

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 Templatized Persistent Segment Tree $\mathcal{O}(\log n)$ per query
- 19 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query
- 20 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query
- 21 Treap $\mathcal{O}(\log n)$ per query
- 22 FFT $\mathcal{O}(n \log(n))$

23 MOD int, extended Euclidean

24 Rabin Miller prime check

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
2 "target remote | vgdb

```

2 crc.sh

```

1 #!/bin/env bash
4 starts=($(sed '/^\s*$/d' $1 | grep -n "//\!start" | cut -f1 -d:))
3 finishes=($(sed '/^\s*$/d' $1 | grep -n "//\!finish" | cut -f1 -d:))
7 for ((i=0;i<${#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)) | \
7       head -$j | tr -d '[[:space:]]' | cksum | cut -f1 -d ' ' | tail -c
      4
8   done #whitespace 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

```

12 #include <bits/stdc++.h>
13 typedef long long ll;
14 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>;
15
16 int main(){
17   ordered_set<int> cur;                                #221
18   cur.insert(1);
19   cur.insert(3);
20   cout << cur.order_of_key(2) << endl; // the number of elements in the
      set less than 2
21   cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in
      the set(0-based)
22   cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in
      the set(0-based)
23 }

```

%626

22

23

University of Tartu

#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
12 }
```

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; // vertexes 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
11    return ang < min_delta;
12 }                                                       %029
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;                                         #688
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;                           %561
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 0 + 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);
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;                               #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;                                         #054
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) {                                         #673
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 }



---


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; #009
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); #597
31         } else {
32             linesBySide[0].push_back({line.b, line.a});
33         }
34     }
35     sort(linesBySide[0].begin(), linesBySide[0].end(), [] (Segment l,
36         Segment r) {
37         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a >
38             normal(r.d())*r.a;
39     });
40     sort(linesBySide[1].begin(), linesBySide[1].end(), [] (Segment l,
41         Segment r) {
42         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a <
43             normal(r.d())*r.a;
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(); #144
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; #417
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;
90             }
91         }
92     }
93 
```

```

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
101return 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);                                %025
10}
11 struct Hull {
12     vector< pair< int, int >, pair< int, int > >> hull;
13     vector< pair< int, int >, pair< int, int > >>::iterator
14     ↵ upper_begin;
15     template < typename Iterator >
16     void extend_hull(Iterator begin, Iterator end) { // O(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);                                         #082
32             }
33         }
34     }
35     Hull(vector< pair< int, int > > &vert) { // atleast 2 distinct
36         ↵ points
37         sort(vert.begin(), vert.end());                                // O(n log(n))
38         extend_hull(vert.begin(), vert.end());
39         int diff = hull.size();
38         extend_hull(vert.rbegin(), vert.rend());
39         upper_begin = hull.begin() + diff;                            %572
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,
46                                     ↵ (int)-2e9), make_pair(0, 0)));
47     if (it_low != hull.begin())
48         --it_low;
49     auto v1 = make_pair(it_low->second.first - it_low->first.first,
50                         it_low->second.second -
51                         ↵ it_low->first.second);
52     auto v2 = make_pair(p.first - it_low->first.first, p.second -
53                         ↵ it_low->first.second);                                #248
54     if (cross(v1, v2) < 0) // < 0 is inclusive, <=0 is exclusive
55         return false;
56     }
57     auto it_up = lower_bound(hull.rbegin(), hull.rbegin() +
58                               (hull.end() - upper_begin),
59                               make_pair(make_pair(p.first, (int)2e9),
60                                     ↵ make_pair(0, 0)));
61     if (it_up - hull.rbegin() == hull.end() - upper_begin)
62         --it_up;
63     auto v1 = make_pair(it_up->first.first - it_up->second.first,
64                         it_up->first.second - it_up->second.second);
65     if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
66         return false;
67     }
68     return true;                                              %435
69     template < typename T > // The function can have only one local min
70     ↵ and max and may be constant
71     // only at min and max.
72     vector< pair< pair< int, int >, pair< int, int > > >::iterator max(
73         function< T(const pair< pair< int, int >, pair< int, int > > &) >
74         ↵ f) { // O(log(n))
75         auto l = hull.begin();
76         auto r = hull.end();
77         vector< pair< pair< int, int >, pair< int, int > > >::iterator best
78         ↵ = hull.end();
```

```

74 T best_val;
75 while (r - l > 2) {
76     auto mid = l + (r - 1) / 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         } 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        T l_val = f(*l);
108        if (best == hull.end() || l_val > best_val) {
109            best = l;
110            best_val = l_val;
111        }
112        if (r - l > 1) {
113            T l_nxt_val = f(*(l + 1));
114            if (best == hull.end() || l_nxt_val > best_val) {
115                best = l + 1;
116                best_val = l_nxt_val;
117            }
118        }
119    }
120    return best;
121}
122vector<pair<pair<int, int>, pair<int, int>>>::iterator
123    closest(
124        pair<int, int>
125        p) { // p can't be internal(can be on border), hull must
126        // have atleast 3 points
127        #836
128        const pair<pair<int, int>, pair<int, int>> &ref_p =
129        → hull.front(); // O(log(n))
130        return max(function<double(const pair<pair<int, int>, pair<
131        → int, int>> &) >(
132        [&p, &ref_p](const pair<pair<int, int>, pair<int, int>>
133        &seg) { // accuracy of used type should be
134        coord-2
135        if (p == seg.first) return 10 - M_PI;
136        auto v1 =
137            make_pair(seg.second.first - seg.first.first,
138            → seg.second.second - seg.first.second); #927
139        auto v2 = make_pair(p.first - seg.first.first, p.second -
140            → seg.first.second);
141        ll cross_prod = cross(v1, v2);
142        if (cross_prod > 0) { // order the backside by angle
143            auto v1 = make_pair(ref_p.first.first - p.first,
144            → ref_p.first.second - p.second);
145            auto v2 = make_pair(seg.first.first - p.first,
146            → seg.first.second - p.second);
147            ll dot_prod = dot(v1, v2);
148            ll cross_prod = cross(v2, v1);
149            return atan2(cross_prod, dot_prod) / 2;
150        }
151        ll dot_prod = dot(v1, v2); #295
152        double res = atan2(dot_prod, cross_prod);
153        if (dot_prod <= 0 && res > 0) res = -M_PI;
154        if (res > 0) {
155            res += 20;
156        } else {
157            res = 10 - res;
158        }
159        return res;
160    });
161    #543
162    pair<int, int> forw_tan(pair<int, int> p) { // can't be internal
163        or on border
164        const pair<pair<int, int>, pair<int, int>> &ref_p =
165        → hull.front(); // O(log(n))
166        auto best_seg = max(function<double(const pair<pair<int, int>,
167        → pair<int, int>> &) >(
168        [&p, &ref_p](const pair<pair<int, int>, pair<int, int>>
169        &seg) { // accuracy of used type should be
170        coord-2
171        auto v1 = make_pair(ref_p.first.first - p.first,
172            → ref_p.first.second - p.second);
173        auto v2 = make_pair(seg.first.first - p.first,
174            → seg.first.second - p.second);
175        ll dot_prod = dot(v1, v2);
176        ll cross_prod = cross(v2, v1); // cross(v1, v2) for
177        backtan!!!
178        return atan2(cross_prod, dot_prod); // order by signed
179        → angle
180    });
181    #146
182}

```

```

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    ↵ > %543
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; #627
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); #440
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     }

```

```

} else {
    it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
    if (dot(dir, hull.front().first) <= opt_val) {
        it_r2 = upper_bound(hull.begin(), it_max + 1, line, inc_comp) -
            ↪ 1;
    } else {
        it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
    }
}
return make_pair(it_r1, it_r2);
}
pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
pair< pair< int, int >, pair< int, int > > res;
ll dia_sq = 0;
auto it1 = hull.begin();
auto it2 = upper_begin;
auto v1 = make_pair(hull.back().second.first -
    ↪ hull.back().first.first,
                    hull.back().second.second -
    ↪ hull.back().first.second);
while (it2 != hull.begin()) {
    auto v2 = make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
                        (it2 - 1)->second.second - (it2 - 1)->first.second);
    ll decider = cross(v1, v2);
    if (decider > 0) break;
    --it2;
}
while (it2 != hull.end()) { // check all antipodal pairs
    if (dist_sq(it1->first, it2->first) > dia_sq) {
        res = make_pair(it1->first, it2->first);
        dia_sq = dist_sq(res.first, res.second);
    }
    auto v1 =
        make_pair(it1->second.first - it1->first.first,
                  ↪ it1->second.second - it1->first.second);
    auto v2 =
        make_pair(it2->second.first - it2->first.first,
                  ↪ it2->second.second - it2->first.second);
    ll decider = cross(v1, v2);
    if (decider == 0) { // report cross pairs at parallel lines.
        if (dist_sq(it1->second, it2->first) > dia_sq) {
            res = make_pair(it1->second, it2->first);
            dia_sq = dist_sq(res.first, res.second);
        }
        if (dist_sq(it1->first, it2->second) > dia_sq) { #456
            res = make_pair(it1->first, it2->second);
            dia_sq = dist_sq(res.first, res.second);
        }
    }
    ++it1;
    ++it2;
} else if (decider < 0) {
}
#762
%112
#083
#107
#456

```

```

247     ++it1;
248 } else {
249     ++it2;
250 }
251 }
252 return res;
253 }
#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){ #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){
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             cur_state=nxt_state;
39         }
40     }
41     return root;
42 }
#791
%670


---


//auxiliary functions for searching and counting
43 node *walk(node *cur, char c){ //longest prefix in dict that is suffix
    // of walked string.
#463
%570
44
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 }
#286
53 void cnt_matches(node *root, vector<char> &match_in){ #570
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) #462
70         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
71 }
#657
72 int main(){ #462
    //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);
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) { #388
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) { #163
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; #239
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; #252
46 }
47 int paths_to_end = 0;
48 void set_as_end() { //All suffixes of current node are marked as
49   ↳ ending nodes.
50 }
51
52
53
54
55
56
57
58 struct SufAutomaton {
59   AutoNode *last;
60   AutoNode *root; #914
61   void extend(char new_c) {
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; #458
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; #550
75       } else {
76         AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1,
77           ↳ new_c);
78         new_end->suf = eq_sbstr
79         AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
80         while (w_edge_to_eq_sbstr != 0 &&
81           ↳ w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
82           w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
83           w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
84         }
85       }
86     }
87   }
88   SufAutomaton(string to_suffix) {
89     root = new AutoNode;
90     root->len = 0;
91     root->suf = NULL;
92     last = root;
93   }

```

```

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


---


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    }#787
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 → _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){}
16     ll rem() const {
17         if(dir == 0){
18             return cap-*flow;
19         }
20         else{
21             return rcap**flow;
22         }
23     }
24     void add_flow(ll f){
25         if(dir == 0)
26             *flow += f;
27         else
28             *flow -= f;
29         assert(*flow <= cap);
30         assert(-*flow <= rcap);
31     }
32     operator ll() const { return rem(); }
33     void operator-=(ll x){ add_flow(x); }
34     void operator+=(ll x){ add_flow(-x); }
35 };
36     int source,sink;
37     vector<vector<int>> adj;
38     vector<vector<FlowTracker>> cap;
39     vector<Edge> edges;
40     MaxFlow(int _source, int _sink):source(_source),sink(_sink){ #080
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,lvl;

```

```

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         lvl.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){ #717
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         bool dinic_bfs(){
70             fill(now.begin(),now.end(),0);
71             fill(lvl.begin(),lvl.end(),0);
72             lvl[source] = 1;
73             vector<int> bfs(1,source);
74             for(int i = 0; i < bfs.size(); ++i){
75                 int u = bfs[i];
76                 for(int j = 0; j < adj[u].size(); ++j){
77                     int v = adj[u][j];
78                     if(cap[u][j] > 0 && lvl[v] == 0){
79                         lvl[v] = lvl[u]+1;
80                         bfs.push_back(v);
81                     }
82                 }
83             }
84             return lvl[sink] > 0;
85         }
86         ll dinic_dfs(int u, ll flow){
87             if(u == sink)
88                 return flow;
89             while(now[u] < adj[u].size()){
90                 int v = adj[u][now[u]];
91                 if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){ #014
92                     ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
93                     if(res > 0){
94                         cap[u][now[u]] -= res;
95                         return res;
96                     }
97                 }
98             }
99             ++now[u];
}

```

```

99     return 0;
100 }
111 ll calc_max_flow(){                                #197
101     prep();
102     ll ans = 0;
103     while(dinic_bfs()){
104         ll cur = 0;
105         do{
106             cur = dinic_dfs(source, INF);
107             ans += cur;
108         }while(cur > 0);
109     }
110     return ans;
111 }
112 ll flow_on_edge(int edge_index){
113     assert(edge_index < edges.size());
114     return *edges[edge_index].flow;
115 }
116 };
117 int main(){
118     int n,m;
119     cin >> n >> m;
120     auto mf = MaxFlow(1,n); // arguments source and sink, memory usage
121     → O(largest node index + input size), sink doesn't need to be
122     → last index
123     int edge_index;
124     for(int i = 0; i < m; ++i){
125         int a,b,c;
126         cin >> a >> b >> c;
127         //mf.add_edge(a,b,c); // for directed edges
128         edge_index = mf.add_edge(a,b,c,c); // store edge index if care
129         → about flow value
130     }
131     cout << mf.calc_max_flow() << '\n';
132     //cout << mf.flow_on_edge(edge_index) << endl; // return flow on
133     → this edge
134 }

```

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(){           %230
12     tot_cost=0;
13     tot_flow=0;
14     //Does not work with negative cycles.

```

```

14     ll sink_pot=0;
15     min_dist[0] = 0;
16     for(int i=1; i<=v; ++i){ //in case 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. %412
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[min_node]<min_dist[i]
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             if(min_dist[v-1]==inf) break;
63             for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
64                 for(int j=0; j<v; ++j){ //Found path from source to sink becomes
65                     → 0 cost.
66             }

```

```

63     if(cost[i][j]!=inf){
64         cost[i][j]+=min_dist[i];
65         cost[i][j]-=min_dist[j];
66     }
67 }
68 sink_pot+=min_dist[v-1];
69 tot_flow+=node_flow[v-1];
70 tot_cost+=sink_pot*node_flow[v-1];
71 int cur=v-1;
72 while(cur!=0){ //Backtrack along found path that now has 0 cost.
73     rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
74     rem_flow[cur][prev_node[cur]]+=node_flow[v-1];           #533
75     cost[cur][prev_node[cur]]=0;
76     if(rem_flow[prev_node[cur]][cur]==0)
77         cost[prev_node[cur]][cur]=inf;
78     cur=prev_node[cur];
79 }
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 }

```

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;                                #534
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] ?
60                 -edge->cost : edge->cost);
61               int ni = edge->from(&nodes[i])->index;          #554
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     }  

71     //Now look at the last level
72     bool valid = false;
73     for(int i=0;i<n;i++)
74       if(state[n-1][i].p > state[n][i].p) {
75         valid = true;
76         vector<Edge*> path;
77         int cap = 1000000000;
78         Node* cur = &nodes[i];
79         int clev = n;
80         vector<bool> expr(n, false);
81         while(!expr[cur->index]) {                      #455
82           expr[cur->index] = true;
83           State cstate = state[clev][cur->index];
84           cur = cstate.used->from(cur);
85           path.push_back(cstate.used);
86         }
87         reverse(path.begin(), path.end());
88       }
89       int i=0;
90       Node* cur2 = cur;
91       do {                                              #881
92         cur2 = path[i]->from(cur2);
93         i++;
94       } while(cur2 != cur);
95       path.resize(i);
96     }
97     for(auto edge : path) {
98       cap = min(cap, edge->getCap(cur));
99       cur = edge->from(cur);
100    }
101   for(auto edge : path) {                                #554
102
103 }
```

#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;                                #119
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(){#205
29      if(!par)
30        return this;
31      while(par->par)
32        par = par->par;
33      return par;
34    }
35    void clean(){#205
36      if(!no_dmst){
37        in_use = false;
38        for(auto &cur : out_cands)
39          to_process[cur.first].clean(cur.second);
40      }
41    }
42    Node *con_to_root(){#644
43      if(!par)
44        return this;
45      while(par->par)
46        par = par->par;
47      return par;
48    }
49  };
50  void clean(){#205
51    if(!no_dmst){
52      for(auto &cur : to_process)
53        cur.clean();
54    }
55  }
56  void update(){#205
57    for(auto &cur : to_process)
58      cur.update();
59  }
60  void print(){#205
61    for(auto &cur : to_process)
62      cur.print();
63  }
64  void print(){#205
65    for(auto &cur : to_process)
66      cur.print();
67  }
68  void print(){#205
69    for(auto &cur : to_process)
70      cur.print();
71  }
72  void print(){#205
73    for(auto &cur : to_process)
74      cur.print();
75  }
76  void print(){#205
77    for(auto &cur : to_process)
78      cur.print();
79  }
80  void print(){#205
81    for(auto &cur : to_process)
82      cur.print();
83  }
84  void print(){#205
85    for(auto &cur : to_process)
86      cur.print();
87  }
88  void print(){#205
89    for(auto &cur : to_process)
90      cur.print();
91  }
92  void print(){#205
93    for(auto &cur : to_process)
94      cur.print();
95  }
96  void print(){#205
97    for(auto &cur : to_process)
98      cur.print();
99  }
100 }
```

#186

#119

#205

#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         } #41
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(); #646
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(); #576
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); #594
84         }
85     }
86     par = super; //root or anc() of first Node encountered in a loop
87     return super;
88 }
89 Node *cur_root;
90 vector<Node> graph;
91 vector<Edge> edges;
92 DMST(int n, vector<EdgeDesc> &desc, int r){ //Self loops and multiple
93     ↵ edges are okay. #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(){ #362
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(){ #600
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){ #477
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; #711
134

```

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 vert graph[nmax];
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	15 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$	#078	#346	#991	%603	
61	int res = 0;							61	int res = 0;				
62	for(int i=1; i<=n; ++i)							62	for(int i=1; i<=n; ++i)				
63	res += graph[i].cnt_adj_bridges();							63	res += graph[i].cnt_adj_bridges();				
64	cout<<res/2<<endl;							64	cout<<res/2<<endl;				
65	}							65 }					
								15 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$					
1	struct Graph {							1	struct Graph {				
2	int n;							2	int n;				
3	vector<vector<int> > conn;							3	vector<vector<int> > conn;				
4	Graph(int nsiz) {							4	Graph(int nsiz) {				
5	n = nsiz;							5	n = nsiz;				
6	conn.resize(n);							6	conn.resize(n);				
7	}							7	}				
8	void add_edge(int u, int v) {							8	void add_edge(int u, int v) {				
9	conn[u].push_back(v);							9	conn[u].push_back(v);				
10	}							10	}				
11	void _topsort_dfs(int pos, vector<int> &result, vector<bool>							11	void _topsort_dfs(int pos, vector<int> &result, vector<bool>				
12	~> &expr, vector<vector<int> > &revconn) {							12	~> &expr, vector<vector<int> > &revconn) {				
13	if(expr[pos])							13	if(expr[pos])				
14	return;							14	return;				
15	expr[pos] = true;							15	expr[pos] = true;				
16	for(auto next : revconn[pos])							16	for(auto next : revconn[pos])				
17	_topsort_dfs(next, result, expr, revconn);							17	_topsort_dfs(next, result, expr, revconn);				
18	result.push_back(pos);							18	result.push_back(pos);				
19	}							19	}				
20	vector<int> topsort() {							20	vector<int> topsort() {				
21	vector<vector<int> > revconn(n);							21	vector<vector<int> > revconn(n);				
22	for(int u = 0; u < n; u++) {							22	for(int u = 0; u < n; u++) {				
23	for(auto v : conn[u])							23	for(auto v : conn[u])				
24	revconn[v].push_back(u);							24	revconn[v].push_back(u);				
25	}							25	}				
26	vector<int> result;							26	vector<int> result;				
27	vector<bool> expr(n, false);							27	vector<bool> expr(n, false);				
28	for(int i=0; i < n; i++)							28	for(int i=0; i < n; i++)				
29	_topsort_dfs(i, result, expr, revconn);							29	_topsort_dfs(i, result, expr, revconn);				
30	reverse(result.begin(), result.end());							30	reverse(result.begin(), result.end());				
31	return result;							31	return result;				
32	void dfs(int pos, vector<int> &result, vector<bool> &expr) {							32	void dfs(int pos, vector<int> &result, vector<bool> &expr) {				
33	if(expr[pos])							33	if(expr[pos])				
34	return;							34	return;				
35	expr[pos] = true;							35	expr[pos] = true;				
36	for(auto next : conn[pos])							36	for(auto next : conn[pos])				
37	dfs(next, result, expr);							37	dfs(next, result, expr);				
38	result.push_back(pos);							38	result.push_back(pos);				
39	}							39	}				
40	vector<vector<int> > scc(){ // tested on							40	vector<vector<int> > scc(){ // tested on				
41	~> https://www.hackerearth.com/practice/algorithms/graphs/strongly-connected-components/							41	~> https://www.hackerearth.com/practice/algorithms/graphs/strongly-connected-components/				
42	vector<int> order = topsort();							42	vector<int> order = topsort();				
43	reverse(order.begin(), order.end());							43	reverse(order.begin(), order.end());				
44	vector<bool> expr(n, false);							44	vector<bool> expr(n, false);				
45	vector<vector<int> > results;							45	vector<vector<int> > results;				
	for(auto it = order.rbegin(); it != order.rend(); ++it){								for(auto it = order.rbegin(); it != order.rend(); ++it){				

```

46     vector<int> component;
47     _topsort_dfs(*it, component, explr, conn);
48     sort(component.begin(), component.end());
49     results.push_back(component);
50   }
51   sort(results.begin(), results.end());
52   return results;
53 }
54 //Solution for:
55 → http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
56 int main() {
57     int n, m;
58     cin >> n >> m;
59     Graph g(2*m);
60     for(int i=0; i<n; i++) {
61         int a, sa, b, sb;
62         cin >> a >> sa >> b >> sb;
63         a--;
64         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     }
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); #388
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) { #759
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); #567
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);
60    }

```

```

59 }
60 void add(int l, int r, int to_add, int pos = 1, int lb = 0, int rb #168
61   ← = -1) {
62   propagate(pos);
63   if(rb == -1) rb = n;
64   if(l <= lb && rb <= r) {
65     nodes[pos].lazy_add += to_add;
66     return;
67   }
68   int mid = (lb + rb) / 2;
69   if(l < mid)
70     add(l, r, to_add, pos*2, lb, mid); #620
71   if(mid < r)
72     add(l, r, to_add, pos*2+1, mid, rb);
73   nodes[pos].value = get_value(pos*2) + get_value(pos*2+1);
74 }
75 long long get(int l, int r, int pos = 1, int lb = 0, int rb = -1) {
76   propagate(pos);
77   if(rb == -1) rb = n;
78   if(l <= lb && rb <= r) return get_value(pos);
79   int mid = (lb + rb) / 2;
80   long long result = 0; #133
81   if(l < mid)
82     result += get(l, r, pos*2, lb, mid);
83   if(mid < r)
84     result += get(l, r, pos*2+1, mid, rb);
85   return result;
86 }
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 #599
11   ← &rhs_seg){
12    sum_val = lhs_seg.sum_val + rhs_seg.sum_val;
13    min_val = min(lhs_seg.min_val, rhs_seg.min_val);
14  }
15  struct Lazy{ #237
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;
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
40      ← children
41      apply_to_lazy(seg_len, rhs_lazy);
42      init();
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 >=
92             1, len <= 1){
93             if(l_tmp & 1){
94                 upd.apply_to_lazy(len, lazys[l_tmp]);
95                 upd.apply_to_seg(len, segs[l_tmp]);
96                 is_lazy[l_tmp] = true;
97                 ++l_tmp;
98             }
99             if(r_tmp & 1){ #311
100                --r_tmp;
101                upd.apply_to_lazy(len, lazys[r_tmp]);
102                upd.apply_to_seg(len, segs[r_tmp]);
103                is_lazy[r_tmp] = true;
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 > #872
113        void query(int l, int r, void (Segment::*query_func)(int,
114             QueryArgs...), QueryArgs &&...query_args){ #475
115            l+=offset;
116            r+=offset;
117            push(l);
118            push(r-1);
119            int len = 1;
120            int r_orig = r;
121            for(; l< r; l>>=1, r>>=1, len <= 1){ //Segments applied in order
122                to quarry
123                if(l & 1){
124                    (segs[l++].*query_func)(len, query_args...);
125                }
126                for(; r < r_orig; ){
127                    r<<=1;
128                    len>>=1;
129                    if(r_orig & len){
130                        (segs[r++].*query_func)(len, query_args...);
131                    }
132                }
133            };
134            int main(){ #784
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){ #891
149                        Lazy tmp;

```

```

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 }

```

18 Tempted 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 qleft, int qright) {
22            qleft = max(qleft, lend);
23            qright = min(qright, rend);
24            if (qleft == lend && qright == rend) {
25                return value;
26            } else if (qleft > qright) {
27                return comp().identity();
28            } else {
29                return comp()(left->query(qleft, qright), right->query(qleft,
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 qleft, int qright, int revision) {
62        return roots[revision]->query(qleft, qright);
63    }
64    T query (int qleft, int qright) {
65        return roots[last_revision()]->query(qleft, qright);
66    }

```

#128 #890 %280

19 Tempted 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) { #593
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)) { #646
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) { #841
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);
76         structure = DS (vertexc + 5, initial);
77         aux = DS (50, initial);
78     }
79     void set (int vertex, int value) { #793
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]]; #219
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],
97             ord[v])));
98         return aux.query(1, cur_id); %515
99     }
100    void print () {
101        for (int i = 1; i <= vertexc; i++)
102            cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' <<
103            par[i] << endl;
104    };
105    int main () {
106        int vertexc;
107        cin >> vertexc;
108        HLD<int, dummy> hld (vertexc);
109        for (int i = 0; i < vertexc - 1; i++) {
110            int u, v;
111            cin >> u >> v;
112            hld.add_edge(u, v);
113        }
114        hld.build();
115        hld.print();
116        int queryc;
117        cin >> queryc;
118        for (int i = 0; i < queryc; i++) {
119            int u, v;
120            cin >> u >> v;
121            hld.query_path(u, v);
122            cout << endl;
123        }

```

20 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates,
2   elements, return_value.
3 // Includes coordinate compression.
4 template < typename elem_t, typename coord_t, coord_t n_inf, typename
5   ↪ ret_t >
6 class BIT {
7   vector< coord_t > positions;
8   vector< elem_t > elems;
9   bool initiated = false;
10 public:
11   BIT() {
12     positions.push_back(n_inf);                                #774
13   }
14   void initiate() {
15     if (initiated) {
16       for (elem_t &c_elem : elems)
17         c_elem.initiate();
18     } else {
19       initiated = true;
20       sort(positions.begin(), positions.end());
21       positions.resize(unique(positions.begin(), positions.end()) -
22         ↪ positions.begin());
23       elems.resize(positions.size());                            #919
24     }
25   }
26   template < typename... loc_form >
27   void update(coord_t cord, loc_form... args) {
28     if (initiated) {
29       int pos = lower_bound(positions.begin(), positions.end(), cord) -
30         ↪ positions.begin();
31       for (; pos < positions.size(); pos += pos & -pos)
32         elems[pos].update(args...);
33     } else {
34       positions.push_back(cord);                                #522
35     }
36   }
37   template < typename... loc_form >
38   ret_t query(coord_t cord, loc_form... args) { //sum in open interval
39     ↪ (-inf, cord)
40     ret_t res = 0;
41     int pos = (lower_bound(positions.begin(), positions.end(), cord) -
42       ↪ positions.begin())-1;
43     for (; pos > 0; pos -= pos & -pos)
44       res += elems[pos].query(args...);
45     return res;                                              #677
46   };
47   template < typename internal_type >
48   struct wrapped {
49     internal_type a = 0;
50     void update(internal_type b) {
51       a += b;
52     }
53   };
54 }
```

```

55   internal_type query() {
56     return a;
57   }
58 }
59 // Should never be called, needed for compilation
60 void initiate() {
61   cerr << 'i' << endl;
62 }
63 void update() {
64   cerr << 'u' << endl;
65 }
66 };
67 int main() {                                                 %330
68   // return type should be same as type inside wrapped
69   BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll >
70   ↪ fenwick;
71   int dim = 2;
72   vector< tuple< int, int, ll > > to_insert;
73   to_insert.emplace_back(1, 1, 1);
74   // set up all positions that are to be used for update
75   for (int i = 0; i < dim; ++i) {
76     for (auto &cur : to_insert)
77       fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include
78       ↪ value which won't be used
79     fenwick.initiate();
80   }
81   // actual use
82   for (auto &cur : to_insert)
83     fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
84   cout << fenwick.query(2, 2) << '\n';
85 }
```

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) {           #698
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    }
22  };
23 }
```

#391

%330

#295

```

21     if(rch) total += rch->total;
22 }
23 deque<Node> nodes;
24 Node* root = 0;
25 pair<Node*, Node*> split(int key, Node* cur) {
26     if(cur == 0) return {0, 0};
27     pair<Node*, Node*> result;
28     if(key <= cur->key) {
29         auto ret = split(key, cur->lch);
30         cur->lch = ret.second;
31         result = {ret.first, cur};
32     } else {
33         auto ret = split(key, cur->rch);
34         cur->rch = ret.first;
35         result = {cur, ret.second};
36     }
37     cur->update();
38     return result;
39 }
40 Node* merge(Node* left, Node* right) {
41     if(left == 0) return right;
42     if(right == 0) return left;
43     Node* top;
44     if(left->priority < right->priority) {
45         left->rch = merge(left->rch, right);
46         top = left;
47     } else {
48         right->lch = merge(left, right->lch);
49         top = right;
50     }
51     top->update();
52     return top;
53 }
54 void insert(int key, int value) {
55     nodes.push_back(Node(key, value));
56     Node* cur = &nodes.back();
57     pair<Node*, Node*> ret = split(key, root);
58     cur = merge(ret.first, cur);
59     cur = merge(cur, ret.second);
60     root = cur;
61 }
62 void erase(int key) {
63     Node *left, *mid, *right;
64     tie(left, mid) = split(key, root);
65     tie(mid, right) = split(key+1, mid);
66     root = merge(left, right);
67 }
68 long long sum_upto(int key, Node* cur) {
69     if(cur == 0) return 0;
70     if(key <= cur->key) {
71         return sum_upto(key, cur->lch);
72     } else {
73         long long result = cur->value + sum_upto(key, cur->rch);
74 }

```

#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); #509
81 }
82 };
83 //Solution for:
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;
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 }

```

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)); #086
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;
16         }
17     }
18 }

```

```

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 #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 };

```

#077 #228 %196

23 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;

```

#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)                                     #985
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;} %567
38 }

```

24 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      #406
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;                      #398
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; %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

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

$$[n+1]_m = n[n]_m + [n]_{m-1}$$

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

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

$$\{\cdot\}_m^{n+1} = k \{\cdot\}_k^n + \{\cdot\}_{k-1}^n$$

$$\sum_{k=0}^n \{\cdot\}_k(x)_k = x^n$$

$C_n = \frac{1}{n+1} \binom{2n}{n}$ — 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 $b_n = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} a_k$

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

- $a_i = i^k$, $b_i = \{\cdot\}_i^k i!$

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

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

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

$c_n = \sum_{k=0}^n \binom{n}{k} a_k b_{n-k}$ (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. Start with $B_0(x) = \frac{1}{a_0}$
2. 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
```