 }

```

17 Generic segment tree(lazy, noncommutative)

```

1 struct Segment{
2     ll sum_val=0;
3     ll min_val=0;
4     void find_sum(int seg_len, ll &cur_sum){
5         cur_sum = cur_sum + sum_val;
6     }
7     void find_min(int seg_len, ll &cur_min){
8         cur_min = min(cur_min, min_val);
9     }
10    void recalc(int seg_len, const Segment &lhs_seg, const Segment
    ↪ &rhs_seg){ #599
11        sum_val = lhs_seg.sum_val + rhs_seg.sum_val;
12        min_val = min(lhs_seg.min_val, rhs_seg.min_val);
13    }
14 };
15 struct Lazy{
16     ll add_val;
17     ll assign_val; //LLONG_MIN if no assign;
18     void init(){
19         add_val = 0;
20         assign_val = LLONG_MIN; #237
21     }
22     Lazy(){ init(); }
23     void apply_to_lazy(int seg_len, Lazy &child) const{
24         if(assign_val != LLONG_MIN){
25             child.add_val = 0;
26             child.assign_val = assign_val;
27         }
28         child.add_val += add_val;
29     }
30     void apply_to_seg(int seg_len, Segment &cur) const{ #242
31         if(assign_val != LLONG_MIN){
32             cur.min_val = assign_val;
33             cur.sum_val = seg_len * assign_val;
34         }
35         cur.min_val += add_val;
36         cur.sum_val += seg_len * add_val;
37     } //Following code should not need to be modified %047
38     void split(int seg_len, Lazy &lhs_lazy, Lazy &rhs_lazy){
39         apply_to_lazy(seg_len, lhs_lazy); //Empty current and pass on to
    ↪ children
40         apply_to_lazy(seg_len, rhs_lazy);
41         init();
42     }
43 };
44 // Highly optimized generic segment tree with lazy propagation
45 class SegTree{ //indexes start from 0, ranges are [beg, end)

```

```

46 private:
47     int offset;
48     int height;
49     Segment *segs;
50     Lazy *lazys;
51     vector<bool> is_lazy;
52     void split(int len, int idx){
53         is_lazy[idx] = false;
54         lazys[idx].apply_to_seg(len/2, segs[2*idx]);
55         lazys[idx].apply_to_seg(len/2, segs[2*idx+1]);
56         lazys[idx].split(len/2, lazys[2*idx], lazys[2*idx+1]);
57         is_lazy[2*idx] = true;
58         is_lazy[2*idx+1] = true;
59     }
60     void push(int bot_idx){
61         for(int s = height-1; s>0; --s){
62             int idx = bot_idx>>s;
63             if(is_lazy[idx]){ //Lazys can be below other lazys
64                 split(1<<s, idx);
65             }
66         }
67     }
68     void build(int len, int idx){
69         for(; idx; len<=1, idx>>=1){
70             segs[idx].recalc(len, segs[2*idx], segs[2*idx+1]);
71         }
72     }
73 public:
74     SegTree(int tree_size){
75         offset = tree_size;
76         height = 32 - __builtin_clz(tree_size);
77         segs = new Segment[2*tree_size];
78         lazys = new Lazy[2*tree_size];
79         is_lazy.resize(2*tree_size, false);
80     }
81     ~SegTree(){
82         delete[] segs;
83         delete[] lazys;
84     }
85     void modify(int l, int r, const Lazy &upd){
86         l+=offset;
87         r+=offset;
88         push(l);
89         push(r-1);
90         int len = 1;
91         for(int l_tmp = l, r_tmp = r; l_tmp<r_tmp; l_tmp >= 1, r_tmp >=
            ↳ 1, len <= 1){
92             if(l_tmp & 1){
93                 upd.apply_to_lazy(len, lazys[l_tmp]);
94                 upd.apply_to_seg(len, segs[l_tmp]);
95                 is_lazy[l_tmp] = true;
96                 ++l_tmp;
97             }
98             if(r_tmp & 1){

```

#311

#376

#434

#042

#843

#341

```

99         --r_tmp;
100         upd.apply_to_lazy(len, lazys[r_tmp]);
101         upd.apply_to_seg(len, segs[r_tmp]);
102         is_lazy[r_tmp] = true;
103     }
104 }
105 len = 1<<(__builtin_ctz(l)+1);
106 l >= __builtin_ctz(l) + 1;
107 build(len, l);
108 len = 1<<(__builtin_ctz(r)+1);
109 r >= __builtin_ctz(r) + 1;
110 build(len, r);
111 }
112 template< typename ...QueryArgs >
113 void query(int l, int r, void (Segment::*query_func)(int,
            ↳ QueryArgs...), QueryArgs &&...query_args){
114     l+=offset;
115     r+=offset;
116     push(l);
117     push(r-1);
118     int len = 1;
119     int r_orig = r;
120     for(; l<r; l>=1, r>=1, len <= 1){ //Segments applied in order
            ↳ to query
121         if(l & 1){
122             (segs[l++].*query_func)(len, query_args...);
123         }
124     }
125     for(;r < r_orig;){
126         r<=1;
127         len>=1;
128         if(r_orig & len){
129             (segs[r++].*query_func)(len, query_args...);
130         }
131     }
132 }
133 };
134 int main(){
135     int n, m; //solves Mechanics Practice LAZY
136     cin>>n>>m;
137     SegTree seg_tree(n);
138     for(int i=0; i<n; ++i){
139         Lazy tmp;
140         scanf("%lld", &tmp.assign_val);
141         seg_tree.modify(i, i+1, tmp);
142     }
143     for(int i=0; i<m; ++i){
144         int o;
145         int l, r;
146         scanf("%d %d %d", &o, &l, &r);
147         --l;
148         if(o==1){
149             Lazy tmp;

```

#872

#475

#784

%891

```

150     scanf("%lld", &tmp.add_val);
151     seg_tree.modify(l, r, tmp);
152 } else if(o==2){
153     Lazy tmp;
154     scanf("%lld", &tmp.assign_val);
155     seg_tree.modify(l, r, tmp);
156 } else {
157     ll res=0;
158     seg_tree.query(l, r, &Segment::find_sum, res);
159     printf("%lld\n",res);
160 }
161 }
162 }

```

18 Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query

```

1 template<typename T, typename comp>
2 class PersistentST {
3     struct Node {
4         Node *left, *right;
5         int lend, rend;
6         T value;
7         Node (int position, T _value) {
8             left = NULL;
9             right = NULL;
10            lend = position;
11            rend = position;
12            value = _value;
13        }
14        Node (Node *_left, Node *_right) {
15            left = _left;
16            right = _right;
17            lend = left->lend;
18            rend = right->rend;
19            value = comp()(left->value, right->value);
20        }
21        T query (int qlleft, int qright) {
22            qlleft = max(qlleft, lend);
23            qright = min(qright, rend);
24            if (qlleft == lend && qright == rend) {
25                return value;
26            } else if (qlleft > qright) {
27                return comp().identity;
28            } else {
29                return comp()(left->query(qlleft, qright), right->query(qlleft,
30                    qright));
31            }
32        };
33        int size;
34        Node **tree;
35        vector<Node*> roots;
36    public:
37        PersistentST () {}
38        PersistentST (int _size, T initial) {

```

#479

#373

#766

```

39     for (int i = 0; i < 32; i++) {
40         if ((1 << i) > _size) {
41             size = 1 << i;
42             break;
43         }
44     }
45     tree = new Node* [2 * size + 5];
46     for (int i = size; i < 2 * size; i++)
47         tree[i] = new Node (i - size, initial);
48     for (int i = size - 1; i > 0; i--)
49         tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
50     roots = vector<Node*> (1, tree[1]);
51 }
52 void set (int position, T _value) {
53     tree[size + position] = new Node (position, _value);
54     for (int i = (size + position) / 2; i >= 1; i /= 2)
55         tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
56     roots.push_back(tree[1]);
57 }
58 int last_revision () {
59     return (int) roots.size() - 1;
60 }
61 T query (int qlleft, int qright, int revision) {
62     return roots[revision]->query(qlleft, qright);
63 }
64 T query (int qlleft, int qright) {
65     return roots[last_revision()]->query(qlleft, qright);
66 }
67 };

```

#250

#128

#890

%280

19 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
12    * the
13    * group operation. It should have the following methods:
14    * * DS () - empty constructor
15    * * DS (int size, T initial) - constructs the structure with the given
16    *   size,
17    *   initially filled with initial.
18    * * void set (int index, T value) - set the value at index `index` to
19    *   `value`
20    * * T query (int left, int right) - return the "sum" of elements
21    *   between left and right, inclusive.
22    */
23 template<typename T, class DS>

```

%932

```

20 class HLD {
21     int vertexc;
22     vector<int> *adj;
23     vector<int> subtree_size;
24     DS structure;
25     DS aux;
26     void build_sizes (int vertex, int parent) {
27         subtree_size[vertex] = 1;
28         for (int child : adj[vertex]) {
29             if (child != parent) {
30                 build_sizes(child, vertex);
31                 subtree_size[vertex] += subtree_size[child];
32             }
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);
70         ord = vector<int> (vertexc + 5);
71         chain_root = vector<int> (vertexc + 5);
72         par = vector<int> (vertexc + 5);
73         cur = 0;

```

#037

#593

#646

#738

#841

```

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 };
107 int main () {
108     int vertexc;
109     cin >> vertexc;
110     HLD<int, dummy> hld (vertexc);
111     for (int i = 0; i < vertexc - 1; i++) {
112         int u, v;
113         cin >> u >> v;
114         hld.add_edge(u, v);
115     }
116     hld.build(0);
117     hld.print();
118     int queryc;
119     cin >> queryc;
120     for (int i = 0; i < queryc; i++) {
121         int u, v;
122         cin >> u >> v;
123         hld.query_path(u, v);
124         cout << endl;
125     }
126 }

```

#793

#219

%515

20 Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ 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
47 template < typename internal_type >
48 struct wrapped {
49     internal_type a = 0;
50     void update(internal_type b) {
51         a += b;

```

#774

#919

#522

#677

```

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     }
72     fenwick.initiate();
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 }

```

#391

%330

21 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        }
18        void update() {
19            total = value;
20            if(lch) total += lch->total;

```

#698

#295

```

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) {
72         return sum_upto(key, cur->lch);
73     } else {
74         long long result = cur->value + sum_upto(key, cur->rch);

```

#233

#230

#510

#760

#634

```

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     int m;
89     Treap treap;
90     cin >> m;
91     for(int i=0; i<m; i++) {
92         int type;
93         cin >> type;
94         if(type == 1) {
95             int x, y;
96             cin >> x >> y;
97             treap.insert(x, y);
98         } else if(type == 2) {
99             int x;
100             cin >> x;
101             treap.erase(x);
102         } else {
103             int l, r;
104             cin >> l >> r;
105             cout << treap.get(l, r) << endl;
106         }
107     }
108     return 0;

```

#509

%959

22 FFT $\mathcal{O}(n \log(n))$

```

1 //Assumes a is a power of two
2 vector<complex<long double>> fastFourierTransform(vector<complex<long
3     double>> a, bool inverse) {
4     const long double PI = acos(-1.0L);
5     int n = a.size();
6     //Precalculate w
7     vector<complex<long double>> w(n, 0.0L);
8     w[0] = 1;
9     for(int tpow = 1; tpow < n; tpow *= 2)
10         w[tpow] = polar(1.0L, 2*PI * tpow/n * (inverse ? -1 : 1));
11     for(int i=3, last = 2; i<n; i++) {
12         if(w[i] == 0.0L) {
13             w[i] = w[last] * w[i-last];
14         } else {
15             last = i;

```

#086

```

16 }
17 //Rearrange a
18 for(int block = n; block > 1; block /= 2) {
19     int half = block/2;
20     vector<complex<long double>> na(n);
21     for(int s=0; s < n; s += block) {
22         for(int i=0;i<block;i++)
23             na[s + half*(i%2) + i/2] = a[s+i];
24     }
25     a = na;
26 }
27 //Now do the calculation
28 for(int block = 2; block <= n; block *= 2) {
29     vector<complex<long double>> na(n);
30     int wb = n/block, half = block/2;
31     for(int s=0; s < n; s += block) {
32         for(int i=0;i<half; i++) {
33             na[s+i] = a[s+i] + w[wb*i] * a[s+half+i];
34             na[s+half+i] = a[s+i] - w[wb*i] * a[s+half+i];
35         }
36     }
37     a = na;
38 }
39 return a;
40 }
41 struct Polynomial {
42     vector<long double> a;
43     long double& operator[](int ind) {
44         return a[ind];
45     }
46     Polynomial& operator*=(long double r) {
47         for(auto &c : a)
48             c *= r;
49         return *this;
50     }
51     Polynomial operator*(long double r) {return Polynomial(*this) *= r;}
52     Polynomial& operator/=(long double r) {
53         for(auto &c : a)
54             c /= r;
55         return *this;
56     }
57     Polynomial operator/(long double r) {return Polynomial(*this) /= r;}
58     Polynomial& operator+=(Polynomial r) {
59         if(a.size() < r.a.size())
60             a.resize(r.a.size(), 0.0L);
61         for(int i=0;i<(int)r.a.size();i++)
62             a[i] += r[i];
63         return *this;
64     }
65     Polynomial operator+(Polynomial r) {return Polynomial(*this) += r;}
66     Polynomial& operator-=(Polynomial r) {
67         if(a.size() < r.a.size())
68             a.resize(r.a.size(), 0.0L);
69         for(int i=0;i<(int)r.a.size();i++)

```

#092

#515

#447

#663

#015

```

70     a[i] -= r[i];
71     return *this;
72 }
73 Polynomial operator-(Polynomial r) {return Polynomial(*this) -= r;}
74 Polynomial operator*(Polynomial r) {
75     int n = 1;
76     while(n < (int)(a.size() + r.a.size() - 1) )
77         n *= 2;
78     vector<complex<long double>> fl(n, 0.0L), fr(n, 0.0L);
79     for(int i=0;i<(int)a.size();i++)
80         fl[i] = a[i];
81     for(int i=0;i<(int)r.a.size();i++)
82         fr[i] = r[i];
83     fl = fastFourierTransform(fl, false);
84     fr = fastFourierTransform(fr, false);
85     vector<complex<long double>> ret(n);
86     for(int i=0;i<n;i++)
87         ret[i] = fl[i] * fr[i];
88     ret = fastFourierTransform(ret, true);
89     Polynomial result;
90     result.a.resize(a.size() + r.a.size() - 1);
91     for(int i=0;i<(int)result.a.size();i++)
92         result[i] = ret[i].real() / n;
93     return result;
94 }
95 };

```

#623

#077

#228

%196

23 MOD int, extended Euctclidean

```

1 pair<int, int> extendedEuclideanAlgorithm(int a, int b) {
2     if(b == 0)
3         return make_pair(1, 0);
4     pair<int, int> ret = extendedEuclideanAlgorithm(b, a%b);
5     return {ret.second, ret.first - a/b * ret.second};
6 }
7 struct Modint {
8     static const int MOD = 1000000007;
9     int val;
10    Modint(int nval = 0) {
11        val = nval;
12    }
13    Modint& operator+=(Modint r) {
14        val = (val + r.val) % MOD;
15        return *this;
16    }
17    Modint operator+(Modint r) {return Modint(*this) += r;}
18    Modint& operator-=(Modint r) {
19        val = (val + MOD - r.val) % MOD;
20        return *this;
21    }
22    Modint operator-(Modint r) {return Modint(*this) -= r;}
23    Modint& operator*=(Modint r) {
24        val = 1LL * val * r.val % MOD;
25        return *this;

```

#412

#052

```

26 }
27 Modint operator*(Modint r) {return Modint(*this) *= r;}
28 Modint inverse() {
29     int ret = extendedEuclideanAlgorithm(val, MOD).first;
30     if(ret < 0)
31         ret += MOD;
32     return ret;
33 }
34 Modint& operator/=(Modint r) {
35     return operator*=(r.inverse() );
36 }
37 Modint operator/(Modint r) {return Modint(*this) /= r;}
38 };

```

#985
%567

24 Rabin Miller prime check

```

1 __int128 pow_mod(__int128 a, ll n, __int128 mod) {
2     __int128 res = 1;
3     for (ll i = 0; i < 64; ++i) {
4         if (n & (1LL << i))
5             res = (res * a) % mod;
6         a = (a * a) % mod;
7     }
8     return res;
9 }
10 bool is_prime(ll n) { //guaranteed for 64 bit numbers
11     if (n == 2 || n == 3) return true;
12     if (!(n & 1) || n == 1) return false;
13     static vector< char > witnesses = {2, 3, 5, 7, 11, 13, 17, 19, 23,
14         ↪ 29, 31, 37};
15     ll s = __builtin_ctz(n - 1);
16     ll d = (n - 1) >> s;
17     __int128 mod = n;
18     for (__int128 a : witnesses) {
19         if (a >= mod) break;
20         a = pow_mod(a, d, mod);
21         if (a == 1 || a == mod - 1) continue;
22         for (ll r = 1; r < s; ++r) {
23             a = a * a % mod;
24             if (a == 1) return false;
25             if (a == mod - 1) break;
26         }
27         if (a != mod - 1) return false;
28     }
29     return true;

```

#406
#398
%043

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
```