

Contents

1	Setup	1
2	crc.sh	1
3	gcc ordered set	2
4	Numerical integration with Simpson's rule	2
5	Triangle centers	2
6	2D line segment	3
7	Convex polygon algorithms	5
8	Aho Corasick $\mathcal{O}(\alpha \sum \text{len})$	8
9	Suffix automaton $\mathcal{O}((n+q) \log(\alpha))$	10
10	Dinic	11
11	Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$	13
12	Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$	14
13	Lazy Segment Tree $\mathcal{O}(\log n)$ per query	16
14	Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query	18
15	Templated HLD $\mathcal{O}(M(n) \log n)$ per query	18
16	Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query	20
17	Treap $\mathcal{O}(\log n)$ per query	21
18	FFT $\mathcal{O}(n \log(n))$	23
19	MOD int, extended Euctclidean	24
20	Rabbin Miller prime check	25
21	2-Sat $\mathcal{O}(n)$	25

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 #whitespaces 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, tree_order_statistics_node_update>;
9 int main(){
10     ordered_set<int> cur;
11     cur.insert(1);
12     cur.insert(3);
13     cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
14     cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
15     cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
16 }
```

#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;
11    return integrate(f,a,m,computing_power/2)+integrate(f,m,b,computing_power/2);
12 }
```

#300

%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 the real line
11    return ang < min_delta;
12 }
13 point circum_center(){
14     if(collinear())
15         return point(NAN,NAN);
16     //squared lengths of sides
17     double a2, b2, c2;
18     a2 = norm(B - C);
19     b2 = norm(A - C);
20     c2 = norm(A - B);
21     //barycentric coordinates of the circumcenter
22     double c_A, c_B, c_C;
23     c_A = a2 * (b2 + c2 - a2); //sin(2 * alpha) may be used as well
24     c_B = b2 * (a2 + c2 - b2);
25     c_C = c2 * (a2 + b2 - c2);
26     double sum = c_A + c_B + c_C;
27     c_A /= sum;
28     c_B /= sum;
29     c_C /= sum;
30     // cartesian coordinates of the circumcenter
31     return c_A * A + c_B * B + c_C * C;
32 }
33 point centroid(){ //center of mass
34     return (A + B + C) / 3.0;
35 }
36 point ortho_center(){ //euler line
37     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();
```

#647

%029

#688

%561

```

42 return 0 + 1.5 * (centroid() - 0);
43 };
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;
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 }

```

#530
%132

#812

%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 };
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 }

```

#054

#673

#724

#872

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

```

#355

#009

#597

#681

#144

#417

#994

#591

```

83     aub = max(aub, alb);
84     {
85         long double x1 = 10.a.x + (alb - 10.a.y) / 10.d().y * 10.d().x;
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         long double x2 = 11.a.x + (aub - 11.a.y) / 11.d().y * 11.d().x;
92         result += (aub - alb) * (x1 + x2) / 2;
93     }
94     if(applyStart[0][i+1] < applyStart[1][j+1]) {
95         i++;
96     } else {
97         j++;
98     }
99 }
100 }
101 return result;
102 }

```

#346

#348

%183

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 upper_begin;
14     template < typename Iterator >
15     void extend_hull(Iterator begin, Iterator end) { // O(n)
16         vector< pair< int, int > > res;
17         for (auto it = begin; it != end; ++it) {
18             if (res.empty() || *it != res.back()) {
19                 while (res.size() >= 2) {
20                     auto v1 = make_pair(res[res.size() - 1].first - res[res.size() - 2].first,
21                                         res[res.size() - 1].second - res[res.size() - 2].second);
22                     auto v2 = make_pair(it->first - res[res.size() - 2].first,
23                                         it->second - res[res.size() - 2].second);
24                     if (cross(v1, v2) > 0)
25                         break;
26                     res.pop_back();
27                 }
28                 res.push_back(*it);
29             }
30         }
31         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 points
35         sort(vert.begin(), vert.end()); // O(n log(n))
36         extend_hull(vert.begin(), vert.end());
37         int diff = hull.size();
38         extend_hull(vert.rbegin(), vert.rend());
39         upper_begin = hull.begin() + diff;
40     }
41     bool contains(pair< int, int > p) { // O(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, (int)-2e9), make_pair(0, 0)));
46             if (it_low != hull.begin())
47                 --it_low;
48             auto v1 = make_pair(it_low->second.first - it_low->first.first,
49                                 it_low->second.second - it_low->first.second);
49             auto v2 = make_pair(p.first - it_low->first.first, p.second - it_low->first.second);
50             if (cross(v1, v2) < 0) // < 0 is inclusive, <= 0 is exclusive
51                 return false;
52         }

```

#423

#082

%572

#248

```

53     }
54     {
55         auto it_up = lower_bound(hull.rbegin(), hull.rbegin() + (hull.end() - upper_begin),
56                                 make_pair(make_pair(p.first, (int)2e9), make_pair(0, 0)));
57         if (it_up - hull.rbegin() == hull.end() - upper_begin)
58             --it_up;
59         auto v1 = make_pair(it_up->first.first - it_up->second.first,
60                             it_up->first.second - it_up->second.second);
61         auto v2 = make_pair(p.first - it_up->second.first, p.second - it_up->second.second);
62         if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
63             return false;
64     }
65     return true;
66 }
67 template < typename T > // The function can have only one local min and max and may be constant
68                         // only at min and max.
69 vector< pair< pair< int, int >, pair< int, int > > >::iterator max(
70     function< T(const pair< pair< int, int >, pair< int, int > &) > f) { // O(log(n))
71     auto l = hull.begin();
72     auto r = hull.end();
73     vector< pair< pair< int, int >, pair< int, int > > >::iterator best = 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 the range.
83             best = l;
84             best_val = l_val;
85         }
86         if (l_nxt_val > l_val) {
87             if (mid_val < l_val) {
88                 r = mid;
89             } else {
90                 if (mid_nxt_val > mid_val) {
91                     l = mid + 1;
92                 } else {
93                     r = mid + 1;
94                 }
95             }
96         } else {
97             if (mid_val < l_val) {
98                 l = mid + 1;
99             } else {
100                 if (mid_nxt_val > mid_val) {
101                     l = mid + 1;
102                 } else {
103                     r = mid + 1;
104                 }
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 closest(
123     pair< int, int >
124     p) { // p can't be internal(can be on border), hull must have atleast 3 points
125     const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))

```

```

126     return max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
127         [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
128             &seg) { // accuracy of used type should be coord-2
129             if (p == seg.first) return 10 - M_PI;
130             auto v1 =
131                 make_pair(seg.second.first - seg.first.first, seg.second.second - seg.first.second); #927
132             auto v2 = make_pair(p.first - seg.first.first, p.second - seg.first.second);
133             ll cross_prod = cross(v1, v2);
134             if (cross_prod > 0) { // order the backside by angle
135                 auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
136                 auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
137                 ll dot_prod = dot(v1, v2);
138                 ll cross_prod = cross(v2, v1);
139                 return atan2(cross_prod, dot_prod) / 2;
140             }
141             ll dot_prod = dot(v1, v2); #295
142             double res = atan2(dot_prod, cross_prod);
143             if (dot_prod <= 0 && res > 0) res = -M_PI;
144             if (res > 0) {
145                 res += 20;
146             } else {
147                 res = 10 - res;
148             }
149             return res;
150         }));
151     } #543
152     pair< int, int > forw_tan(pair< int, int > p) { // can't be internal or on border
153         const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))
154         auto best_seg = max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
155             [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
156                 &seg) { // accuracy of used type should be coord-2
157                 auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
158                 auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
159                 ll dot_prod = dot(v1, v2);
160                 ll cross_prod = cross(v2, v1); // cross(v1, v2) for backian!!!
161                 return atan2(cross_prod, dot_prod); // order by signed angle #146
162             }));
163         return best_seg->first;
164     } #658
165     vector< pair< pair< int, int >, pair< int, int > > >::iterator max_in_dir(
166         pair< int, int > v) { // first is the ans. O(log(n))
167         return max(function< ll(const pair< pair< int, int >, pair< int, int > > &) >(
168             [&v](const pair< pair< int, int >, pair< int, int > > &seg) { return dot(v, seg.first); }));
169     }
170     pair< vector< pair< pair< int, int >, pair< int, int > > >::iterator,
171         vector< pair< pair< int, int >, pair< int, int > > >::iterator > #543
172     intersections(pair< pair< int, int >, pair< int, int > > line) { // O(log(n))
173         int x = line.second.first - line.first.first;
174         int y = line.second.second - line.first.second;
175         auto dir = make_pair(-y, x);
176         auto it_max = max_in_dir(dir);
177         auto it_min = max_in_dir(make_pair(y, -x));
178         ll opt_val = dot(dir, line.first);
179         if (dot(dir, it_max->first) < opt_val || dot(dir, it_min->first) > opt_val)
180             return make_pair(hull.end(), hull.end());
181         vector< pair< pair< int, int >, pair< int, int > > >::iterator it_r1, it_r2; #627
182         function< bool(const pair< pair< int, int >, pair< int, int > > &,
183             const pair< pair< int, int >, pair< int, int > > &) >
184             inc_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
185                 const pair< pair< int, int >, pair< int, int > > &rgt) {
186                 return dot(dir, lft.first) < dot(dir, rgt.first);
187             });
188         function< bool(const pair< pair< int, int >, pair< int, int > > &,
189             const pair< pair< int, int >, pair< int, int > > &) >
190             dec_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
191                 const pair< pair< int, int >, pair< int, int > > &rgt) { #440
192                 return dot(dir, lft.first) > dot(dir, rgt.first);
193             });
194         if (it_min <= it_max) {
195             it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
196             if (dot(dir, hull.front().first) >= opt_val) {
197                 it_r2 = upper_bound(hull.begin(), it_min + 1, line, dec_comp) - 1;
198             } else {

```



```

199     it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
200 }
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) - 1;
205     } else {
206         it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
207     }
208 }
209 return make_pair(it_r1, it_r2);
210 }
211 pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
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 - hull.back().first.first,
217                         hull.back().second.second - hull.back().first.second);
218     while (it2 != hull.begin()) {
219         auto v2 = make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
220                             (it2 - 1)->second.second - (it2 - 1)->first.second);
221         ll decider = cross(v1, v2);
222         if (decider > 0) break;
223         --it2;
224     }
225     while (it2 != hull.end()) { // check all antipodal pairs
226         if (dist_sq(it1->first, it2->first) > dia_sq) {
227             res = make_pair(it1->first, it2->first);
228             dia_sq = dist_sq(res.first, res.second);
229         }
230         auto v1 =
231             make_pair(it1->second.first - it1->first.first, it1->second.second - it1->first.second);
232         auto v2 =
233             make_pair(it2->second.first - it2->first.first, it2->second.second - it2->first.second);
234         ll decider = cross(v1, v2);
235         if (decider == 0) { // report cross pairs at parallel lines.
236             if (dist_sq(it1->second, it2->first) > dia_sq) {
237                 res = make_pair(it1->second, it2->first);
238                 dia_sq = dist_sq(res.first, res.second);
239             }
240             if (dist_sq(it1->first, it2->second) > dia_sq) {
241                 res = make_pair(it1->first, it2->second);
242                 dia_sq = dist_sq(res.first, res.second);
243             }
244             ++it1;
245             ++it2;
246         } else if (decider < 0) {
247             ++it1;
248         } else {
249             ++it2;
250         }
251     }
252     return res;
253 }
254 };

```

#762

%112

#083

#107

#456

#543

%204

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){
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){

```

#666


```

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){
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]];
31                     break;
32                 }
33                 suf=suf->suffix;
34             }
35             if(cur.first->size() > i+1)
36                 nxt_state.push_back(cur);
37         }
38     }
39     cur_state=nxt_state;
40 }
41 return root;
42 }
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 }
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 }
60 void add_cnt(node *root){ //After counting matches propagate 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)
70         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
71 }
72 int main(){ //http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
73     int n, len;
74     scanf("%d %d", &len, &n);
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);

```

#251

#697

#791

%670

%570

%286

#462

%657

```

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 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) {
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
33            ↪ substring
34            match_len = min(match_len, len); //includes depth limit(needed for finding matches)
35            if (has_nxt(c)) { //as suffixes are in classes match_len must be tracked externally
36                ++match_len;
37                return nxt(c)->lower_depth(depth);
38            }
39            if (suf)
40                return suf->walk(c, depth, match_len);
41            return this;
42        }
43        int paths_to_end = 0;
44        void set_as_end() { //All suffixes of current node are marked as ending nodes.
45            paths_to_end = 1;
46            if (suf) suf->set_as_end();
47        }
48        bool vis = false;
49        void calc_paths_to_end() { //Call ONCE from ROOT. For each node calculates number of ways to reach an end
50            ↪ node.
51            if (!vis) { //paths_to_end is ocurence count for any strings in 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) {

```

#388

#163

#239

#252

#257

#914

```

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 {
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, new_c);
77             new_end->suf = eq_sbstr;
78             AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
79             while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
80                 w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
81                 w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
82             }
83         }
84     }
85     last = new_end;
86 }
87 SufAutomaton(string to_suffix) {
88     root = new AutoNode;
89     root->len = 0;
90     root->suf = NULL;
91     last = root;
92     for (char c : to_suffix) extend(c);
93 }
94 };

```

#458

#550

#193

%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 = 0):u(_u),v(_v),c(_c),rc(_rc){
9             }
10    };
11    struct FlowTracker{
12        shared_ptr<ll> flow;
13        ll cap, rcap;
14        bool dir;
15        FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
16        ↪ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
17        ll rem() const {
18            if(dir == 0){
19                return cap-*flow;
20            }
21            else{
22                return rcap-*flow;
23            }
24        }
25        void add_flow(ll f){
26            if(dir == 0)
27                *flow += f;
28            else
29                *flow -= f;
30            assert(*flow <= cap);
31            assert(-*flow <= rcap);
32        }
33        operator ll() const { return rem(); }
34        void operator+=(ll x){ add_flow(x); }
35        void operator+=(ll x){ add_flow(-x); }
36    };
37    int source,sink;
38    vector<vector<int>> > adj;
39    vector<vector<FlowTracker>> > cap;

```

#787

#844

#287

```

39     vector<Edge> edges;
40     MaxFlow(int _source, int _sink):source(_source),sink(_sink){
41         assert(source != sink);
42     }
43     int add_edge(int u, int v, ll c, ll rc = 0){
44         edges.push_back(Edge(u,v,c,rc));
45         return edges.size()-1;
46     }
47     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,0));
60             if(edge.u != edge.v){
61                 adj[edge.v].push_back(edge.u);
62                 cap[edge.v].push_back(FlowTracker(edge.c,edge.rc,flow,1));
63             }
64             assert(cap[edge.u].back() == edge.c);
65             edge.flow = flow;
66         }
67     }
68     bool dinic_bfs(){
69         fill(now.begin(),now.end(),0);
70         fill(lv1.begin(),lv1.end(),0);
71         lv1[source] = 1;
72         vector<int> bfs(1,source);
73         for(int i = 0; i < bfs.size(); ++i){
74             int u = bfs[i];
75             for(int j = 0; j < adj[u].size(); ++j){
76                 int v = adj[u][j];
77                 if(cap[u][j] > 0 && lv1[v] == 0){
78                     lv1[v] = lv1[u]+1;
79                     bfs.push_back(v);
80                 }
81             }
82         }
83         return lv1[sink] > 0;
84     }
85     ll dinic_dfs(int u, ll flow){
86         if(u == sink)
87             return flow;
88         while(now[u] < adj[u].size()){
89             int v = adj[u][now[u]];
90             if(lv1[v] == lv1[u] + 1 && cap[u][now[u]] != 0){
91                 ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
92                 if(res > 0){
93                     cap[u][now[u]] -= res;
94                     return res;
95                 }
96             }
97             ++now[u];
98         }
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;

```

#080

#328

#717

#038

#010

#014

#197

#817

```

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 O(largest node index + input size), sink
    ↪ doesn't need to be last index
122     int edge_index;
123     for(int i = 0; i < m; ++i){
124         int a,b,c;
125         cin >> a >> b >> c;
126         //mf.add_edge(a,b,c); // for directed edges
127         edge_index = mf.add_edge(a,b,c,c); // store edge index if care about flow value
128     }
129     cout << mf.calc_max_flow() << '\n';
130     //cout << mf.flow_on_edge(edge_index) << endl; // return flow on this edge
131 }

```

%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 y.
5 ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO inf IF NOT USED
6 ll min_dist[nmax];
7 int prev_node[nmax];
8 ll node_flow[nmax];
9 bool visited[nmax];
10 ll tot_cost, tot_flow; //output
11 void min_cost_max_flow(){
12     tot_cost=0; //Does not work with negative cycles.
13     tot_flow=0;
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 can be removed.
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 for Dijkstra.
38             min_dist[i]=inf;
39             visited[i]=false;
40         }
41         min_dist[0]=0;
42         node_flow[0]=inf;
43         int min_node;
44         while(true){ //Use Dijkstra to calculate potentials
45             int min_node=v;
46             for(int i=0; i<v; ++i){
47                 if((!visited[i]) && min_dist[i]<min_dist[min_node])
48                     min_node=i;
49             }
50             if(min_node==v) break
51             visited[min_node]=true;

```

%230

%655

#988

%412

#948

```

52     for(int i=0; i<v; ++i){
53         if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){
54             min_dist[i]=min_dist[min_node]+cost[min_node][i];
55             prev_node[i]=min_node;
56             node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
57         }
58     }
59 }
60 if(min_dist[v-1]==inf) break
61 for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
62     for(int j=0; j<v; ++j){ //Found path from source to sink becomes 0 cost.
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;
117         min_cost_max_flow();
118         cout<<tot_cost<<'\\n';
119     }
120 }

```

#413

#323

#533

%265

12 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) {
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] ? edge->cost : -edge->cost);
60                             int ni = edge->from(&nodes[i])->index;
61                             if(np < state[lev+1][ni].p) {
62                                 state[lev+1][ni].p = np;
63                                 state[lev+1][ni].used = edge;
64                             }
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;

```

#042

#965

#534

#507

#834

#554

#916


```

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

```

#455

#881

#554

%455

13 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) {

```

#173

#388

#114

#759

#567

#168

#620

#133

%280

```

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 = -1) {
47     propagate(pos);
48     if(rb == -1) rb = n;
49     if(l <= lb && rb <= r) {
50         nodes[pos].set(to_set);
51         return;
52     }
53     int mid = (lb + rb) / 2;
54     if(l < mid)
55         set(l, r, to_set, pos*2, lb, mid);
56     if(mid < r)
57         set(l, r, to_set, pos*2+1, mid, rb);
58     nodes[pos].value = get_value(pos*2) + get_value(pos*2+1);
59 }
60 void add(int l, int r, int to_add, int pos = 1, int lb = 0, int rb = -1) {
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)
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)
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 };
87 //Solution for: http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/LAZY
88 int main() {
89     int n, m;
90     cin >> n >> m;
91     SegmentTree stree(n);
92     for(int i=0; i<n; i++) {
93         int a;
94         cin >> a;
95         stree.set(i, i+1, a);
96     }
97     for(int i=0; i<m; i++) {
98         int type;
99         cin >> type;
100         if(type == 1) {
101             int l, r, d;
102             cin >> l >> r >> d;
103             stree.add(l-1, r, d);
104         } else if(type == 2) {
105             int l, r, x;
106             cin >> l >> r >> x;
107             stree.set(l-1, r, x);
108         } else {
109             int l, r;
110             cin >> l >> r;
111             cout << stree.get(l-1, r) << '\n';
112         }

```

113 }
114 }

14 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 qlen, int rright) {
22            qlen = max(qlen, lend);
23            rright = min(rright, rend);
24            if (qlen == lend && rright == rend) {
25                return value;
26            } else if (qlen > rright) {
27                return comp().identity;
28            } else {
29                return comp()(left->query(qlen, rright), right->query(qlen, rright));
30            }
31        }
32    };
33    int size;
34    Node **tree;
35    vector<Node*> roots;
36 public:
37     PersistentST () {}
38     PersistentST (int _size, T initial) {
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 qlen, int rright, int revision) {
62         return roots[revision]->query(qlen, rright);
63     }
64     T query (int qlen, int rright) {
65         return roots[last_revision()]->query(qlen, rright);
66     }
67 };

```

#479

#373

#766

#250

#128

#890

%280

15 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`
17  * * T query (int left, int right) - return the "sum" of elements between left and right, inclusive.
18  */
19 template<typename T, class DS>
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;
74         build_sizes(1, -1);
75         build_hld(1, -1, 1);

```

%932

#037

#593

#646

#738

#841

```

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     int cur_id = 0;
84     while (chain_root[u] != chain_root[v]) {
85         if (ord[u] > ord[v]) {
86             cur_id++;
87             aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
88             u = par[chain_root[u]];
89         } else {
90             cur_id++;
91             aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
92             v = par[chain_root[v]];
93         }
94     }
95     cur_id++;
96     aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
97     return aux.query(1, cur_id);
98 }
99 void print () {
100     for (int i = 1; i <= vertexc; i++)
101         cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' << par[i] << endl;
102 }
103 };
104 int main () {
105     int vertexc;
106     cin >> vertexc;
107     HLD<int, dummy> hld (vertexc);
108     for (int i = 0; i < vertexc - 1; i++) {
109         int u, v;
110         cin >> u >> v;
111         hld.add_edge(u, v);
112     }
113     hld.build(0);
114     hld.print();
115     int queryc;
116     cin >> queryc;
117     for (int i = 0; i < queryc; i++) {
118         int u, v;
119         cin >> u >> v;
120         hld.query_path(u, v);
121         cout << endl;
122     }
123 }

```

#793

#219

%515

16 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()) - positions.begin());
20             elems.resize(positions.size());
21         }
22     }
23     template < typename... loc_form >
24     void update(coord_t cord, loc_form... args) {

```

#774

#919

```

25     if (initiated) {
26         int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
27         for (; pos < positions.size(); pos += pos & -pos)
28             elems[pos].update(args...);
29     } else {
30         positions.push_back(cord);
31     }
32 }
33 template < typename... loc_form >
34 ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
35     ret_t res = 0;
36     int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
37     for (; pos > 0; pos -= pos & -pos)
38         res += elems[pos].query(args...);
39     return res;
40 }
41 };
42 template < typename internal_type >
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 > fenwick;
62     int dim = 2;
63     vector< tuple< int, int, ll > > to_insert;
64     to_insert.emplace_back(1, 1, 1);
65     // set up all positions that are to be used for update
66     for (int i = 0; i < dim; ++i) {
67         for (auto &cur : to_insert)
68             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include value which won't be used
69         fenwick.initiate();
70     }
71     // actual use
72     for (auto &cur : to_insert)
73         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
74     cout << fenwick.query(2, 2)<<'\n';
75 }

```

#522

#677

#391

%330

17 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;
21            if(rch) total += rch->total;

```

#698

#295

```

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);
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: http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP
84 int main() {
85     ios_base::sync_with_stdio(false);
86     cin.tie(0);
87     int m;
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;

```

#233

#230

#510

#760

#634

#509

%959


```

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 }

```

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

```

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

```

#086

#092

#515

#447

#663

```

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++)
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 };

```

#015

#623

#077

#228

%196

19 MOD int, extended Euclidean

```

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;
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());

```

#412

#052

#985

```

36 }
37 Modint operator/(Modint r) {return Modint(*this) /= r;}
38 };

```

%567

20 Rabbin 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, 29, 31, 37};
14     ll s = __builtin_ctz(n - 1);
15     ll d = (n - 1) >> s;
16     __int128 mod = n;
17     for (__int128 a : witnesses) {
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 }

```

#406

#398

%043

21 2-Sat $\mathcal{O}(n)$

```

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> &explr, vector<vector<int> > &revconn) {
12        if (explr[pos])
13            return;
14        explr[pos] = true;
15        for (auto next : revconn[pos])
16            _topsort_dfs(next, result, explr, revconn);
17        result.push_back(pos);
18    }
19    vector<int> topsort() {
20        vector<vector<int> > revconn(n);
21        for (int u = 0; u < n; u++) {
22            for (auto v : conn[u])
23                revconn[v].push_back(u);
24        }
25        vector<int> result;
26        vector<bool> explr(n, false);
27        for (int i = 0; i < n; i++)
28            _topsort_dfs(i, result, explr, revconn);
29        reverse(result.begin(), result.end());
30        return result;
31    }
32    void dfs(int pos, vector<int> &result, vector<bool> &explr) {
33        if (explr[pos])
34            return;
35        explr[pos] = true;
36        for (auto next : conn[pos])
37            dfs(next, result, explr);
38        result.push_back(pos);

```

#078

#346

#991

```
39     }
40 };
41 //Solution for: http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
42 int main() {
43     ios_base::sync_with_stdio(false);
44     cin.tie(0);
45     int n, m;
46     cin >> n >> m;
47     Graph g(2*m);
48     for(int i=0; i<n; i++) {
49         int a, sa, b, sb;
50         cin >> a >> sa >> b >> sb;
51         a--, b--;
52         g.add_edge(2*a + 1 - sa, 2*b + sb);
53         g.add_edge(2*b + 1 - sb, 2*a + sa);
54     }
55     vector<int> state(2*m, 0);
56     {
57         vector<int> order = g.topsort();
58         vector<bool> explr(2*m, false);
59         for(auto u : order) {
60             vector<int> traversed;
61             g.dfs(u, traversed, explr);
62             if(traversed.size() > 0 && !state[traversed[0]^1]) {
63                 for(auto c : traversed)
64                     state[c] = 1;
65             }
66         }
67     }
68     for(int i=0; i < m; i++) {
69         if(state[2*i] == state[2*i+1]) {
70             cout << "IMPOSSIBLE\n";
71             return 0;
72         }
73     }
74     for(int i=0; i < m; i++) {
75         cout << state[2*i+1] << '\n';
76     }
77     return 0;
78 }
```

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

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

$$[n+1_k] = n[n_k] + [n_{k-1}]$$

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

$\{n_k\}$ — Stirling numbers of the second kind; number of partitions of set $1, \dots, n$ into k disjoint subsets.

$$\{n+1_k\} = k\{n_k\} + \{n_{k-1}\}$$

$$\sum_{k=0}^n \{n_k\} x^k = x^n$$

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

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

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 $A(x) = \sum_{n=0}^{\infty} a_n x^n / n!$

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

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

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

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
```