

## Contents

|    |   |    |
|----|---|----|
| 1  | Setup   | 1  |
| 2  | crc.sh  | 1  |
| 3  | gcc ordered set   | 2  |
| 4  | Triangle centers  | 2  |
| 5  | Convex polygon algorithms   | 3  |
| 6  | 2D line segment   | 6  |
| 7  | Dinic   | 9  |
| 8  | Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$ | 11 |
| 9  | Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$     | 12 |
| 10 | Aho Corasick $\mathcal{O}( \text{alpha}  \sum \text{len})$                      | 14 |
| 11 | Suffix automaton $\mathcal{O}((n + q) \log( \text{alpha} ))$                    | 15 |
| 12 | Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$             | 16 |
| 13 | Templated HLD $\mathcal{O}(M(n) \log n)$ per query                              | 17 |
| 14 | Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query               | 19 |
| 15 | FFT $\mathcal{O}(n \log(n))$  | 21 |
| 16 | MOD int, extended Euclidean   | 22 |
| 17 | Rabin Miller prime check  | 23 |
| 18 | Numerical integration with Simpson's rule                                       | 23 |
| 19 | Factsheet   | 24 |

## 1 Setup

```

1 set smartindent cindent
2 set ts=4 sw=4 expandtab
3 syntax enable
4 set clipboard=unnamedplus
5
6 "colorscheme elflord
7 "setxkbmap -option caps:escape

```

## 2 crc.sh

```

1#!/bin/env bash
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 -$j | tr -d '[:space:]' | cksum | cut -f1 -d ' ' | tail -c 4
8   done
9   echo
10 done //check last number in each block$ \hfill %298 $ 

```

### 3 gcc ordered set

---

```

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

#222  
%662

---

### 4 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 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();
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);

```

#623  
%446  
#385  
%742  
#193  
%031

---

```

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;                                     #157
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 }

```

%980

## 5 Convex polygon algorithms

```

11 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
51 cross(const pair< int, int > &v1, const pair< int, int > &v2) {
6   return (ll)v1.first * v2.second - (ll)v2.first * v1.second;
7 }
8
91 dist_sq(const pair< int, int > &p1, const pair< int, int > &p2) {
10  return (ll)(p2.first - p1.first) * (p2.first - p1.first) +
11    (ll)(p2.second - p1.second) * (p2.second - p1.second);
12 }
13
14 struct Hull {
15   vector< pair< pair< int, int >, pair< int, int > > > hull;
16   vector< pair< pair< int, int >, pair< int, int > > > ::iterator upper_begin;
17   template < typename Iterator >
18   void extend_hull(Iterator begin, Iterator end) { // O(n)          #025
19     vector< pair< int, int > > res;
20     for (auto it = begin; it != end; ++it) {
21       if (res.empty() || *it != res.back()) {
22         while (res.size() >= 2) {
23           auto v1 = make_pair(res[res.size() - 1].first - res[res.size() - 2].first,
24                                res[res.size() - 1].second - res[res.size() - 2].second); #048
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           }
30           res.pop_back();
31         }
32         res.push_back(*it);
33       }
34     }
35     for (int i = 0; i < res.size() - 1; ++i) {
36       hull.emplace_back(res[i], res[i + 1]);
37     }
38   }
39   Hull(vector< pair< int, int > > &vert) { // atleast 2 distinct points
40     sort(vert.begin(), vert.end());           // O(n log(n))
41     extend_hull(vert.begin(), vert.end());
42     int diff = hull.size();
43     extend_hull(vert.rbegin(), vert.rend()); #789
44     upper_begin = hull.begin() + diff;
45   }
46   bool contains(pair< int, int > p) { // O(log(n))
47     if (p < hull.front().first || p > upper_begin->first) return false;
48   {
49     auto it_low = lower_bound(hull.begin(), upper_begin,
50                               make_pair(make_pair(p.first, (int)-2e9), make_pair(0, 0)));
51     if (it_low != hull.begin()) {
52       --it_low;
53     }
54     auto v1 = make_pair(it_low->second.first - it_low->first.first,
55                         it_low->second.second - it_low->first.second); #867
56     auto v2 = make_pair(p.first - it_low->first.first, p.second - it_low->first.second);
57     if (cross(v1, v2) < 0) // < 0 is inclusive, <=0 is exclusive
58       return false;
59   }

```

%025

#674

#789

%432

#867

```

60
61     auto it_up = lower_bound(hull.rbegin(), hull.rbegin() + (hull.end() - upper_begin),
62                             make_pair(make_pair(p.first, (int)2e9), make_pair(0, 0)));
63     if (it_up - hull.rbegin() == hull.end() - upper_begin) {
64         --it_up;
65     }
66     auto v1 = make_pair(it_up->first.first - it_up->second.first,
67                          it_up->first.second - it_up->second.second);
68     auto v2 = make_pair(p.first - it_up->second.first, p.second - it_up->second.second);
69     if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
70         return false;
71     }
72     return true;
73 }
74 template < typename T > // The function can have only one local min and max and may be constant
75 // only at min and max.
76 vector< pair< pair< int, int >, pair< int, int > >::iterator max(
77     function< T(const pair< pair< int, int >, pair< int, int > > &) > f) { // O(log(n))
78     auto l = hull.begin();
79     auto r = hull.end();
80     vector< pair< pair< int, int >, pair< int, int > >::iterator best = hull.end();
81     T best_val;
82     while (r - l > 2) {
83         auto mid = l + (r - l) / 2;
84         T l_val = f(*l);
85         T l_nxt_val = f(*(l + 1));
86         T mid_val = f(*mid);
87         T mid_nxt_val = f(*(mid + 1));
88         if (best == hull.end() ||
89             l_val > best_val) { // If max is at l we may remove it from the range.
90             best = l;
91             best_val = l_val;
92         }
93         if (l_nxt_val > l_val) {
94             if (mid_val < l_val) {
95                 r = mid;
96             } else {
97                 if (mid_nxt_val > mid_val) {
98                     l = mid + 1;
99                 } else {
100                     r = mid + 1;
101                 }
102             }
103         } else {
104             if (mid_val < l_val) {
105                 l = mid + 1;
106             } else {
107                 if (mid_nxt_val > mid_val) {
108                     l = mid + 1;
109                 } else {
110                     r = mid + 1;
111                 }
112             }
113         }
114     }
115     T l_val = f(*l);
116     if (best == hull.end() || l_val > best_val) {
117         best = l;
118         best_val = l_val;
119     }
120     if (r - l > 1) {
121         T l_nxt_val = f(*(l + 1));
122         if (best == hull.end() || l_nxt_val > best_val) {
123             best = l + 1;
124             best_val = l_nxt_val;
125         }
126     }
127     return best;
128 }
129 vector< pair< pair< int, int >, pair< int, int > >::iterator closest(
130     pair< int, int >
131     p) { // p can't be internal(can be on border), hull must have atleast 3 points
132     const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))

```

#744  
%644

#242  
#012

#373  
#332

#930  
%331

```

133     return max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
134         [&p, &ref_p](const pair< pair< int, int >, pair< int, int > > &seg) { // accuracy of used type should be coord-2
135             if (p == seg.first) return 10 - M_PI;
136             auto v1 =
137                 make_pair(seg.second.first - seg.first.first, seg.second.second - seg.first.second); #685
138             auto v2 = make_pair(p.first - seg.first.first, p.second - seg.first.second);
139             ll cross_prod = cross(v1, v2);
140             if (cross_prod > 0) { // order the backside by angle
141                 auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
142                 auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
143                 ll dot_prod = dot(v1, v2);
144                 ll cross_prod = cross(v2, v1);
145                 return atan2(cross_prod, dot_prod) / 2;
146             }
147             ll dot_prod = dot(v1, v2);
148             double res = atan2(dot_prod, cross_prod);
149             if (dot_prod <= 0 && res > 0) res = -M_PI;
150             if (res > 0) {
151                 res += 20;
152             } else {
153                 res = 10 - res;
154             }
155             return res;
156         });
157     });
158 });
159 pair< int, int > forw_tan(pair< int, int > p) { // can't be internal or on border
160     const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))
161     auto best_seg = max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
162         [&p, &ref_p](const pair< pair< int, int >, pair< int, int > > &seg) { // accuracy of used type should be coord-2
163             auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
164             auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
165             ll dot_prod = dot(v1, v2);
166             ll cross_prod = cross(v2, v1); // cross(v1, v2) for backtan!!!
167             return atan2(cross_prod, dot_prod); // order by signed angle #291
168         });
169     );
170     return best_seg->first;
171 }
172 vector< pair< pair< int, int >, pair< int, int > >::iterator max_in_dir(
173     pair< int, int > v) { // first is the ans. O(log(n))
174     return max(function< ll(const pair< pair< int, int >, pair< int, int > > &) >(
175         [&v](const pair< pair< int, int >, pair< int, int > > &seg) { return dot(v, seg.first); }));
176 }
177 pair< vector< pair< int, int >, pair< int, int > >::iterator,
178     vector< pair< int, int >, pair< int, int > >::iterator > #013
179 intersections(pair< pair< int, int >, pair< int, int > > line) { // O(log(n))
180     int x = line.second.first - line.first.first;
181     int y = line.second.second - line.first.second;
182     auto dir = make_pair(-y, x);
183     auto it_max = max_in_dir(dir);
184     auto it_min = max_in_dir(make_pair(y, -x));
185     ll opt_val = dot(dir, line.first);
186     if (dot(dir, it_max->first) < opt_val || dot(dir, it_min->first) > opt_val) {
187         return make_pair(hull.end(), hull.end());
188     }
189     vector< pair< pair< int, int >, pair< int, int > >::iterator it_r1, it_r2; #913
190     function< bool(const pair< pair< int, int >, pair< int, int > > &,
191                     const pair< pair< int, int >, pair< int, int > > &) >
192         inc_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
193                         const pair< pair< int, int >, pair< int, int > > &rgt) {
194             return dot(dir, lft.first) < dot(dir, rgt.first);
195         });
196     function< bool(const pair< pair< int, int >, pair< int, int > > &,
197                     const pair< pair< int, int >, pair< int, int > > &) >
198         dec_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
199                         const pair< pair< int, int >, pair< int, int > > &rgt) {
200             return dot(dir, lft.first) > dot(dir, rgt.first);
201         });
202     if (it_min <= it_max) {
203         it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
204         if (dot(dir, hull.front().first) >= opt_val) {
205             it_r2 = upper_bound(hull.begin(), it_min + 1, line, dec_comp) - 1;

```

```

206     } else {
207         it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
208     }
209 } else {
210     it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
211     if (dot(dir, hull.front().first) <= opt_val) {
212         it_r2 = upper_bound(hull.begin(), it_max + 1, line, inc_comp) - 1;
213     } else {
214         it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
215     }
216 }
217 return make_pair(it_r1, it_r2);
218 }
219 pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
220     pair< pair< int, int >, pair< int, int > > res;
221     ll dia_sq = 0;
222     auto it1 = hull.begin();
223     auto it2 = upper_begin;
224     auto v1 = make_pair(hull.back().second.first - hull.back().first.first,
225                         hull.back().second.second - hull.back().first.second);
226     while (it2 != hull.begin()) {
227         auto v2 = make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
228                             (it2 - 1)->second.second - (it2 - 1)->first.second);
229         ll decider = cross(v1, v2);
230         if (decider > 0) break;
231         --it2;
232     }
233     while (it2 != hull.end()) { // check all antipodal pairs
234         if (dist_sq(it1->first, it2->first) > dia_sq) {
235             res = make_pair(it1->first, it2->first);
236             dia_sq = dist_sq(res.first, res.second);
237         }
238         auto v1 =
239             make_pair(it1->second.first - it1->first.first, it1->second.second - it1->first.second);
240         auto v2 =
241             make_pair(it2->second.first - it2->first.first, it2->second.second - it2->first.second);
242         ll decider = cross(v1, v2);
243         if (decider == 0) { // report cross pairs at parallel lines.
244             if (dist_sq(it1->second, it2->first) > dia_sq) {
245                 res = make_pair(it1->second, it2->first);
246                 dia_sq = dist_sq(res.first, res.second);
247             }
248             if (dist_sq(it1->first, it2->second) > dia_sq) {
249                 res = make_pair(it1->first, it2->second);
250                 dia_sq = dist_sq(res.first, res.second);
251             }
252             ++it1;
253             ++it2;
254         } else if (decider < 0) {
255             ++it1;
256         } else {
257             ++it2;
258         }
259     }
260     return res;
261 }
262 };

```

#454  
%442  
#671  
#674  
#466  
#502  
%215

## 6 2D line segment

```

1 const long double PI = acos(-1.0L);
2
3 struct Vec {
4     long double x, y;
5
6     Vec& operator+=(Vec r) {
7         x += r.x, y += r.y;
8         return *this;
9     }
10    Vec operator-(Vec r) {return Vec(*this) -= r;}
11
12    Vec& operator*=(long double s) {

```

```

13     x += r.x, y += r.y;
14     return *this;
15 }
16 Vec operator+(Vec r) {return Vec(*this) += r;}
17 Vec operator-() {return {-x, -y};}
18 Vec& operator*=(long double r) {
19     x *= r, y *= r;
20     return *this;
21 }
22 Vec operator*(long double r) {return Vec(*this) *= r;}
23 Vec& operator/=(long double r) {
24     x /= r, y /= r;
25     return *this;
26 }
27 Vec operator/(long double r) {return Vec(*this) /= r;}
28
29 long double operator*(Vec r) {
30     return x * r.x + y * r.y;
31 }
32 };
33 ostream& operator<<(ostream& l, Vec r) {
34     return l << '(' << r.x << ", " << r.y << ')';
35 }
36 long double len(Vec a) {
37     return hypot(a.x, a.y);
38 }
39 long double cross(Vec l, Vec r) {
40     return l.x * r.y - l.y * r.x;
41 }
42 long double angle(Vec a) {
43     return fmod(atan2(a.y, a.x)+2*PI, 2*PI);
44 }
45 Vec normal(Vec a) {
46     return Vec({-a.y, a.x}) / len(a);
47 } #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
11 Vec intersection(Segment l, Segment r) {
12     Vec dl = l.d(), dr = r.d();
13     if(cross(dl, dr) == 0)
14         return {nanl(""), nanl("")};
15
16     long double h = cross(dr, l.a-r.a) / len(dr);
17     long double dh = cross(dr, dl) / len(dr);
18
19     return l.a + dl * (h / -dh);
20 }
21
22 //Returns the area bounded by halfplanes
23 long double getArea(vector<Segment> lines) {
24     long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
25
26     vector<Segment> linesBySide[2];
27     for(auto line : lines) {
28         if(line.b.y == line.a.y) {
29             if(line.a.x < line.b.x)
30                 lowerbound = max(lowerbound, line.a.y);
31             else
32                 upperbound = min(upperbound, line.a.y);
33         }
34         else if(line.a.y < line.b.y)
35             linesBySide[1].push_back(line);
36     } #942

```



```
    i++;
111    else
112        j++;
113    }
114 }
115 return result;
116 }
```

%540

7 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;
40    vector<Edge> edges;
41    MaxFlow(int _source, int _sink):source(_source),sink(_sink){
42        assert(source != sink);
43    }
44    int add_edge(int u, int v, ll c, ll rc = 0){
45        edges.push_back(Edge(u,v,c,rc));
46        return edges.size()-1;
47    }
48    vector<int> now,lvl;
49    void prep(){
50        int max_id = max(source,sink);
51        for(auto edge : edges)
52            max_id = max(max_id,max(edge.u,edge.v));
53        adj.resize(max_id+1);
54        cap.resize(max_id+1);
55        now.resize(max_id+1);
56        lvl.resize(max_id+1);
57        for(auto &edge : edges){
58            auto flow = make_shared<ll>(0);
59            adj[edge.u].push_back(edge.v);
60            cap[edge.u].push_back(FlowTracker(edge.c,edge.rc,flow,0));
61            if(edge.u != edge.v){
62                adj[edge.v].push_back(edge.u);
63                cap[edge.v].push_back(FlowTracker(edge.c,edge.rc,flow,1));
64            }
65        }
66    }
67    void print(){
68        cout << "Adjacency List: " << endl;
69        for(int i = 0; i < adj.size(); i++)
70            cout << i << ": ";
71        cout << endl;
72        cout << "Capacity Matrix: " << endl;
73        for(int i = 0; i < cap.size(); i++)
74            for(int j = 0; j < cap[i].size(); j++)
75                cout << cap[i][j] << " ";
76        cout << endl;
77        cout << "Flow Matrix: " << endl;
78        for(int i = 0; i < flow.size(); i++)
79            for(int j = 0; j < flow[i].size(); j++)
80                cout << flow[i][j] << " ";
81        cout << endl;
82    }
83    void print(){
84        cout << "Adjacency List: " << endl;
85        for(int i = 0; i < adj.size(); i++)
86            cout << i << ": ";
87        cout << endl;
88        cout << "Capacity Matrix: " << endl;
89        for(int i = 0; i < cap.size(); i++)
90            for(int j = 0; j < cap[i].size(); j++)
91                cout << cap[i][j] << " ";
92        cout << endl;
93        cout << "Flow Matrix: " << endl;
94        for(int i = 0; i < flow.size(); i++)
95            for(int j = 0; j < flow[i].size(); j++)
96                cout << flow[i][j] << " ";
97        cout << endl;
98    }
99}

```

```

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(lvl.begin(),lvl.end(),0);
71     lvl[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 && lvl[v] == 0){
78                 lvl[v] = lvl[u]+1;
79                 bfs.push_back(v);
80             }
81         }
82     }
83     return lvl[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(lvl[v] == lvl[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;
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),
122     // sink doesn't need to be 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 about flow value
129     }
130     cout << mf.calc_max_flow() << '\n';
131     //cout << mf.flow_on_edge(edge_index) << endl; // return flow on this edge

```

## 8 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(){ #660
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){ //in case of no negative edges Bellman-Ford can be removed.
17         min_dist[i]=inf;
18     }
19     for(int i=0; i<v-1; ++i){ #004
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     }
28     for(int i=0; i<v; ++i){ //Apply potentials to edge costs. #513
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. #262
37     while(true){ #019
38         for(int i=0; i<=v; ++i){ //node after sink is used as start value for Dijkstra.
39             min_dist[i]=inf;
40             visited[i]=false;
41         }
42         min_dist[0]=0;
43         node_flow[0]=inf;
44         int min_node;
45         while(true){ //Use Dijkstra to calculate potentials #782
46             int min_node=v;
47             for(int i=0; i<v; ++i){
48                 if(!visited[i]) && min_dist[i]<min_dist[min_node]){
49                     min_node=i;
50                 }
51             }
52             if(min_node==v){
53                 break;
54             }
55             visited[min_node]=true;
56             for(int i=0; i<v; ++i){
57                 if(!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){ #192
58                     min_dist[i]=min_dist[min_node]+cost[min_node][i];
59                     prev_node[i]=min_node;
60                     node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
61                 }
62             }
63         }
64         if(min_dist[v-1]==inf){ #497
65             break;
66         }
67         for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
68             for(int j=0; j<v; ++j){ //Found path from source to sink becomes 0 cost.
69                 if(cost[i][j]!=inf){
70                     cost[i][j]+=min_dist[i];
71                     cost[i][j]-=min_dist[j];
72                 }
73             }
74         }
75     }
76 }
```

```

University of Tartu


---


72     }
73 }
74 sink_pot+=min_dist[v-1];
75 tot_flow+=node_flow[v-1];
76 tot_cost+=sink_pot*node_flow[v-1];
77 int cur=v-1;
78 while(cur!=0){ //Backtrack along found path that now has 0 cost.
79     rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
80     rem_flow[cur][prev_node[cur]]+=node_flow[v-1];
81     cost[cur][prev_node[cur]]=0;
82     if(rem_flow[prev_node[cur]][cur]==0){
83         cost[prev_node[cur]][cur]=inf;
84     }
85     cur=prev_node[cur];
86 }
87 }
88 }
89 }


---


#854
#907
%090

```

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

```

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


---


#042
#695
#304
#814

```

```

53     int n = nodes.size();
54     int result = 0;
55
56     struct State {
57         int p;
58         Edge* used;
59     };
60
61     while(1) {
62         vector<vector<State>> state(1, vector<State>(n, {0, 0}));
63
64         for(int lev = 0; lev < n; lev++) {
65             state.push_back(state[lev]);
66             for(int i=0;i<n;i++)
67                 if(lev == 0 || state[lev][i].p < state[lev-1][i].p) {
68
69                     for(Edge* edge : nodes[i].conn) if(edge->getCap(&nodes[i]) > 0) {
70                         int np = state[lev][i].p + (edge->u == &nodes[i] ? edge->cost : -edge->cost);
71                         int ni = edge->from(&nodes[i])->index;
72
73                         if(np < state[lev+1][ni].p) {
74                             state[lev+1][ni].p = np;
75                             state[lev+1][ni].used = edge;
76                         }
77                     }
78                 }
79             }
80
81             //Now look at the last level
82             bool valid = false;
83
84             for(int i=0;i<n;i++)
85                 if(state[n-1][i].p > state[n][i].p) {
86                     valid = true;
87
88                     vector<Edge*> path;
89
90                     int cap = 1000000000;
91                     Node* cur = &nodes[i];
92                     int clev = n;
93
94                     vector<bool> exprl(n, false);
95
96                     while(!exprl[cur->index]) {
97                         exprl[cur->index] = true;
98
99                         State cstate = state[clev][cur->index];
100                        cur = cstate.used->from(cur);
101
102                        path.push_back(cstate.used);
103                     }
104
105                     reverse(path.begin(), path.end());
106
107                     {
108                         int i=0;
109                         Node* cur2 = cur;
110
111                         do {
112                             cur2 = path[i]->from(cur2);
113                             i++;
114                         }while(cur2 != cur);
115
116                         path.resize(i);
117                     }
118
119                     for(auto edge : path) {
120                         cap = min(cap, edge->getCap(cur));
121                         cur = edge->from(cur);
122                     }
123
124                     for(auto edge : path) {
125                         result += edge->addFlow(cur, cap);
#025
#963
#478
#873
#528

```

```

University of Tartu
126         cur = edge->from(cur);
127     }
128 }
129     if(!valid) break;
130 }
131 }
132     return result;
133 }
134 }
135 }
136 };

```

#131

%254

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

```

1 const int alpha_size=26;
2 struct node{
3     node *nxt[alpha_size]; //May use other structures to move in trie
4     node *suffix;
5     node(){
6         memset(nxt, 0, alpha_size*sizeof(node *));
7     }
8     int cnt=0;
9 };
10 node *aho_corasick(vector<vector<char> > &dict){ #480
11     node *root= new node;
12     root->suffix = 0;
13     vector<pair<vector<char> *, node *> > cur_state;
14     for(vector<char> &s : dict)
15         cur_state.emplace_back(&s, root);
16     for(int i=0; !cur_state.empty(); ++i){
17         vector<pair<vector<char> *, node *> > nxt_state;
18         for(auto &cur : cur_state){
19             node *nxt=cur.second->nxt[(*cur.first)[i]];
20             if(nxt){
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             }
36             if(cur.first->size() > i+1)
37                 nxt_state.push_back(cur);
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         }
51         cur = cur->suffix;
52     }
53 }
54 void cnt_matches(node *root, vector<char> &match_in){ #074
55     node *cur = root;
56     for(char c : match_in){
57         cur = walk(cur, c);
58         ++cur->cnt;

```

#888

#786

#940

%064

%074

```

59 }
60 }
61 void add_cnt(node *root){ //After counting matches propagete ONCE to suffixes for final counts
62     vector<node *> to_visit = {root};
63     for(int i=0; i<to_visit.size(); ++i){
64         node *cur = to_visit[i];
65         for(int j=0; j<alpha_size; ++j){
66             if(cur->nxt[j]){
67                 to_visit.push_back(cur->nxt[j]);
68             }
69         }
70     }
71     for(int i=to_visit.size()-1; i>0; --i){
72         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
73     }
74 }

```

%286

#605

%459

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

```

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
36            ← tracked externally
37            ++match_len;
38            return nxt(c)->lower_depth(depth);
39        }
40        if (suf)
41            return suf->walk(c, depth, match_len);
42        return this;
43    }
44    int paths_to_end = 0;
45    void set_as_end() { //All suffixes of current node are marked as ending nodes.
46        paths_to_end = 1;
47        if (suf) suf->set_as_end();
48    }
49    bool vis = false;
50    void calc_paths_to_end() { //Call ONCE from ROOT. For each node calculates number of ways to reach an
51        ← end node.
52        if (!vis) { //paths_to_end is occurrence count for any strings in current suffix
53            ← equivalence class.
54        }
55    }

```

#486

#952

#795

#152

```

50     vis = true;
51     for (auto cur : nxt_char) {
52         cur.second->calc_paths_to_end();
53         paths_to_end += cur.second->paths_to_end;
54     }
55 }
56 };
57 };
58 struct SufAutomaton {
59     AutoNode *last;
60     AutoNode *root;
61     void extend(char new_c) {
62         AutoNode *new_end = new AutoNode; // The equivalence class containing the whole new string
63         new_end->len = last->len + 1;
64         AutoNode *suf_w_nxt = last; // The whole old string class
65         while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) { // is turned into the longest suffix which
66             // can be turned into a substring of old state
67             // by appending new_c
68             suf_w_nxt->set_nxt(new_c, new_end);
69             suf_w_nxt = suf_w_nxt->suf;
70         }
71         if (!suf_w_nxt) { // The new character isn't part of the old string
72             new_end->suf = root;
73         } else {
74             AutoNode *max_sbstr = suf_w_nxt->nxt(new_c); // Equivalence class containing longest
75             // substring which is a suffix of the new state.
76             if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
77                 new_end->suf = max_sbstr;
78             } else {
79                 AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
80                 new_end->suf = eq_sbstr;
81                 // Make suffixes of suf_w_nxt point to eq_sbstr instead of max_sbstr
82                 AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
83                 while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
84                     w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
85                     w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
86                 }
87             }
88         }
89         last = new_end;
90     }
91     SufAutomaton(string to_suffix) {
92         root = new AutoNode;
93         root->len = 0;
94         root->suf = NULL;
95         last = root;
96         for (char c : to_suffix) extend(c);
97     }
98 };

```

#738 #885 #873 #256 #409 %070

## 12 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$

```

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
9 public:
10    BIT() {
11        positions.push_back(n_inf);
12    }
13    void initiate() {
14        if (initiated) {
15            for (elem_t &c_elem : elems)
16                c_elem.initiate();
17        } else {
18            initiated = true;
19            sort(positions.begin(), positions.end());
20            positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
21            elems.resize(positions.size());

```

#448

```

22     }
23 }
24 template < typename... loc_form >
25 void update(coord_t cord, loc_form... args) {
26     if (initiated) {
27         int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
28         for (; pos < positions.size(); pos += pos & -pos)
29             elems[pos].update(args...);
30     } else {
31         positions.push_back(cord);
32     }
33 }
34 template < typename... loc_form >
35 ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
36     ret_t res = 0;
37     int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
38     for (; pos > 0; pos -= pos & -pos)
39         res += elems[pos].query(args...);
40     return res;
41 }
42 };
43 template < typename internal_type >
44 struct wrapped {
45     internal_type a = 0;
46     void update(internal_type b) {
47         a += b;
48     }
49     internal_type query() {
50         return a;
51     }
52     // Should never be called, needed for compilation
53     void initiate() {
54         cerr << 'i' << endl;
55     }
56     void update() {
57         cerr << 'u' << endl;
58     }
59 };
60 int main() {
61     // return type should be same as type inside wrapped
62     BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
63     int dim = 2;
64     vector< tuple< int, int, ll > > to_insert;
65     to_insert.emplace_back(1, 1, 1);
66     // set up all positions that are to be used for update
67     for (int i = 0; i < dim; ++i) {
68         for (auto &cur : to_insert)
69             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include value which won't be used
70         fenwick.initiate();
71     }
72     // actual use
73     for (auto &cur : to_insert)
74         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
75     cout << fenwick.query(2, 2) << '\n';
76 }

```

## 13 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query

```

1 class dummy {
2     public:
3     dummy () {
4     }
5
6     dummy (int, int) {
7     }
8
9     void set (int, int) {
10    }
11
12    int query (int left, int right) {
13        cout << this << ' ' << left << ' ' << right << endl;
14    }

```

```

15 };
```

---

```

16 /* T should be the type of the data stored in each vertex;
17 * DS should be the underlying data structure that is used to perform the
18 * group operation. It should have the following methods:
19 * * DS () - empty constructor
20 * * DS (int size, T initial) - constructs the structure with the given size,
21 * initially filled with initial.
22 * * void set (int index, T value) - set the value at index `index` to `value`.
23 * * T query (int left, int right) - return the "sum" of elements between left and right, inclusive.
24 */
25
26 template<typename T, class DS>
27 class HLD {
28     int vertexc;
29     vector<int> *adj;
30     vector<int> subtree_size;
31     DS structure;
32     DS aux;
33
34     void build_sizes (int vertex, int parent) { #037
35         subtree_size[vertex] = 1;
36         for (int child : adj[vertex]) {
37             if (child != parent) {
38                 build_sizes(child, vertex);
39                 subtree_size[vertex] += subtree_size[child];
40             }
41         }
42     }
43
44     int cur;
45     vector<int> ord;
46     vector<int> chain_root;
47     vector<int> par;
48     void build_hld (int vertex, int parent, int chain_source) { #593
49         cur++;
50         ord[vertex] = cur;
51         chain_root[vertex] = chain_source;
52         par[vertex] = parent;
53
54         if (adj[vertex].size() > 1) { #467
55             int big_child, big_size = -1;
56             for (int child : adj[vertex]) {
57                 if ((child != parent) &&
58                     (subtree_size[child] > big_size)) {
59                     big_child = child;
60                     big_size = subtree_size[child];
61                 }
62             }
63
64             build_hld(big_child, vertex, chain_source);
65             for (int child : adj[vertex]) {
66                 if ((child != parent) && (child != big_child)) {
67                     build_hld(child, vertex, child);
68                 }
69             }
70         }
71     }
72
73     public:
74     HLD (int _vertexc) {
75         vertexc = _vertexc;
76         adj = new vector<int> [vertexc + 5];
77     }
78
79     void add_edge (int u, int v) { #458
80         adj[u].push_back(v);
81         adj[v].push_back(u);
82     }
83
84     void build (T initial) {
85         subtree_size = vector<int> (vertexc + 5);
86         ord = vector<int> (vertexc + 5);
87         chain_root = vector<int> (vertexc + 5);

```

```

88     par = vector<int> (vertexc + 5);
89     cur = 0;
90     build_sizes(1, -1);
91     build_hld(1, -1, 1);
92     structure = DS (vertexc + 5, initial);
93     aux = DS (50, initial);
94 }
95
96 void set (int vertex, int value) {
97     structure.set(ord[vertex], value);
98 }
99
100 T query_path (int u, int v) { /* returns the "sum" of the path u->v */
101     int cur_id = 0;
102     while (chain_root[u] != chain_root[v]) {
103         if (ord[u] > ord[v]) {
104             cur_id++;
105             aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
106             u = par[chain_root[u]];
107         } else {
108             cur_id++;
109             aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
110             v = par[chain_root[v]];
111         }
112     }
113
114     cur_id++;
115     aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
116
117     return aux.query(1, cur_id);
118 }
119
120 void print () {
121     for (int i = 1; i <= vertexc; i++) {
122         cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' << par[i] << endl;
123     }
124 }
125 };
126
127 int main () {
128     int vertexc;
129     cin >> vertexc;
130
131     HLD<int, dummy> hld (vertexc);
132     for (int i = 0; i < vertexc - 1; i++) {
133         int u, v;
134         cin >> u >> v;
135
136         hld.add_edge(u, v);
137     }
138     hld.build();
139     hld.print();
140
141     int queryc;
142     cin >> queryc;
143     for (int i = 0; i < queryc; i++) {
144         int u, v;
145         cin >> u >> v;
146
147         hld.query_path(u, v);
148         cout << endl;
149     }
150 }

```

## 14 Templatized 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

```

```

8   Node (int position, T _value) {
9     left = NULL;
10    right = NULL;
11    lend = position;
12    rend = position;
13    value = _value;
14  }
15
16  Node (Node *_left, Node *_right) {
17    left = _left;
18    right = _right;
19    lend = left->lend;
20    rend = right->rend;
21    value = comp()(left->value, right->value);
22  }
23
24  T query (int qleft, int qright) {
25    qleft = max(qleft, lend);
26    qright = min(qright, rend);
27
28    if (qleft == lend && qright == rend) {
29      return value;
30    } else if (qleft > qright) {
31      return comp().identity();
32    } else {
33      return comp()(left->query(qleft, qright),
34                    right->query(qleft, qright));
35    }
36  }
37 };
38
39  int size;
40  Node **tree;
41  vector<Node*> roots;
42 public:
43 PersistentST () {}
44
45 PersistentST (int _size, T initial) {
46   for (int i = 0; i < 32; i++) {
47     if ((1 << i) > _size) {
48       size = 1 << i;
49       break;
50     }
51   }
52
53   tree = new Node* [2 * size + 5];
54
55   for (int i = size; i < 2 * size; i++) {
56     tree[i] = new Node (i - size, initial);
57   }
58
59   for (int i = size - 1; i > 0; i--) {
60     tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
61   }
62
63   roots = vector<Node*> (1, tree[1]);
64 }
65
66 void set (int position, T _value) {
67   tree[size + position] = new Node (position, _value);
68   for (int i = (size + position) / 2; i >= 1; i /= 2) {
69     tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
70   }
71   roots.push_back(tree[1]);
72 }
73
74 int last_revision () {
75   return (int) roots.size() - 1;
76 }
77
78 T query (int qleft, int qright, int revision) {
79   return roots[revision]->query(qleft, qright);
80 }

```

#479

#373

#815

#236

#942

#685

```

81     T query (int qleft, int qright) {
82         return roots[last_revision()]->query(qleft, qright);
83     }
84 }
85 
```

#495  
%026

## 15 FFT $\mathcal{O}(n \log(n))$

```

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

16 //Rearrange a
17 for(int block = n; block > 1; block /= 2) {
18     int half = block/2;
19     vector<complex<long double>> na(n);
20     for(int s=0; s < n; s += block)
21         for(int i=0;i<block;i++)
22             na[s + half*(i%2) + i/2] = a[s+i];
23     a = na;
24 }
25 
```

#659  
#843

```

26 //Now do the calculation
27 for(int block = 2; block <= n; block *= 2) {
28     vector<complex<long double>> na(n);
29     int wb = n/block, half = block/2;
30
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     a = na;
37 }
38 
```

#741

```

39
40     return a;
41 }
42
43
44 struct Polynomial {
45     vector<long double> a;
46
47     long double& operator[](int ind) {
48         return a[ind];
49     }
50
51     Polynomial& operator*=(long double r) {
52         for(auto &c : a)
53             c *= r;
54         return *this;
55     }
56     Polynomial operator*(long double r) {return Polynomial(*this) *= r;}
57
58     Polynomial& operator/=(long double r) {
59         for(auto &c : a)
60             c /= r;
61         return *this;
62     }
63     Polynomial operator/(long double r) {return Polynomial(*this) /= r;}
64
65     Polynomial& operator+=(Polynomial r) {

```

#694  
#211

```

66     if(a.size() < r.a.size())
67         a.resize(r.a.size(), 0.0L);
68     for(int i=0;i<(int)r.a.size();i++)
69         a[i] += r[i];
70     return *this;
71 }
72 Polynomial operator+(Polynomial r) {return Polynomial(*this) += r;}
73
74 Polynomial& operator-=(Polynomial r) {
75     if(a.size() < r.a.size())
76         a.resize(r.a.size(), 0.0L);
77     for(int i=0;i<(int)r.a.size();i++)
78         a[i] -= r[i];
79     return *this;
80 }
81 Polynomial operator-(Polynomial r) {return Polynomial(*this) -= r;}
82
83 Polynomial operator*(Polynomial r) {
84     int n = 1;
85     while(n < (int)(a.size() + r.a.size() - 1))
86         n *= 2;                                         #955
87
88     vector<complex<long double>> fl(n, 0.0L), fr(n, 0.0L);
89     for(int i=0;i<(int)a.size();i++)
90         fl[i] = a[i];
91     for(int i=0;i<(int)r.a.size();i++)
92         fr[i] = r[i];
93
94     fl = fastFourierTransform(fl, false);
95     fr = fastFourierTransform(fr, false);
96
97     vector<complex<long double>> ret(n);
98     for(int i=0;i<n;i++)
99         ret[i] = fl[i] * fr[i];                         #007
100    ret = fastFourierTransform(ret, true);
101
102    Polynomial result;
103    result.a.resize(a.size() + r.a.size() - 1);
104    for(int i=0;i<(int)result.a.size();i++)
105        result[i] = ret[i].real() / n;
106    return result;
107 }                                                 %194
108 };

```

## 16 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
8
9 struct Modint {
10     static const int MOD = 1000000007;
11     int val;
12
13     Modint(int nval = 0) {
14         val = nval;                                #412
15     }
16
17     Modint& operator+=(Modint r) {
18         val = (val + r.val) % MOD;
19         return *this;
20     }
21     Modint operator+(Modint r) {return Modint(*this) += r;}
22
23     Modint& operator-=(Modint r) {
24         val = (val + MOD - r.val) % MOD;
25         return *this;                            #052
26     }
27     Modint operator-(Modint r) {return Modint(*this) -= r;}

```

```

28     Modint& operator*=(Modint r) {
29         val = 1LL * val * r.val % MOD;
30         return *this;
31     }
32     Modint operator*(Modint r) {return Modint(*this) *= r;}
33
34     Modint inverse() {
35         int ret = extendedEuclideanAlgorithm(val, MOD).first;
36         if(ret < 0)
37             ret += MOD;
38         return ret;
39     }
40 }
41
42     Modint& operator/=(Modint r) {
43         return operator*=(r.inverse());
44     }
45     Modint operator/(Modint r) {return Modint(*this) /= r;}
46 };

```

#985  
%567

## 17 Rabin Miller prime check

```

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

```

#804  
#908  
%812

## 18 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 }

```

#430  
%360

# 19 Factsheet

## 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

$\left[ \begin{smallmatrix} n \\ m \end{smallmatrix} \right]$  — Stirling numbers of the first kind; number of permutations of  $n$  elements with  $k$  cycles

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

$$(x)_n = x(x-1)\dots x - n + 1 = \sum_{k=0}^n (-1)^{n-k} \left[ \begin{smallmatrix} n \\ k \end{smallmatrix} \right] x^k$$

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

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

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

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

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

### Binomial transform

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

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

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

### Burnside's lemma

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

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

The number of equivalence classes then can be calculated as follows:  $C = \frac{1}{|G|} \sum_{f \in G} |X^f|$ , where  $X^f$

is the set of *fixed points* of  $f$ :  $X^f = \{x | f(x) = x\}$

### Generating functions

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

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

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

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

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

### General linear recurrences

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

### Inverse polynomial modulo $x^l$

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

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

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

### Fast subset convolution

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

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

### Hadamard transform

Treat array  $a$  of size  $2^k$  as  $k$ -dimentional 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
```

- **Fermat's little theorem.** Let  $p$  be prime. Then, for each integer  $a$ :

$$a^{p-1} \equiv 1 \pmod{p}.$$

Thus:

$$a^k \equiv a^k \pmod{(p-1)} \pmod{p}.$$

Also:

$$a^{p-2} \equiv a^{-1} \pmod{p}.$$

- **Iterating over subsets.** Let `mask` be the binary representation of a set. Then `for (int i = mask; i != 0; i = (i - 1) & mask)` will iterate over all the nonempty subsets of `mask`.

- **Chinese remainder theorem.** We know that:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \end{aligned}$$

where  $n_1$  and  $n_2$  are (co)prime. We want to find  $a_{1,2}$  so that:

$$x \equiv a_{1,2} \pmod{n_1 \cdot n_2}.$$

A solution is given by:

$$a_{1,2} = a_1 m_2 n_2 + a_2 m_1 n_1,$$

where  $m_1$  and  $m_2$  are integers so that  $m_1 n_1 + m_2 n_2 = 1$ . Those values can be found using the Extended Euclidean algorithm.

- **Sum of harmonic series.**

$$\frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n} \in \mathcal{O}(\log n)$$

- **Number of primes below...**

|        |        |
|--------|--------|
| $10^2$ | 25     |
| $10^3$ | 168    |
| $10^4$ | 1229   |
| $10^5$ | 9592   |
| $10^6$ | 78498  |
| $10^7$ | 664579 |