

## Содержание

1	Strategy.txt	1
2	flows/dinic.cpp	2
3	flows/globalcut.cpp	2
4	flows/hungary.cpp	3
5	flows/mincost.cpp	3
6	geometry/halfplanes.cpp	4
7	geometry/primitives.cpp	5
8	geometry/svg.cpp	6
9	graphs/2sat.cpp	7
10	graphs/directed_mst.cpp	7
11	math/fft_recursive.cpp	8
12	math/golden_search.cpp	9
13	math/numbers.txt	9
14	strings/automaton.cpp	10
15	strings/eertree.cpp	10
16	strings/suffix_array.cpp	11
17	strings/ukkonen.cpp	11
18	structures/convex_hull_trick.cpp	12
19	structures/heavy_light.cpp	13
20	structures/ordered_set.cpp	14
21	structures/splay.cpp	14
22	structures/treap.cpp	15

## 1 Strategy.txt

- 1 - Проверить руками сэмплы
- Подумать как дебагать после написания
- 2 - Выписать сложные формулы и все +-1
- 2 - Проверить имена файлов
- Прогнать сэмплы
- 3 - Переполнения int, переполнения long long
- Выход за границу массива: \_GLIBCXX\_DEBUG
- Переполнения по модулю: в
  - ↪ псевдо-онлайн-генераторе, в функциях-обертках
- Проверить мультитест на разных тестах
- 5 - Прогнать минимальный по каждому параметру тест
- Прогнать псевдо-максимальный тест(немного
  - ↪ чисел, но очень большие или очень маленькие)
- Представить что не зайдет и заранее написать
  - ↪ assert'ы, прогнать слегка модифицированные
  - ↪ тесты
- 7 - cout.precision: в том числе в интерактивных
  - ↪ задачах
- Удалить debug-output, отсечения для тестов,
  - ↪ вернуть оригинальный main, удалить
  - ↪ \_GLIBCXX\_DEBUG
- 9 - Вердикт может врать
- Если много тестов(>3), дописать в конец каждого
  - ↪ теста ответ, чтобы не забыть
- (WA) Потестить не только ответ, но и содержимое
  - ↪ значимых массивов, переменных
- (WA) Изменить тест так, чтобы ответ не менялся:
  - ↪ поменять координаты местами, сжать/растянуть
  - ↪ координаты, поменять ROOT дерева
- (WA) Подвигать размер блока в корневой или
  - ↪ битсете
- (WA) Поставить assert'ы, возможно написать
  - ↪ чекер с assert'ом
- (WA) Проверить, что программа не печатает
  - ↪ что-либо неожиданное, что должно попадать под
  - ↪ PE: inf - 2, не лекс. мин. решение, одинаковые
  - ↪ числа вместо разных, неправильное количество
  - ↪ чисел, пустой ответ, перечитать output format
- (TL) cin -> scanf -> getchar
- (TL) Упихать в кэш большие массивы, поменять
  - ↪ местами for'ы или измерения массива
- (RE) Проверить формулы на деление на 0, выход
  - ↪ за область определения(sqrt(-eps), acos(1 +
  - ↪ eps))

## 2 flows/dinic.cpp

```
1 namespace Dinic {
2  const int maxn = 10010;
3
4  struct Edge {
5      int to, c, f;
6  } es[maxn*2];
7  int ne = 0;
8
9  int n;
10 vector<int> e[maxn];
11 int q[maxn], d[maxn], pos[maxn];
12 int S, T;
13
14 void addEdge(int u, int v, int c) {
15     assert(c <= 1000000000);
16     es[ne] = {v, c, 0};
17     e[u].push_back(ne++);
18     es[ne] = {u, 0, 0};
19     e[v].push_back(ne++);
20 }
21
22 bool bfs() {
23     for(i, n) d[i] = maxn;
24     d[S] = 0, q[0] = S;
25     int lq = 0, rq = 1;
26     while (lq != rq) {
27         int v = q[lq++];
28         for (int id: e[v]) if (es[id].f < es[id].c) {
29             int to = es[id].to;
30             if (d[to] == maxn)
31                 d[to] = d[v] + 1, q[rq++] = to;
32         }
33     }
34     return d[T] != maxn;
35 }
36
37 int dfs(int v, int curf) {
38     if (v == T || curf == 0) return curf;
39     for (int &i = pos[v]; i < (int)e[v].size(); ++i) {
40         int id = e[v][i];
41         int to = es[id].to;
42         if (es[id].f < es[id].c && d[v] + 1 == d[to]) {
43             if (int ret = dfs(to, min(curf, es[id].c - es[id].f))) {
44                 es[id].f += ret;
45                 es[id^1].f -= ret;
46                 return ret;
47             }
48         }
49     }
50     return 0;
51 }
52
53 i64 dinic(int S, int T) {
54     Dinic::S = S, Dinic::T = T;
55     i64 res = 0;
56     while (bfs()) {
57         for(i, n) pos[i] = 0;
58         while (int f = dfs(S, 1e9)) {
59             assert(f <= 1000000000);
60             res += f;
61         }
62     }
63     return res;
64 }
65
66 } // namespace Dinic
67
68 void test() {
69     Dinic::n = 4;
70     Dinic::addEdge(0, 1, 1);
71     Dinic::addEdge(0, 2, 2);
72     Dinic::addEdge(2, 1, 1);
73     Dinic::addEdge(1, 3, 2);
74     Dinic::addEdge(2, 3, 1);
75     cout << Dinic::dinic(0, 3) << endl; // 3
76
77 }
```

## 3 flows/globalcut.cpp

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define forn(i,n) for (int i = 0; i < int(n); ++i)
4 const int inf = 1e9 + 1e5;
5
6 const int maxn = 505;
7 namespace StoerWagner {
8     int g[maxn][maxn];
9     int dist[maxn];
10    bool used[maxn];
11    int n;
12
13    void addEdge(int u, int v, int c) {
14        g[u][v] += c;
15        g[v][u] += c;
16    }
17
18    int run() {
19        vector<int> vertices;
20        forn (i, n)
21            vertices.push_back(i);
22        int mincut = inf;
23        while (vertices.size() > 1) {
24            int u = vertices[0];
25            for (auto v: vertices) {
26                used[v] = false;
27                dist[v] = g[u][v];
28            }
29            used[u] = true;
30            forn (ii, vertices.size() - 2) {
31                for (auto v: vertices)
32                    if (!used[v])
33                        if (used[u] || dist[v] > dist[u])
34                            u = v;
35                used[u] = true;
36                for (auto v: vertices)
37                    if (!used[v])
38                        dist[v] += g[u][v];
39            }
40            int t = -1;
41            for (auto v: vertices)
42                if (!used[v])
43                    t = v;
44            assert(t != -1);
45            mincut = min(mincut, dist[t]);
46            vertices.erase(find(vertices.begin(), vertices.end(), t));
47            for (auto v: vertices)
48                addEdge(u, v, g[v][t]);
49        }
50        return mincut;
51    }
52};
53
54 int main() {
55     StoerWagner::n = 4;
56     StoerWagner::addEdge(0, 1, 5);
57     StoerWagner::addEdge(2, 3, 5);
58     StoerWagner::addEdge(1, 2, 4);
59     cerr << StoerWagner::run() << '\n';
60 }
```

## 4 flows/hungary.cpp

```

1 // left half is the smaller one
2 namespace Hungary {
3     const int maxn = 505;
4     int a[maxn][maxn];
5     int p[2][maxn];
6     int match[maxn];
7     bool used[maxn];
8     int from[maxn];
9     int mind[maxn];
10    int n, m;
11
12    int hungary(int v) {
13        used[v] = true;
14        int u = match[v];
15        int best = -1;
16        for (i, m + 1) {
17            if (used[i])
18                continue;
19            int nw = a[u][i] - p[0][u] - p[1][i];
20            if (nw <= mind[i]) {
21                mind[i] = nw;
22                from[i] = v;
23            }
24            if (best == -1 || mind[best] > mind[i])
25                best = i;
26        }
27        v = best;
28        int delta = mind[best];
29        for (i, m + 1) {
30            if (used[i]) {
31                p[1][i] -= delta;
32                p[0][match[i]] += delta;
33            } else
34                mind[i] -= delta;
35        }
36        if (match[v] == -1)
37            return v;
38        return hungary(v);
39    }
40
41    void check() {
42        int edges = 0, res = 0;
43        for (i, m)
44            if (match[i] != -1) {
45                ++edges;
46                assert(p[0][match[i]] + p[1][i] == a[match[i]][i]);
47                res += a[match[i]][i];
48            } else
49                assert(p[1][i] == 0);
50        assert(res == -p[1][m]);
51        for (i, n) for (j, m)
52            assert(p[0][i] + p[1][j] <= a[i][j]);
53    }
54
55    int run() {
56        for (i, n)
57            p[0][i] = 0;
58        for (i, m + 1) {
59            p[1][i] = 0;
60            match[i] = -1;
61        }
62        for (i, n) {
63            match[m] = i;
64            fill(used, used + m + 1, false);
65            fill(mind, mind + m + 1, inf);
66            fill(from, from + m + 1, -1);
67            int v = hungary(m);
68            while (v != m) {
69                int w = from[v];
70                match[v] = match[w];
71                v = w;
72            }
73        }
74        check();
75        return -p[1][m];
76    }
77};

```

## 5 flows/mincost.cpp

```

1 namespace MinCost {
2     const ll infc = 1e12;
3
4     struct Edge {
5         int to;
6         ll c, f, cost;
7
8         Edge(int to, ll c, ll cost): to(to), c(c), f(0), cost(cost) {}
9     };
10
11    int N, S, T;
12    int totalFlow;
13    ll totalCost;
14    const int maxn = 505;
15    vector<Edge> edge;
16    vector<int> g[maxn];
17
18    void addEdge(int u, int v, ll c, ll cost) {
19        g[u].push_back(edge.size());
20        edge.emplace_back(v, c, cost);
21        g[v].push_back(edge.size());
22        edge.emplace_back(u, 0, -cost);
23    }
24
25    ll dist[maxn];
26    int fromEdge[maxn];
27
28    bool inQueue[maxn];
29    bool fordBellman() {
30        for (i, N)
31            dist[i] = infc;
32        dist[S] = 0;
33        inQueue[S] = true;
34        vector<int> q;
35        q.push_back(S);
36        for (int ii = 0; ii < int(q.size()); ++ii) {
37            int u = q[ii];
38            inQueue[u] = false;
39            for (int e: g[u]) {
40                if (edge[e].f == edge[e].c)
41                    continue;
42                int v = edge[e].to;
43                ll nw = edge[e].cost + dist[u];
44                if (nw >= dist[v])
45                    continue;
46                dist[v] = nw;
47                fromEdge[v] = e;
48                if (!inQueue[v]) {
49                    inQueue[v] = true;
50                    q.push_back(v);
51                }
52            }
53        }
54        return dist[T] != infc;
55    }
56
57    ll pot[maxn];
58    bool dijkstra() {
59        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
60            ⇨ greater<pair<ll, int>>> q;
61        for (i, N)
62            dist[i] = infc;
63        dist[S] = 0;
64        q.emplace(dist[S], S);
65        while (!q.empty()) {
66            int u = q.top().second;
67            ll cdist = q.top().first;
68            q.pop();
69            if (cdist != dist[u])
70                continue;
71            for (int e: g[u]) {
72                int v = edge[e].to;
73                if (edge[e].c == edge[e].f)
74                    continue;
75                ll w = edge[e].cost + pot[u] - pot[v];
76                assert(w >= 0);
77                ll ndist = w + dist[u];
78                if (ndist >= dist[v])
79                    continue;
80                dist[v] = ndist;
81                fromEdge[v] = e;
82                q.emplace(dist[v], v);
83            }
84        }
85        if (dist[T] == infc)
86            return false;
87        for (i, N) {
88            if (dist[i] == infc)
89                continue;
90            pot[i] += dist[i];
91        }
92        return true;
93    }
94
95    int run() {
96        for (i, N)
97            dist[i] = infc;
98        dist[S] = 0;
99        inQueue[S] = true;
100        vector<int> q;
101        q.push_back(S);
102        for (int ii = 0; ii < int(q.size()); ++ii) {
103            int u = q[ii];
104            inQueue[u] = false;
105            for (int e: g[u]) {
106                if (edge[e].f == edge[e].c)
107                    continue;
108                int v = edge[e].to;
109                ll nw = edge[e].cost + dist[u];
110                if (nw >= dist[v])
111                    continue;
112                dist[v] = nw;
113                fromEdge[v] = e;
114                if (!inQueue[v]) {
115                    inQueue[v] = true;
116                    q.push_back(v);
117                }
118            }
119        }
120        return dist[T] != infc;
121    }
122
123    ll pot[maxn];
124    bool dijkstra() {
125        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
126            ⇨ greater<pair<ll, int>>> q;
127        for (i, N)
128            dist[i] = infc;
129        dist[S] = 0;
130        q.emplace(dist[S], S);
131        while (!q.empty()) {
132            int u = q.top().second;
133            ll cdist = q.top().first;
134            q.pop();
135            if (cdist != dist[u])
136                continue;
137            for (int e: g[u]) {
138                int v = edge[e].to;
139                if (edge[e].c == edge[e].f)
140                    continue;
141                ll w = edge[e].cost + pot[u] - pot[v];
142                assert(w >= 0);
143                ll ndist = w + dist[u];
144                if (ndist >= dist[v])
145                    continue;
146                dist[v] = ndist;
147                fromEdge[v] = e;
148                q.emplace(dist[v], v);
149            }
150        }
151        if (dist[T] == infc)
152            return false;
153        for (i, N) {
154            if (dist[i] == infc)
155                continue;
156            pot[i] += dist[i];
157        }
158        return true;
159    }
160
161    int run() {
162        for (i, N)
163            dist[i] = infc;
164        dist[S] = 0;
165        inQueue[S] = true;
166        vector<int> q;
167        q.push_back(S);
168        for (int ii = 0; ii < int(q.size()); ++ii) {
169            int u = q[ii];
170            inQueue[u] = false;
171            for (int e: g[u]) {
172                if (edge[e].f == edge[e].c)
173                    continue;
174                int v = edge[e].to;
175                ll nw = edge[e].cost + dist[u];
176                if (nw >= dist[v])
177                    continue;
178                dist[v] = nw;
179                fromEdge[v] = e;
180                if (!inQueue[v]) {
181                    inQueue[v] = true;
182                    q.push_back(v);
183                }
184            }
185        }
186        return dist[T] != infc;
187    }
188
189    ll pot[maxn];
190    bool dijkstra() {
191        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
192            ⇨ greater<pair<ll, int>>> q;
193        for (i, N)
194            dist[i] = infc;
195        dist[S] = 0;
196        q.emplace(dist[S], S);
197        while (!q.empty()) {
198            int u = q.top().second;
199            ll cdist = q.top().first;
200            q.pop();
201            if (cdist != dist[u])
202                continue;
203            for (int e: g[u]) {
204                int v = edge[e].to;
205                if (edge[e].c == edge[e].f)
206                    continue;
207                ll w = edge[e].cost + pot[u] - pot[v];
208                assert(w >= 0);
209                ll ndist = w + dist[u];
210                if (ndist >= dist[v])
211                    continue;
212                dist[v] = ndist;
213                fromEdge[v] = e;
214                q.emplace(dist[v], v);
215            }
216        }
217        if (dist[T] == infc)
218            return false;
219        for (i, N) {
220            if (dist[i] == infc)
221                continue;
222            pot[i] += dist[i];
223        }
224        return true;
225    }
226
227    int run() {
228        for (i, N)
229            dist[i] = infc;
230        dist[S] = 0;
231        inQueue[S] = true;
232        vector<int> q;
233        q.push_back(S);
234        for (int ii = 0; ii < int(q.size()); ++ii) {
235            int u = q[ii];
236            inQueue[u] = false;
237            for (int e: g[u]) {
238                if (edge[e].f == edge[e].c)
239                    continue;
240                int v = edge[e].to;
241                ll nw = edge[e].cost + dist[u];
242                if (nw >= dist[v])
243                    continue;
244                dist[v] = nw;
245                fromEdge[v] = e;
246                if (!inQueue[v]) {
247                    inQueue[v] = true;
248                    q.push_back(v);
249                }
250            }
251        }
252        return dist[T] != infc;
253    }
254
255    ll pot[maxn];
256    bool dijkstra() {
257        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
258            ⇨ greater<pair<ll, int>>> q;
259        for (i, N)
260            dist[i] = infc;
261        dist[S] = 0;
262        q.emplace(dist[S], S);
263        while (!q.empty()) {
264            int u = q.top().second;
265            ll cdist = q.top().first;
266            q.pop();
267            if (cdist != dist[u])
268                continue;
269            for (int e: g[u]) {
270                int v = edge[e].to;
271                if (edge[e].c == edge[e].f)
272                    continue;
273                ll w = edge[e].cost + pot[u] - pot[v];
274                assert(w >= 0);
275                ll ndist = w + dist[u];
276                if (ndist >= dist[v])
277                    continue;
278                dist[v] = ndist;
279                fromEdge[v] = e;
280                q.emplace(dist[v], v);
281            }
282        }
283        if (dist[T] == infc)
284            return false;
285        for (i, N) {
286            if (dist[i] == infc)
287                continue;
288            pot[i] += dist[i];
289        }
290        return true;
291    }
292
293    int run() {
294        for (i, N)
295            dist[i] = infc;
296        dist[S] = 0;
297        inQueue[S] = true;
298        vector<int> q;
299        q.push_back(S);
300        for (int ii = 0; ii < int(q.size()); ++ii) {
301            int u = q[ii];
302            inQueue[u] = false;
303            for (int e: g[u]) {
304                if (edge[e].f == edge[e].c)
305                    continue;
306                int v = edge[e].to;
307                ll nw = edge[e].cost + dist[u];
308                if (nw >= dist[v])
309                    continue;
310                dist[v] = nw;
311                fromEdge[v] = e;
312                if (!inQueue[v]) {
313                    inQueue[v] = true;
314                    q.push_back(v);
315                }
316            }
317        }
318        return dist[T] != infc;
319    }
320
321    ll pot[maxn];
322    bool dijkstra() {
323        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
324            ⇨ greater<pair<ll, int>>> q;
325        for (i, N)
326            dist[i] = infc;
327        dist[S] = 0;
328        q.emplace(dist[S], S);
329        while (!q.empty()) {
330            int u = q.top().second;
331            ll cdist = q.top().first;
332            q.pop();
333            if (cdist != dist[u])
334                continue;
335            for (int e: g[u]) {
336                int v = edge[e].to;
337                if (edge[e].c == edge[e].f)
338                    continue;
339                ll w = edge[e].cost + pot[u] - pot[v];
340                assert(w >= 0);
341                ll ndist = w + dist[u];
342                if (ndist >= dist[v])
343                    continue;
344                dist[v] = ndist;
345                fromEdge[v] = e;
346                q.emplace(dist[v], v);
347            }
348        }
349        if (dist[T] == infc)
350            return false;
351        for (i, N) {
352            if (dist[i] == infc)
353                continue;
354            pot[i] += dist[i];
355        }
356        return true;
357    }
358
359    int run() {
360        for (i, N)
361            dist[i] = infc;
362        dist[S] = 0;
363        inQueue[S] = true;
364        vector<int> q;
365        q.push_back(S);
366        for (int ii = 0; ii < int(q.size()); ++ii) {
367            int u = q[ii];
368            inQueue[u] = false;
369            for (int e: g[u]) {
370                if (edge[e].f == edge[e].c)
371                    continue;
372                int v = edge[e].to;
373                ll nw = edge[e].cost + dist[u];
374                if (nw >= dist[v])
375                    continue;
376                dist[v] = nw;
377                fromEdge[v] = e;
378                if (!inQueue[v]) {
379                    inQueue[v] = true;
380                    q.push_back(v);
381                }
382            }
383        }
384        return dist[T] != infc;
385    }
386
387    ll pot[maxn];
388    bool dijkstra() {
389        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
390            ⇨ greater<pair<ll, int>>> q;
391        for (i, N)
392            dist[i] = infc;
393        dist[S] = 0;
394        q.emplace(dist[S], S);
395        while (!q.empty()) {
396            int u = q.top().second;
397            ll cdist = q.top().first;
398            q.pop();
399            if (cdist != dist[u])
400                continue;
401            for (int e: g[u]) {
402                int v = edge[e].to;
403                if (edge[e].c == edge[e].f)
404                    continue;
405                ll w = edge[e].cost + pot[u] - pot[v];
406                assert(w >= 0);
407                ll ndist = w + dist[u];
408                if (ndist >= dist[v])
409                    continue;
410                dist[v] = ndist;
411                fromEdge[v] = e;
412                q.emplace(dist[v], v);
413            }
414        }
415        if (dist[T] == infc)
416            return false;
417        for (i, N) {
418            if (dist[i] == infc)
419                continue;
420            pot[i] += dist[i];
421        }
422        return true;
423    }
424
425    int run() {
426        for (i, N)
427            dist[i] = infc;
428        dist[S] = 0;
429        inQueue[S] = true;
430        vector<int> q;
431        q.push_back(S);
432        for (int ii = 0; ii < int(q.size()); ++ii) {
433            int u = q[ii];
434            inQueue[u] = false;
435            for (int e: g[u]) {
436                if (edge[e].f == edge[e].c)
437                    continue;
438                int v = edge[e].to;
439                ll nw = edge[e].cost + dist[u];
440                if (nw >= dist[v])
441                    continue;
442                dist[v] = nw;
443                fromEdge[v] = e;
444                if (!inQueue[v]) {
445                    inQueue[v] = true;
446                    q.push_back(v);
447                }
448            }
449        }
450        return dist[T] != infc;
451    }
452
453    ll pot[maxn];
454    bool dijkstra() {
455        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
456            ⇨ greater<pair<ll, int>>> q;
457        for (i, N)
458            dist[i] = infc;
459        dist[S] = 0;
460        q.emplace(dist[S], S);
461        while (!q.empty()) {
462            int u = q.top().second;
463            ll cdist = q.top().first;
464            q.pop();
465            if (cdist != dist[u])
466                continue;
467            for (int e: g[u]) {
468                int v = edge[e].to;
469                if (edge[e].c == edge[e].f)
470                    continue;
471                ll w = edge[e].cost + pot[u] - pot[v];
472                assert(w >= 0);
473                ll ndist = w + dist[u];
474                if (ndist >= dist[v])
475                    continue;
476                dist[v] = ndist;
477                fromEdge[v] = e;
478                q.emplace(dist[v], v);
479            }
480        }
481        if (dist[T] == infc)
482            return false;
483        for (i, N) {
484            if (dist[i] == infc)
485                continue;
486            pot[i] += dist[i];
487        }
488        return true;
489    }
490
491    int run() {
492        for (i, N)
493            dist[i] = infc;
494        dist[S] = 0;
495        inQueue[S] = true;
496        vector<int> q;
497        q.push_back(S);
498        for (int ii = 0; ii < int(q.size()); ++ii) {
499            int u = q[ii];
500            inQueue[u] = false;
501            for (int e: g[u]) {
502                if (edge[e].f == edge[e].c)
503                    continue;
504                int v = edge[e].to;
505                ll nw = edge[e].cost + dist[u];
506                if (nw >= dist[v])
507                    continue;
508                dist[v] = nw;
509                fromEdge[v] = e;
510                if (!inQueue[v]) {
511                    inQueue[v] = true;
512                    q.push_back(v);
513                }
514            }
515        }
516        return dist[T] != infc;
517    }
518
519    ll pot[maxn];
520    bool dijkstra() {
521        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
522            ⇨ greater<pair<ll, int>>> q;
523        for (i, N)
524            dist[i] = infc;
525        dist[S] = 0;
526        q.emplace(dist[S], S);
527        while (!q.empty()) {
528            int u = q.top().second;
529            ll cdist = q.top().first;
530            q.pop();
531            if (cdist != dist[u])
532                continue;
533            for (int e: g[u]) {
534                int v = edge[e].to;
535                if (edge[e].c == edge[e].f)
536                    continue;
537                ll w = edge[e].cost + pot[u] - pot[v];
538                assert(w >= 0);
539                ll ndist = w + dist[u];
540                if (ndist >= dist[v])
541                    continue;
542                dist[v] = ndist;
543                fromEdge[v] = e;
544                q.emplace(dist[v], v);
545            }
546        }
547        if (dist[T] == infc)
548            return false;
549        for (i, N) {
550            if (dist[i] == infc)
551                continue;
552            pot[i] += dist[i];
553        }
554        return true;
555    }
556
557    int run() {
558        for (i, N)
559            dist[i] = infc;
560        dist[S] = 0;
561        inQueue[S] = true;
562        vector<int> q;
563        q.push_back(S);
564        for (int ii = 0; ii < int(q.size()); ++ii) {
565            int u = q[ii];
566            inQueue[u] = false;
567            for (int e: g[u]) {
568                if (edge[e].f == edge[e].c)
569                    continue;
570                int v = edge[e].to;
571                ll nw = edge[e].cost + dist[u];
572                if (nw >= dist[v])
573                    continue;
574                dist[v] = nw;
575                fromEdge[v] = e;
576                if (!inQueue[v]) {
577                    inQueue[v] = true;
578                    q.push_back(v);
579                }
580            }
581        }
582        return dist[T] != infc;
583    }
584
585    ll pot[maxn];
586    bool dijkstra() {
587        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
588            ⇨ greater<pair<ll, int>>> q;
589        for (i, N)
590            dist[i] = infc;
591        dist[S] = 0;
592        q.emplace(dist[S], S);
593        while (!q.empty()) {
594            int u = q.top().second;
595            ll cdist = q.top().first;
596            q.pop();
597            if (cdist != dist[u])
598                continue;
599            for (int e: g[u]) {
600                int v = edge[e].to;
601                if (edge[e].c == edge[e].f)
602                    continue;
603                ll w = edge[e].cost + pot[u] - pot[v];
604                assert(w >= 0);
605                ll ndist = w + dist[u];
606                if (ndist >= dist[v])
607                    continue;
608                dist[v] = ndist;
609                fromEdge[v] = e;
610                q.emplace(dist[v], v);
611            }
612        }
613        if (dist[T] == infc)
614            return false;
615        for (i, N) {
616            if (dist[i] == infc)
617                continue;
618            pot[i] += dist[i];
619        }
620        return true;
621    }
622
623    int run() {
624        for (i, N)
625            dist[i] = infc;
626        dist[S] = 0;
627        inQueue[S] = true;
628        vector<int> q;
629        q.push_back(S);
630        for (int ii = 0; ii < int(q.size()); ++ii) {
631            int u = q[ii];
632            inQueue[u] = false;
633            for (int e: g[u]) {
634                if (edge[e].f == edge[e].c)
635                    continue;
636                int v = edge[e].to;
637                ll nw = edge[e].cost + dist[u];
638                if (nw >= dist[v])
639                    continue;
640                dist[v] = nw;
641                fromEdge[v] = e;
642                if (!inQueue[v]) {
643                    inQueue[v] = true;
644                    q.push_back(v);
645                }
646            }
647        }
648        return dist[T] != infc;
649    }
650
651    ll pot[maxn];
652    bool dijkstra() {
653        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
654            ⇨ greater<pair<ll, int>>> q;
655        for (i, N)
656            dist[i] = infc;
657        dist[S] = 0;
658        q.emplace(dist[S], S);
659        while (!q.empty()) {
660            int u = q.top().second;
661            ll cdist = q.top().first;
662            q.pop();
663            if (cdist != dist[u])
664                continue;
665            for (int e: g[u]) {
666                int v = edge[e].to;
667                if (edge[e].c == edge[e].f)
668                    continue;
669                ll w = edge[e].cost + pot[u] - pot[v];
670                assert(w >= 0);
671                ll ndist = w + dist[u];
672                if (ndist >= dist[v])
673                    continue;
674                dist[v] = ndist;
675                fromEdge[v] = e;
676                q.emplace(dist[v], v);
677            }
678        }
679        if (dist[T] == infc)
680            return false;
681        for (i, N) {
682            if (dist[i] == infc)
683                continue;
684            pot[i] += dist[i];
685        }
686        return true;
687    }
688
689    int run() {
690        for (i, N)
691            dist[i] = infc;
692        dist[S] = 0;
693        inQueue[S] = true;
694        vector<int> q;
695        q.push_back(S);
696        for (int ii = 0; ii < int(q.size()); ++ii) {
697            int u = q[ii];
698            inQueue[u] = false;
699            for (int e: g[u]) {
700                if (edge[e].f == edge[e].c)
701                    continue;
702                int v = edge[e].to;
703                ll nw = edge[e].cost + dist[u];
704                if (nw >= dist[v])
705                    continue;
706                dist[v] = nw;
707                fromEdge[v] = e;
708                if (!inQueue[v]) {
709                    inQueue[v] = true;
710                    q.push_back(v);
711                }
712            }
713        }
714        return dist[T] != infc;
715    }
716
717    ll pot[maxn];
718    bool dijkstra() {
719        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
720            ⇨ greater<pair<ll, int>>> q;
721        for (i, N)
722            dist[i] = infc;
723        dist[S] = 0;
724        q.emplace(dist[S], S);
725        while (!q.empty()) {
726            int u = q.top().second;
727            ll cdist = q.top().first;
728            q.pop();
729            if (cdist != dist[u])
730                continue;
731            for (int e: g[u]) {
732                int v = edge[e].to;
733                if (edge[e].c == edge[e].f)
734                    continue;
735                ll w = edge[e].cost + pot[u] - pot[v];
736                assert(w >= 0);
737                ll ndist = w + dist[u];
738                if (ndist >= dist[v])
739                    continue;
740                dist[v] = ndist;
741                fromEdge[v] = e;
742                q.emplace(dist[v], v);
743            }
744        }
745        if (dist[T] == infc)
746            return false;
747        for (i, N) {
748            if (dist[i] == infc)
749                continue;
750            pot[i] += dist[i];
751        }
752        return true;
753    }
754
755    int run() {
756        for (i, N)
757            dist[i] = infc;
758        dist[S] = 0;
759        inQueue[S] = true;
760        vector<int> q;
761        q.push_back(S);
762        for (int ii = 0; ii < int(q.size()); ++ii) {
763            int u = q[ii];
764            inQueue[u] = false;
765            for (int e: g[u]) {
766                if (edge[e].f == edge[e].c)
767                    continue;
768                int v = edge[e].to;
769                ll nw = edge[e].cost + dist[u];
770                if (nw >= dist[v])
771                    continue;
772                dist[v] = nw;
773                fromEdge[v] = e;
774                if (!inQueue[v]) {
775                    inQueue[v] = true;
776                    q.push_back(v);
777                }
778            }
779        }
780        return dist[T] != infc;
781    }
782
783    ll pot[maxn];
784    bool dijkstra() {
785        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
786            ⇨ greater<pair<ll, int>>> q;
787        for (i, N)
788            dist[i] = infc;
789        dist[S] = 0;
790        q.emplace(dist[S], S);
791        while (!q.empty()) {
792            int u = q.top().second;
793            ll cdist = q.top().first;
794            q.pop();
795            if (cdist != dist[u])
796                continue;
797            for (int e: g[u]) {
798                int v = edge[e].to;
799                if (edge[e].c == edge[e].f)
800                    continue;
801                ll w = edge[e].cost + pot[u] - pot[v];
802                assert(w >= 0);
803                ll ndist = w + dist[u];
804                if (ndist >= dist[v])
805                    continue;
806                dist[v] = ndist;
807                fromEdge[v] = e;
808                q.emplace(dist[v], v);
809            }
810        }
811        if (dist[T] == infc)
812            return false;
813        for (i, N) {
814            if (dist[i] == infc)
815                continue;
816            pot[i] += dist[i];
817        }
818        return true;
819    }
820
821    int run() {
822        for (i, N)
823            dist[i] = infc;
824        dist[S] = 0;
825        inQueue[S] = true;
826        vector<int> q;
827        q.push_back(S);
828        for (int ii = 0; ii < int(q.size()); ++ii) {
829            int u = q[ii];
830            inQueue[u] = false;
831            for (int e: g[u]) {
832                if (edge[e].f == edge[e].c)
833                    continue;
834                int v = edge[e].to;
835                ll nw = edge[e].cost + dist[u];
836                if (nw >= dist[v])
837                    continue;
838                dist[v] = nw;
839                fromEdge[v] = e;
840                if (!inQueue[v]) {
841                    inQueue[v] = true;
842                    q.push_back(v);
843                }
844            }
845        }
846        return dist[T] != infc;
847    }
848
849    ll pot[maxn];
850    bool dijkstra() {
851        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
852            ⇨ greater<pair<ll, int>>> q;
853        for (i, N)
854            dist[i] = infc;
855        dist[S] = 0;
856        q.emplace(dist[S], S);
857        while (!q.empty()) {
858            int u = q.top().second;
859            ll cdist = q.top().first;
860            q.pop();
861            if (cdist != dist[u])
862                continue;
863            for (int e: g[u]) {
864                int v = edge[e].to;
865                if (edge[e].c == edge[e].f)
866                    continue;
867                ll w = edge[e].cost + pot[u] - pot[v];
868                assert(w >= 0);
869                ll ndist = w + dist[u];
870                if (ndist >= dist[v])
871                    continue;
872                dist[v] = ndist;
873                fromEdge[v] = e;
874                q.emplace(dist[v], v);
875            }
876        }
877        if (dist[T] == infc)
878            return false;
879        for (i, N) {
880            if (dist[i] == infc)
881                continue;
882            pot[i] += dist[i];
883        }
884        return true;
885    }
886
887    int run() {
888        for (i, N)
889            dist[i] = infc;
890        dist[S] = 0;
891        inQueue[S] = true;
892        vector<int> q;
893        q.push_back(S);
894        for (int ii = 0; ii < int(q.size()); ++ii) {
895            int u = q[ii];
896            inQueue[u] = false;
897            for (int e: g[u]) {
898                if (edge[e].f == edge[e].c)
899                    continue;
900                int v = edge[e].to;
901                ll nw = edge[e].cost + dist[u];
902                if (nw >= dist[v])
903                    continue;
904                dist[v] = nw;
905                fromEdge[v] = e;
906                if (!inQueue[v]) {
907                    inQueue[v] = true;
908                    q.push_back(v);
909                }
910            }
911        }
912        return dist[T] != infc;
913    }
914
915    ll pot[maxn];
916    bool dijkstra() {
917        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
918            ⇨ greater<pair<ll, int>>> q;
919        for (i, N)
920            dist[i] = infc;
921        dist[S] = 0;
922        q.emplace(dist[S], S);
923        while (!q.empty()) {
924            int u = q.top().second;
925            ll cdist = q.top().first;
926            q.pop();
927            if (cdist != dist[u])
928                continue;
929            for (int e: g[u]) {
930                int v = edge[e].to;
931                if (edge[e].c == edge[e].f)
932                    continue;
933                ll w = edge[e].cost + pot[u] - pot[v];
934                assert(w >= 0);
935                ll ndist = w + dist[u];
936                if (ndist >= dist[v])
937                    continue;
938                dist[v] = ndist;
939                fromEdge[v] = e;
940                q.emplace(dist[v], v);
941            }
942        }
943        if (dist[T] == infc)
944            return false;
945        for (i, N) {
946            if (dist[i] == infc)
947                continue;
948            pot[i] += dist[i];
949        }
950        return true;
951    }
952
953    int run() {
954        for (i, N)
955            dist[i] = infc;
956        dist[S] = 0;
957        inQueue[S] = true;
958        vector<int> q;
959        q.push_back(S);
960        for (int ii = 0; ii < int(q.size()); ++ii) {
961            int u = q[ii];
962            inQueue[u] = false;
963            for (int e: g[u]) {
964                if (edge[e].f == edge[e].c)
965                    continue;
966                int v = edge[e].to;
967                ll nw = edge[e].cost + dist[u];
968                if (nw >= dist[v])
969                    continue;
970                dist[v] = nw;
971                fromEdge[v] = e;
972                if (!inQueue[v]) {
973                    inQueue[v] = true;
974                    q.push_back(v);
975                }
976            }
977        }
978        return dist[T] != infc;
979    }
980
981    ll pot[maxn];
982    bool dijkstra() {
983        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
984            ⇨ greater<pair<ll, int>>> q;
985        for (i, N)
986            dist[i] = infc;
987        dist[S] = 0;
988        q.emplace(dist[S], S);
989        while (!q.empty()) {
990            int u = q.top().second;
991            ll cdist = q.top().first;
992            q.pop();
993            if (cdist != dist[u])
994                continue;
995            for (int e: g[u]) {
996                int v = edge[e].to;
997                if (edge[e].c == edge[e].f)
998                    continue;
999                ll w = edge[e].cost + pot[u] - pot[v];

```

```

95 bool push() {
96     //2 variants
97     //if (!fordBellman())
98     if (!dikstra())
99         return false;
100     ++totalFlow;
101     int u = T;
102     while (u != S) {
103         int e = fromEdge[u];
104         totalCost += edge[e].cost;
105         edge[e].f++;
106         edge[e ^ 1].f--;
107         u = edge[e ^ 1].to;
108     }
109     return true;
110 }
111};
112
113int main() {
114    MinCost::N = 3, MinCost::S = 1, MinCost::T = 2;
115    MinCost::addEdge(1, 0, 3, 5);
116    MinCost::addEdge(0, 2, 4, 6);
117    while (MinCost::push());
118    cout << MinCost::totalFlow << ' ' << MinCost::totalCost << '\n';
119}

```

## 6 geometry/halfplanes.cpp

```

1#include <bits/stdc++.h>
2using namespace std;
3#define forn(i, n) for (int i = 0; i < int(n); ++i)
4#define forab(i, a, b) for (int i = int(a); i < int(b); ++i)
5#include "primitives.cpp"
6
7ld det3x3(line &l1, line &l2, line &l3) {
8    return l1.a * (l2.b * l3.c - l2.c * l3.b) +
9        l1.b * (l2.c * l3.a - l2.a * l3.c) +
10        l1.c * (l2.a * l3.b - l2.b * l3.a);
11}
12
13vector<pt> halfplanesIntersecion(vector<line> lines) {
14    sort(lines.begin(), lines.end(), [](const line &a, const line &b) {
15        bool ar = a.right(), br = b.right();
16        if (ar ^ br)
17            return ar;
18        ld prod = (pt{a.a, a.b} % pt{b.a, b.b});
19        if (!eq(prod, 0))
20            return prod > 0;
21        return a.c < b.c;
22    });
23    vector<line> lines2;
24    pt pr;
25    forn(i, lines.size()) {
26        pt cur{lines[i].a, lines[i].b};
27        if (i == 0 || cur != pr)
28            lines2.push_back(lines[i]);
29        pr = cur;
30    }
31    lines = lines2;
32    int n = lines.size();
33    forn(i, n)
34        lines[i].id = i;
35    vector<line> hull;
36    forn(i, 2 * n) {
37        line l = lines[i % n];
38        while ((int) hull.size() >= 2) {
39            ld D = det3x3(*prev(prev(hull.end())), hull.back(), l);
40            if (ge(D, 0))
41                break;
42            hull.pop_back();
43        }
44        hull.push_back(l);
45    }
46    vector<int> firstTime(n, -1);
47    vector<line> v;
48    forn(i, hull.size()) {
49        int cid = hull[i].id;
50        if (firstTime[cid] == -1) {
51            firstTime[cid] = i;
52            continue;
53        }
54        forab(j, firstTime[cid], i)
55            v.push_back(hull[j]);
56        break;
57    }
58    n = v.size();
59    if (v.empty()) {
60        //empty intersection
61        return {};
62    }
63    v.push_back(v[0]);
64    vector<pt> res;
65    pt center{0, 0};
66    forn(i, n) {
67        res.push_back(halfplanesIntersecion(v[i], v[i + 1]));
68        center = center + res.back();
69    }
70    center = center / n;
71    for (auto l: lines)
72        if (lt(l.signedDist(center), 0)) {
73            //empty intersection
74            return {};
75        }
76    return res;
77}

```

## 7 geometry/primitives.cpp

```
1#include <bits/stdc++.h>
2#define forn(i, n) for (int i = 0; i < int(n); ++i)
3using namespace std;
4typedef long double ld;
5
6const ld eps = 1e-9;
7
8bool eq(ld a, ld b) { return fabs1(a - b) < eps; }
9bool le(ld a, ld b) { return b - a > -eps; }
10bool ge(ld a, ld b) { return a - b > -eps; }
11bool lt(ld a, ld b) { return b - a > eps; }
12bool gt(ld a, ld b) { return a - b > eps; }
13ld sqr(ld x) { return x * x; }
14
15#ifdef LOCAL
16#define gassert assert
17#else
18void gassert(bool) {}
19#endif
20
21struct pt {
22    ld x, y;
23
24    pt operator+(const pt &p) const { return pt{x + p.x, y + p.y}; }
25    pt operator-(const pt &p) const { return pt{x - p.x, y - p.y}; }
26    ld operator*(const pt &p) const { return x * p.x + y * p.y; }
27    ld operator%(const pt &p) const { return x * p.y - y * p.x; }
28
29    pt operator*(const ld &a) const { return pt{x * a, y * a}; }
30    pt operator/(const ld &a) const { gassert(!eq(a, 0)); return pt{x
    ↪ a, y / a}; }
31    void operator*=(const ld &a) { x *= a, y *= a; }
32    void operator/=(const ld &a) { gassert(!eq(a, 0)); x /= a, y /= a;
    ↪ }
33
34    bool operator<(const pt &p) const {
35        if (eq(x, p.x)) return lt(y, p.y);
36        return x < p.x;
37    }
38
39    bool operator==(const pt &p) const { return eq(x, p.x) && eq(y,
    ↪ p.y); }
40    bool operator!=(const pt &p) const { return !(*this == p); }
41
42    pt rot() { return pt{-y, x}; }
43    ld abs() const { return hypot1(x, y); }
44    ld abs2() const { return x * x + y * y; }
45};
46
47istream &operator>>(istream &in, pt &p) { return in >> p.x >> p.y; }
48ostream &operator<<(ostream &out, const pt &p) { return out << p.x <<
    ↪ ' ' << p.y; }
49
50//WARNING! do not forget to normalize vector (a,b)
51struct line {
52    ld a, b, c;
53    int id;
54
55    line(pt p1, pt p2) {
56        gassert(p1 != p2);
57        pt n = (p2 - p1).rot();
58        n /= n.abs();
59        a = n.x, b = n.y;
60        c = -(n * p1);
61    }
62
63    bool right() const {
64        return gt(a, 0) || (eq(a, 0) && gt(b, 0));
65    }
66
67    line(ld _a, ld _b, ld _c): a(_a), b(_b), c(_c) {
68        ld d = pt{a, b}.abs();
69        gassert(!eq(d, 0));
70        a /= d, b /= d, c /= d;
71    }
72
73    ld signedDist(pt p) {
74        return p * pt{a, b} + c;
75    }
76};
77
78ld pointSegmentDist(pt p, pt a, pt b) {
79    ld res = min((p - a).abs(), (p - b).abs());
80    if (a != b && ge((p - a) * (b - a), 0) && ge((p - b) * (a - b), 0))
81        res = min(res, fabs1((p - a) % (b - a)) / (b - a).abs());
82    return res;
83}
84
85pt linesIntersection(line l1, line l2) {
86    ld D = l1.a * l2.b - l1.b * l2.a;
87    if (eq(D, 0)) {
88        if (eq(l1.c, l2.c)) {
89            //equal lines
90        } else {
91            //no intersection
92        }
93    }
94    ld dx = -l1.c * l2.b + l1.b * l2.c;
95    ld dy = -l1.a * l2.c + l1.c * l2.a;
96    pt res{dx / D, dy / D};
97    //gassert(eq(l1.signedDist(res), 0));
98    //gassert(eq(l2.signedDist(res), 0));
99    return res;
100}
101
102bool pointInsideSegment(pt p, pt a, pt b) {
103    if (!eq((p - a) % (b - a), 0))
104        return false;
105    return le((a - p) * (b - p), 0);
106}
107
108bool checkSegmentIntersection(pt a, pt b, pt c, pt d) {
109    if (eq((a - b) % (c - d), 0)) {
110        if (pointInsideSegment(a, c, d) || pointInsideSegment(b, c, d)
    ↪ ||
111            pointInsideSegment(c, a, b) || pointInsideSegment(d, a,
    ↪ b)) {
112            //intersection of parallel segments
113            return true;
114        }
115        return false;
116    }
117
118    ld s1, s2;
119
120    s1 = (c - a) % (b - a);
121    s2 = (d - a) % (b - a);
122    if (gt(s1, 0) && gt(s2, 0))
123        return false;
124    if (lt(s1, 0) && lt(s2, 0))
125        return false;
126
127    swap(a, c), swap(b, d);
128
129    s1 = (c - a) % (b - a);
130    s2 = (d - a) % (b - a);
131    if (gt(s1, 0) && gt(s2, 0))
132        return false;
133    if (lt(s1, 0) && lt(s2, 0))
134        return false;
135
136    return true;
137}
138
139//WARNING! run checkSegmentIntersection before and process parallel case
    ↪ manually
140pt segmentsIntersection(pt a, pt b, pt c, pt d) {
141    ld S = (b - a) % (d - c);
142    ld s1 = (c - a) % (d - a);
143    return a + (b - a) / S * s1;
144}
145
146vector<pt> circlesIntersction(pt a, ld r1, pt b, ld r2) {
147    ld d2 = (a - b).abs2();
148    ld d = (a - b).abs();
149
150    if (a == b && eq(r1, r2)) {
151        //equal circles
152    }
153    if (lt(sqr(r1 + r2), d2) || gt(sqr(r1 - r2), d2)) {
154        //empty intersection
155        return {};
156    }
157    int num = 2;
158    if (eq(sqr(r1 + r2), d2) || eq(sqr(r1 - r2), d2))
159        num = 1;
160    ld cosa = (sqr(r1) + d2 - sqr(r2)) / ld(2 * r1 * d);
161    ld oh = cosa * r1;
162    pt h = a + ((b - a) / d * oh);
163    if (num == 1)
164        return {h};
165    ld hp = sqrt1(max(0.L, 1 - cosa * cosa)) * r1;
166
167    pt w = ((b - a) / d * hp).rot();
168    return {h + w, h - w};
169}
170
171//a is circle center, p is point
172vector<pt> circleTangents(pt a, ld r, pt p) {
173    ld d2 = (a - p).abs2();
174    ld d = (a - p).abs();
175
176    if (gt(sqr(r), d2)) {
177        //no tangents
178        return {};
179    }
180    if (eq(sqr(r), d2)) {
181        //point lies on circle - one tangent
182        return {p};
183    }
184
185    pt B = p - a;
186    pt H = B * sqr(r) / d2;
187    ↵
```

```

187 ld h = sqrtl(d2 - sqr(r)) * ld(r) / d;
188 pt w = (B / d * h).rot();
189 H = H + a;
190 return {H + w, H - w};
191}
192
193vector<pt> lineCircleIntersection(line l, pt a, ld r) {
194    ld d = l.signedDist(a);
195    if (gt(fabsl(d), r))
196        return {};
197    pt h = a - pt{l.a, l.b} * d;
198    if (eq(fabsl(d), r))
199        return {h};
200    pt w = pt{l.a, l.b}.rot() * sqrtl(max<ld>(0, sqr(r) - sqr(d)));
201    return {h + w, h - w};
202}
203
204//modified magic from e-maxx
205vector<line> commonTangents(pt a, ld r1, pt b, ld r2) {
206    if (a == b && eq(r1, r2)) {
207        //equal circles
208        return {};
209    }
210    vector<line> res;
211    pt c = b - a;
212    ld z = c.abs2();
213    for (int i = -1; i <= 1; i += 2)
214        for (int j = -1; j <= 1; j += 2) {
215            ld r = r2 * j - r1 * i;
216            ld d = z - sqr(r);
217            if (lt(d, 0))
218                continue;
219            d = sqrtl(max<ld>(0, d));
220            pt magic = pt{r, d} / z;
221            line l(magic * c, magic % c, r1 * i);
222            l.c -= pt{l.a, l.b} * a;
223            res.push_back(l);
224        }
225    return res;
226}

```

## 8 geometry/svg.cpp

```

1struct SVG {
2    FILE *out;
3    ld sc = 50;
4
5    void open() {
6        out = fopen("image.svg", "w");
7        fprintf(out, "<svg xmlns='http://www.w3.org/2000/svg'
↪ viewBox='-1000 -1000 2000 2000'>\n");
8    }
9
10    void line(pt a, pt b) {
11        a = a * sc, b = b * sc;
12        fprintf(out, "<line x1='%Lf' y1='%Lf' x2='%Lf' y2='%Lf'
↪ stroke='black'/>\n", a.x, -a.y, b.x, -b.y);
13    }
14
15    void circle(pt a, ld r = -1, string col = "red") {
16        r = (r == -1 ? 10 : sc * r);
17        a = a * sc;
18        fprintf(out, "<circle cx='%Lf' cy='%Lf' r='%Lf' fill='%s'/>\n",
↪ a.x, -a.y, r, col.c_str());
19    }
20
21    void text(pt a, string s) {
22        a = a * sc;
23        fprintf(out, "<text x='%Lf' y='%Lf'
↪ font-size='10px'%>s</text>\n", a.x, -a.y, s.c_str());
24    }
25
26    void close() {
27        fprintf(out, "</svg>\n");
28        fclose(out);
29    }
30
31    ~SVG() {
32        if (out)
33            close();
34    }
35} svg;

```

## 9 graphs/2sat.cpp

```

1const int maxn = 200100; //2 x number of variables
2
3namespace TwoSAT {
4    int n; //number of variables
5    bool used[maxn];
6    vector<int> g[maxn];
7    vector<int> gr[maxn];
8    int comp[maxn];
9    int res[maxn];
10
11    void addEdge(int u, int v) { //u or v
12        g[u].push_back(v ^ 1);
13        g[v].push_back(u ^ 1);
14        gr[u ^ 1].push_back(v);
15        gr[v ^ 1].push_back(u);
16    }
17
18    vector<int> ord;
19    void dfs1(int u) {
20        used[u] = true;
21        for (int v: g[u]) {
22            if (used[v])
23                continue;
24            dfs1(v);
25        }
26        ord.push_back(u);
27    }
28
29    int COL = 0;
30    void dfs2(int u) {
31        used[u] = true;
32        comp[u] = COL;
33        for (int v: gr[u]) {
34            if (used[v])
35                continue;
36            dfs2(v);
37        }
38    }
39
40    void mark(int u) {
41        res[u / 2] = u % 2;
42        used[u] = true;
43        for (int v: g[u]) {
44            if (used[v])
45                continue;
46            mark(v);
47        }
48    }
49
50    bool run() {
51        fill(res, res + 2 * n, -1);
52        fill(used, used + 2 * n, false);
53        forn(i, 2 * n)
54            if (!used[i])
55                dfs1(i);
56        reverse(ord.begin(), ord.end());
57        assert((int) ord.size() == (2 * n));
58        fill(used, used + 2 * n, false);
59        for (int u: ord) if (!used[u]) {
60            dfs2(u);
61            ++COL;
62        }
63        forn(i, n)
64            if (comp[i * 2] == comp[i * 2 + 1])
65                return false;
66
67        reverse(ord.begin(), ord.end());
68        fill(used, used + 2 * n, false);
69        for (int u: ord) {
70            if (res[u / 2] != -1) {
71                continue;
72            }
73            mark(u);
74        }
75        return true;
76    }
77};
78
79int main() {
80    TwoSAT::n = 2;
81    TwoSAT::addEdge(0, 2); //x or y
82    TwoSAT::addEdge(0, 3); //x or !y
83    TwoSAT::addEdge(3, 3); //!y or !y
84    assert(TwoSAT::run());
85    cout << TwoSAT::res[0] << ' ' << TwoSAT::res[1] << '\n'; //1 0
86}

```

## 10 graphs/directed\_mst.cpp

```

1// WARNING: this code wasn't submitted anywhere
2
3namespace TwoChinese {
4
5    struct Edge {
6        int to, w, id;
7        bool operator<(const Edge& other) const {
8            return to < other.to || (to == other.to && w < other.w);
9        }
10};
11typedef vector<vector<Edge>> Graph;
12
13const int maxn = 2050;
14
15// global, for supplementary algorithms
16int b[maxn];
17int tin[maxn], tup[maxn];
18int dtim; // counter for tin, tout
19vector<int> st;
20int nc; // number of strongly connected components
21int q[maxn];
22
23int answer;
24
25void tarjan(int v, const Graph& e, vector<int>& comp) {
26    b[v] = 1;
27    st.push_back(v);
28    tin[v] = tup[v] = dtim++;
29
30    for (Edge t: e[v]) if (t.w == 0) {
31        int to = t.to;
32        if (b[to] == 0) {
33            tarjan(to, e, comp);
34            tup[v] = min(tup[v], tup[to]);
35        } else if (b[to] == 1) {
36            tup[v] = min(tup[v], tin[to]);
37        }
38    }
39
40    if (tin[v] == tup[v]) {
41        while (true) {
42            int t = st.back();
43            st.pop_back();
44            comp[t] = nc;
45            b[t] = 2;
46            if (t == v) break;
47        }
48        ++nc;
49    }
50}
51
52vector<Edge> bfs(
53    const Graph& e, const vector<int>& init, const vector<int>& comp)
54{
55    int n = e.size();
56    forn(i, n) b[i] = 0;
57    int lq = 0, rq = 0;
58    for (int v: init) b[v] = 1, q[rq++] = v;
59
60    vector<Edge> result;
61
62    while (lq != rq) {
63        int v = q[lq++];
64        for (Edge t: e[v]) if (t.w == 0) {
65            int to = t.to;
66            if (b[to]) continue;
67            if (!comp.empty() && comp[v] != comp[to]) continue;
68            b[to] = 1;
69            q[rq++] = to;
70            result.push_back(t);
71        }
72    }
73
74    return result;
75}
76
77// warning: check that each vertex is reachable from root
78vector<Edge> run(Graph e, int root) {
79    int n = e.size();
80
81    // find minimum incoming weight for each vertex
82    vector<int> minw(n, inf);
83    forn(v, n) for (Edge t: e[v]) {
84        minw[t.to] = min(minw[t.to], t.w);
85    }
86    forn(v, n) for (Edge &t: e[v]) if (t.to != root) {
87        t.w -= minw[t.to];
88    }
89    forn(i, n) if (i != root) answer += minw[i];
90
91    // check if each vertex is reachable from root by zero edges
92    vector<Edge> firstResult = bfs(e, {root}, {});
93    if ((int)firstResult.size() + 1 == n) {
94        return firstResult;
95    }
96}

```

```

96
97 // find strongly connected components and build compressed graph
98 vector<int> comp(n);
99 forn(i, n) b[i] = 0;
100 nc = 0;
101 dtime = 0;
102 forn(i, n) if (!b[i]) tarjan(i, e, comp);
103
104 // multiple edges may be removed here if needed
105 Graph ne(nc);
106 forn(v, n) for (Edge t: e[v]) {
107     if (comp[v] != comp[t.to]) {
108         ne[comp[v]].push_back({comp[t.to], t.w, t.id});
109     }
110 }
111
112 // run recursively on compressed graph
113 vector<Edge> subres = run(ne, comp[root]);
114
115 // find incoming edge id for each component, init queue
116 // if there is an edge (u, v) between different components
117 // than v is added to queue
118 vector<int> incomingId(nc);
119 for (Edge e: subres) {
120     incomingId[e.to] = e.id;
121 }
122
123 vector<Edge> result;
124 vector<int> init;
125 init.push_back(root);
126 forn(v, n) for (Edge t: e[v]) {
127     if (incomingId[comp[t.to]] == t.id) {
128         result.push_back(t);
129         init.push_back(t.to);
130     }
131 }
132
133 // run bfs to add edges inside components and return answer
134 vector<Edge> innerEdges = bfs(e, init, comp);
135 result.insert(result.end(), all(innerEdges));
136
137 assert((int)result.size() + 1 == n);
138 return result;
139}
140
141} // namespace TwoChinese
142
143void test () {
144    auto res = TwoChinese::run({
145        {{1,5,0},{2,5,1}},
146        {{3,1,2}},
147        {{1,2,3},{4,1,4}},
148        {{1,1,5},{4,2,6}},
149        {{2,1,7}}},
150    0);
151    cout << TwoChinese::answer << endl;
152    for (auto e: res) cout << e.id << " ";
153    cout << endl;
154    // 9      0 6 2 7
155}

```

## 11 math/fft\_recursive.cpp

```

1 const int sz = 1<<20;
2
3 int revb[sz];
4 vector<base> ang[21];
5
6 void init(int n) {
7     int lg = 0;
8     while ((1<<lg) != n) {
9         ++lg;
10    }
11    forn(i, n) {
12        revb[i] = (revb[i>>1]>>1)^((i&1)<<(lg-1));
13    }
14
15    ld e = M_PI * 2 / n;
16    ang[lg].resize(n);
17    forn(i, n) {
18        ang[lg][i] = { cos(e * i), sin(e * i) };
19    }
20
21    for (int k = lg - 1; k >= 0; --k) {
22        ang[k].resize(1 << k);
23        forn(i, 1<<k) {
24            ang[k][i] = ang[k+1][i*2];
25        }
26    }
27}
28
29 void fft_rec(base *a, int lg, bool rev) {
30     if (lg == 0) {
31         return;
32     }
33     int len = 1 << (lg - 1);
34     fft_rec(a, lg-1, rev);
35     fft_rec(a+len, lg-1, rev);
36
37     forn(i, len) {
38         base w = ang[lg][i];
39         if (rev) w.im *= -1;
40         base u = a[i];
41         base v = a[i+len] * w;
42         a[i] = u + v;
43         a[i+len] = u - v;
44     }
45}
46
47 void fft(base *a, int n, bool rev) {
48     forn(i, n) {
49         int j = revb[i];
50         if (i < j) swap(a[i], a[j]);
51     }
52     int lg = 0;
53     while ((1<<lg) != n) {
54         ++lg;
55     }
56     fft_rec(a, lg, rev);
57     if (rev) forn(i, n) {
58         a[i] = a[i] * (1.0 / n);
59     }
60}
61
62 const int maxn = 1050000;
63
64 int n;
65 base a[maxn];
66 base b[maxn];
67
68 void test() {
69     int n = 8;
70     init(n);
71     base a[8] = {1,3,5,2,4,6,7,1};
72     fft(a, n, 0);
73     forn(i, n) cout << a[i].re << " "; cout << endl;
74     forn(i, n) cout << a[i].im << " "; cout << endl;
75     // 29 -5.82843 -7 -0.171573 5 -0.171573 -7 -5.82843
76     // 0 -3.41421 6 0.585786 0 -0.585786 -6 3.41421
77}

```



## 12 math/golden\_search.cpp

```
1ld f(ld x) {
2    return 5 * x * x + 100 * x + 1; // -10 is minimum
3}
4
5ld goldenSearch(ld l, ld r) {
6    ld phi = (1 + sqrtl(5)) / 2;
7    ld resphi = 2 - phi;
8    ld x1 = l + resphi * (r - l);
9    ld x2 = r - resphi * (r - l);
10   ld f1 = f(x1);
11   ld f2 = f(x2);
12   forn(iter, 60) {
13       if (f1 < f2) {
14           r = x2;
15           x2 = x1;
16           f2 = f1;
17           x1 = l + resphi * (r - l);
18           f1 = f(x1);
19       } else {
20           l = x1;
21           x1 = x2;
22           f1 = f2;
23           x2 = r - resphi * (r - l);
24           f2 = f(x2);
25       }
26   }
27   return (x1 + x2) / 2;
28}
29
30int main() {
31    std::cout << goldenSearch(-100, 100) << '\n';
32}
```

## 13 math/numbers.txt

Simpson's numerical integration:  
integral from a to b  $f(x) dx =$   
 $(b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))$

Gauss 5-th order numerical integration:  
integral from -1 to 1  
 $x_1, x_3 = \pm \sqrt{0.6}$ ,  $x_2 = 0$   
 $a_1, a_3 = 5/9$ ,  $a_2 = 8/9$

large primes:  $10^{18} + 3$ ,  $+31$ ,  $+3111$

fft modules for  $2^{**20}$ :  
7340033 13631489 26214401 28311553 70254593  
976224257 (largest less than  $10^{**9}$ )

fibonacci numbers:  
1, 2: 1  
45: 1134903170  
46: 1836311903 (max int)  
47: 2971215073 (max unsigned)  
91: 4660046610375530309  
92: 7540113804746346429 (max i64)  
93: 12200160415121876738 (max unsigned i64)

$2^{**31} = 2147483648 = 2.1e9$   
 $2^{**32} = 4294967296 = 4.2e9$   
 $2^{**63} = 9223372036854775808 = 9.2e18$   
 $2^{**64} = 18446744073709551616 = 1.8e19$

highly composite: todo

## 14 strings/automaton.cpp

```

1 int t[maxn][26], lnk[maxn], len[maxn];
2 int sz;
3 int last;
4
5 void init() {
6     sz = 3;
7     last = 1;
8     forn(i, 26) t[2][i] = 1;
9     len[2] = -1;
10    lnk[1] = 2;
11}
12
13 void addchar(int c) {
14     int nlast = sz++;
15     len[nlast] = len[last] + 1;
16     int p = last;
17     for (; !t[p][c]; p = lnk[p]) {
18         t[p][c] = nlast;
19     }
20     int q = t[p][c];
21     if (len[p] + 1 == len[q]) {
22         lnk[nlast] = q;
23     } else {
24         int clone = sz++;
25         len[clone] = len[p] + 1;
26         lnk[clone] = lnk[q];
27         lnk[q] = lnk[nlast] = clone;
28         forn(i, 26) t[clone][i] = t[q][i];
29         for (; t[p][c] == q; p = lnk[p]) {
30             t[p][c] = clone;
31         }
32     }
33     last = nlast;
34}
35
36 bool check(const string& s) {
37     int v = 1;
38     for (int c: s) {
39         c -= 'a';
40         if (!t[v][c]) return false;
41         v = t[v][c];
42     }
43     return true;
44}
45
46 int main() {
47     string s;
48     cin >> s;
49     init();
50     for (int i: s) {
51         addchar(i - 'a');
52     }
53     forn(i, s.length()) {
54         assert(check(s.substr(i)));
55     }
56     cout << sz << endl;
57     return 0;
58}

```

## 15 strings/eertree.cpp

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 const int maxn = 5000100;
4 const int inf = 1e9 + 1e5;
5
6 char buf[maxn];
7 char *s = buf + 1;
8 int to[maxn][2];
9 int suff[maxn];
10 int len[maxn];
11 int sz;
12 int last;
13
14 const int odd = 1;
15 const int even = 2;
16 const int blank = 3;
17
18 inline void go(int &u, int pos) {
19     while (u != blank && s[pos - len[u] - 1] != s[pos])
20         u = suff[u];
21 }
22
23 void add_char(int pos) {
24     go(last, pos);
25     int u = suff[last];
26     go(u, pos);
27     int c = s[pos] - 'a';
28     if (!to[last][c]) {
29         to[last][c] = sz++;
30         len[sz - 1] = len[last] + 2;
31         assert(to[u][c]);
32         suff[sz - 1] = to[u][c];
33     }
34     last = to[last][c];
35}
36
37 void init() {
38     sz = 4;
39     to[blank][0] = to[blank][1] = even;
40     len[blank] = suff[blank] = inf;
41     len[even] = 0, suff[even] = odd;
42     len[odd] = -1, suff[odd] = blank;
43     last = 2;
44}
45
46 void build() {
47     init();
48     scanf("%s", s);
49     for (int i = 0; s[i]; ++i)
50         add_char(i);
51}

```

## 16 strings/suffix\_array.cpp

```

1 string s;
2 int n;
3 int sa[maxn], new_sa[maxn], cls[maxn], new_cls[maxn],
4     cnt[maxn], lcp[maxn];
5 int n_cls;
6
7 void build() {
8     n_cls = 256;
9     forn(i, n) {
10         sa[i] = i;
11         cls[i] = s[i];
12     }
13     for (int d = 0; d < n; d = d ? d*2 : 1) {
14
15         forn(i, n) new_sa[i] = (sa[i] - d + n) % n;
16         forn(i, n_cls) cnt[i] = 0;
17         forn(i, n) ++cnt[cls[i]];
18         forn(i, n_cls) cnt[i+1] += cnt[i];
19         for (int i = n-1; i >= 0; --i)
20             sa[--cnt[cls[new_sa[i]]]] = new_sa[i];
21
22         n_cls = 0;
23         forn(i, n) {
24             if (i && (cls[sa[i]] != cls[sa[i-1]] ||
25                 cls[(sa[i] + d) % n] != cls[(sa[i-1] + d) % n])) {
26                 ++n_cls;
27             }
28             new_cls[sa[i]] = n_cls;
29         }
30         ++n_cls;
31         forn(i, n) cls[i] = new_cls[i];
32     }
33
34     // cls is also a inv permutation of sa if a string is not cyclic
35     // (i.e. a position of i-th lexicographical suffix)
36     int val = 0;
37     forn(i, n) {
38         if (val) --val;
39         if (cls[i] == n-1) continue;
40         int j = sa[cls[i] + 1];
41         while (i + val != n && j + val != n && s[i+val] == s[j+val])
42             ++val;
43         lcp[cls[i]] = val;
44     }
45 }
46
47 int main() {
48     cin >> s;
49     s += '$';
50     n = s.length();
51     build();
52     forn(i, n) {
53         cout << s.substr(sa[i]) << endl;
54         cout << lcp[i] << endl;
55     }
56 }

```

## 17 strings/ukkonen.cpp

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 #define sz(x) ((int) (x).size())
4 #define forn(i,n) for (int i = 0; i < int(n); ++i)
5 const int inf = int(1e9) + int(1e5);
6
7 string s;
8 const int alpha = 26;
9
10 namespace SuffixTree {
11     struct Node {
12         Node *to[alpha];
13         Node *lnk, *par;
14         int l, r;
15
16         Node(int l, int r): l(l), r(r) {
17             memset(to, 0, sizeof(to));
18             lnk = par = 0;
19         }
20     };
21
22     Node *root, *blank, *cur;
23     int pos;
24
25     void init() {
26         root = new Node(0, 0);
27         blank = new Node(0, 0);
28         forn(i, alpha)
29             blank->to[i] = root;
30         root->lnk = root->par = blank->lnk = blank->par = blank;
31         cur = root;
32         pos = 0;
33     }
34
35     int at(int id) {
36         return s[id];
37     }
38
39     void goDown(int l, int r) {
40         if (l >= r)
41             return;
42         if (pos == cur->r) {
43             int c = at(l);
44             assert(cur->to[c]);
45             cur = cur->to[c];
46             pos = min(cur->r, cur->l + 1);
47             ++l;
48         } else {
49             int delta = min(r - l, cur->r - pos);
50             l += delta;
51             pos += delta;
52         }
53         goDown(l, r);
54     }
55
56     void goUp() {
57         if (pos == cur->r && cur->lnk) {
58             cur = cur->lnk;
59             pos = cur->r;
60             return;
61         }
62         int l = cur->l, r = pos;
63         cur = cur->par->lnk;
64         pos = cur->r;
65         goDown(l, r);
66     }
67
68     void setParent(Node *a, Node *b) {
69         assert(a);
70         a->par = b;
71         if (b)
72             b->to[at(a->l)] = a;
73     }
74
75     void addLeaf(int id) {
76         Node *x = new Node(id, inf);
77         setParent(x, cur);
78     }
79
80     void splitNode() {
81         assert(pos != cur->r);
82         Node *mid = new Node(cur->l, pos);
83         setParent(mid, cur->par);
84         cur->l = pos;
85         setParent(cur, mid);
86         cur = mid;
87     }
88
89     bool canGo(int c) {
90         if (pos == cur->r)
91             return cur->to[c];
92         return at(pos) == c;
93     }
94
95     void fixLink(Node *&bad, Node *newBad) {
96

```

```

96     if (bad)
97         bad->lnk = cur;
98     bad = newBad;
99 }
100
101 void addCharOnPos(int id) {
102     Node *bad = 0;
103     while (!canGo(at(id))) {
104         if (cur->r != pos) {
105             splitNode();
106             fixLink(bad, cur);
107             bad = cur;
108         } else {
109             fixLink(bad, 0);
110         }
111         addLeaf(id);
112         goUp();
113     }
114     fixLink(bad, 0);
115     goDown(id, id + 1);
116 }
117
118 int cnt(Node *u, int ml) {
119     if (!u)
120         return 0;
121     int res = min(ml, u->r) - u->l;
122     for (i, alpha)
123         res += cnt(u->to[i], ml);
124     return res;
125 }
126
127 void build(int l) {
128     init();
129     for (i, l)
130         addCharOnPos(i);
131 }
132 };
133
134 int main() {
135     cin >> s;
136     SuffixTree::build(s.size());
137 }

```

## 18 structures/convex\_hull\_trick.cpp

```

1 /*
2  WARNING!!!
3  - finds maximum of A*x+B
4  - double check max coords for int/long long overflow
5  - set min x query in put function
6  - add lines with non-descending A coefficient
7 */
8 struct FastHull {
9     int a[maxn];
10    ll b[maxn];
11    ll p[maxn];
12    int c;
13
14    FastHull(): c(0) {}
15
16    ll get(int x) {
17        if (c == 0)
18            return -inf;
19        int pos = upper_bound(p, p + c, x) - p - 1;
20        assert(pos >= 0);
21        return (ll) a[pos] * x + b[pos];
22    }
23
24    ll divideCeil(ll p, ll q) {
25        assert(q > 0);
26        if (p >= 0)
27            return (p + q - 1) / q;
28        return -((-p) / q);
29    }
30
31    void put(int A, ll B) {
32        while (c > 0) {
33            if (a[c - 1] == A && b[c - 1] >= B)
34                return;
35            ll pt = p[c - 1];
36            if (a[c - 1] * pt + b[c - 1] < A * pt + B) {
37                --c;
38                continue;
39            }
40            ll q = A - a[c - 1];
41            ll np = divideCeil(b[c - 1] - B, q);
42            p[c] = np;
43            a[c] = A;
44            b[c] = B;
45            ++c;
46            return;
47        }
48        if (c == 0) {
49            a[c] = A, b[c] = B;
50            p[c] = -1e9; //min x query
51            ++c;
52            return;
53        }
54    }
55 };
56
57
58 struct SlowHull {
59     vector<pair<int, ll>> v;
60
61     void put(int a, ll b) {
62         v.emplace_back(a, b);
63     }
64
65     ll get(ll x) {
66         ll best = -inf;
67         for (auto p: v)
68             best = max(best, p.first * x + p.second);
69         return best;
70     }
71 };
72
73 int main() {
74     FastHull hull1;
75     SlowHull hull2;
76     vector<int> as;
77     for (ii, 10000)
78         as.push_back(rand() % int(1e8));
79     sort(as.begin(), as.end());
80     for (ii, 10000) {
81         int b = rand() % int(1e8);
82         hull1.put(as[ii], b);
83         hull2.put(as[ii], b);
84         int x = rand() % int(2e8 + 1) - int(1e8);
85         assert(hull1.get(x) == hull2.get(x));
86     }
87 }

```

## 19 structures/heavy\_light.cpp

```

1 const int maxn = 100500;
2 const int maxd = 17;
3
4 vector<int> g[maxn];
5
6 struct Tree {
7     vector<int> t;
8     int base;
9
10    Tree(): base(0) {
11    }
12
13    Tree(int n) {
14        base = 1;
15        while (base < n)
16            base *= 2;
17        t = vector<int>(base * 2, 0);
18    }
19
20    void put(int v, int delta) {
21        assert(v < base);
22        v += base;
23        t[v] += delta;
24        while (v > 1) {
25            v /= 2;
26            t[v] = max(t[v * 2], t[v * 2 + 1]);
27        }
28    }
29
30    //Careful here: cr = 2 * maxn
31    int get(int l, int r, int v = 1, int cl = 0, int cr = 2 * maxn) {
32        cr = min(cr, base);
33        if (l <= cl && cr <= r)
34            return t[v];
35        if (r <= cl || cr <= 1)
36            return 0;
37        int cc = (cl + cr) / 2;
38        return max(get(l, r, v * 2, cl, cc), get(l, r, v * 2 + 1, cc,
39    ↪ cr));
40    }
41
42 namespace HLD {
43     int h[maxn];
44     int timer;
45     int in[maxn], out[maxn], cnt[maxn];
46     int p[maxd][maxn];
47     int vroot[maxn];
48     int vpos[maxn];
49     int ROOT;
50     Tree tree[maxn];
51
52     void dfs1(int u, int prev) {
53         p[0][u] = prev;
54         in[u] = timer++;
55         cnt[u] = 1;
56         for (int v: g[u]) {
57             if (v == prev)
58                 continue;
59             h[v] = h[u] + 1;
60             dfs1(v, u);
61             cnt[u] += cnt[v];
62         }
63         out[u] = timer;
64     }
65
66     int dfs2(int u, int prev) {
67         int to = -1;
68         for (int v: g[u]) {
69             if (v == prev)
70                 continue;
71             if (to == -1 || cnt[v] > cnt[to])
72                 to = v;
73         }
74         int len = 1;
75         for (int v: g[u]) {
76             if (v == prev)
77                 continue;
78             if (to == v) {
79                 vpos[v] = vpos[u] + 1;
80                 vroot[v] = vroot[u];
81                 len += dfs2(v, u);
82             }
83             else {
84                 vroot[v] = v;
85                 vpos[v] = 0;
86                 dfs2(v, u);
87             }
88         }
89         if (vroot[u] == u)
90             tree[u] = Tree(len);
91         return len;
92     }
93
94     void init(int n) {

```

```

95         timer = 0;
96         h[ROOT] = 0;
97         dfs1(ROOT, ROOT);
98         forn (d, maxd - 1)
99             forn (i, n)
100                 p[d + 1][i] = p[d][p[d][i]];
101         vroot[ROOT] = ROOT;
102         vpos[ROOT] = 0;
103         dfs2(ROOT, ROOT);
104         //WARNING: init all trees
105     }
106
107     bool isPrev(int u, int v) {
108         return in[u] <= in[v] && out[v] <= out[u];
109     }
110
111     int lca(int u, int v) {
112         for (int d = maxd - 1; d >= 0; --d)
113             if (!isPrev(p[d][u], v))
114                 u = p[d][u];
115         if (!isPrev(u, v))
116             u = p[0][u];
117         return u;
118     }
119
120     //for each v: h[v] >= toh
121     int getv(int u, int toh) {
122         int res = 0;
123         while (h[u] >= toh) {
124             int rt = vroot[u];
125             int l = max(0, toh - h[rt]), r = vpos[u] + 1;
126             res = max(res, tree[rt].get(l, r));
127             if (rt == ROOT)
128                 break;
129             u = p[0][rt];
130         }
131         return res;
132     }
133
134     int get(int u, int v) {
135         int w = lca(u, v);
136         return max(getv(u, h[w]), getv(v, h[w] + 1));
137     }
138
139     void put(int u, int val) {
140         int rt = vroot[u];
141         int pos = vpos[u];
142         tree[rt].put(pos, val);
143     }
144 };

```

## 20 structures/ordered\_set.cpp

```

1#include <ext/pb_ds/assoc_container.hpp>
2#include <ext/pb_ds/tree_policy.hpp>
3
4typedef __gnu_pbds::tree<int, __gnu_pbds::null_type, std::less<int>,
5    __gnu_pbds::rb_tree_tag,
6    __gnu_pbds::tree_order_statistics_node_update> oset;
7
8#include <iostream>
9
10int main() {
11    oset X;
12    X.insert(1);
13    X.insert(2);
14    X.insert(4);
15    X.insert(8);
16    X.insert(16);
17
18    std::cout << *X.find_by_order(1) << std::endl; // 2
19    std::cout << *X.find_by_order(2) << std::endl; // 4
20    std::cout << *X.find_by_order(4) << std::endl; // 16
21    std::cout << std::boolalpha << (end(X)==X.find_by_order(6)) <<
22    std::endl; // true
23
24    std::cout << X.order_of_key(-5) << std::endl; // 0
25    std::cout << X.order_of_key(1) << std::endl; // 0
26    std::cout << X.order_of_key(3) << std::endl; // 2
27    std::cout << X.order_of_key(4) << std::endl; // 2
28    std::cout << X.order_of_key(400) << std::endl; // 5
29}

```

## 21 structures/splay.cpp

```

1#include <bits/stdc++.h>
2using namespace std;
3#define form(i, n) for (int i = 0; i < (int)(n); ++i)
4
5const int maxn = 100500;
6
7struct node;
8void updson(node* p, node* v, node* was);
9
10struct node {
11    int val;
12    node *l, *r, *p;
13    node() {}
14    node(int val) : val(val), l(r=NULL) {}
15
16    bool isRoot() const { return !p; }
17    bool isRight() const { return p && p->r == this; }
18    bool isLeft() const { return p && p->l == this; }
19    void setLeft(node* t) {
20        if (t) t->p = this;
21        l = t;
22    }
23    void setRight(node* t) {
24        if (t) t->p = this;
25        r = t;
26    }
27};
28
29void updson(node *p, node *v, node *was) {
30    if (p) {
31        if (p->l == was) p->l = v;
32        else p->r = v;
33    }
34    if (v) v->p = p;
35}
36
37void rightRotate(node *v) {
38    assert(v && v->l);
39    node *u = v->l;
40    node *p = v->p;
41    v->setLeft(u->r);
42    u->setRight(v);
43    updson(p, u, v);
44}
45
46void leftRotate(node *v) {
47    assert(v && v->r);
48    node *u = v->r;
49    node *p = v->p;
50    v->setRight(u->l);
51    u->setLeft(v);
52    updson(p, u, v);
53}
54
55void splay(node *v) {
56    while (v->p) {
57        if (!v->p->p) {
58            if (v->isLeft()) rightRotate(v->p);
59            else leftRotate(v->p);
60        } else if (v->isLeft() && v->p->isLeft()) {
61            rightRotate(v->p->p);
62            rightRotate(v->p);
63        } else if (v->isRight() && v->p->isRight()) {
64            leftRotate(v->p->p);
65            leftRotate(v->p);
66        } else if (v->isLeft()) {
67            rightRotate(v->p);
68            leftRotate(v->p);
69        } else {
70            leftRotate(v->p);
71            rightRotate(v->p);
72        }
73    }
74    v->p = NULL;
75}
76
77node *insert(node *t, node *n) {
78    if (!t) return n;
79    int x = n->val;
80    while (true) {
81        if (x < t->val) {
82            if (t->l) {
83                t = t->l;
84            } else {
85                t->setLeft(n);
86                t = t->l;
87                break;
88            }
89        } else {
90            if (t->r) {
91                t = t->r;
92            } else {
93                t->setRight(n);
94                t = t->r;
95                break;
96            }
97        }
98    }
99}

```

```

96     }
97 }
98 }
99 splay(t);
100 return t;
101}
102
103node *insert(node *t, int x) {
104     return insert(t, new node(x));
105}
106
107int main() {
108     node *t = NULL;
109     forn(i, 1000000) {
110         int x = rand();
111         t = insert(t, x);
112     }
113     return 0;
114}

```

## 22 structures/treap.cpp

```

1 struct node {
2     int x, y;
3     node *l, *r;
4     node(int x) : x(x), y(rand()), l(r=NULL) {}
5 };
6
7 void split(node *t, node *&l, node *&r, int x) {
8     if (!t) return (void)(l=r=NULL);
9     if (x <= t->x) {
10         split(t->l, l, t->l, x), r = t;
11     } else {
12         split(t->r, t->r, r, x), l = t;
13     }
14 }
15
16 node *merge(node *l, node *r) {
17     if (!l) return r;
18     if (!r) return l;
19     if (l->y > r->y) {
20         l->r = merge(l->r, r);
21         return l;
22     } else {
23         r->l = merge(l, r->l);
24         return r;
25     }
26 }
27
28 node *insert(node *t, node *n) {
29     node *l, *r;
30     split(t, l, r, n->x);
31     return merge(l, merge(n, r));
32 }
33
34 node *insert(node *t, int x) {
35     return insert(t, new node(x));
36 }
37
38 node *fast_insert(node *t, node *n) {
39     if (!t) return n;
40     node *root = t;
41     while (true) {
42         if (n->x < t->x) {
43             if (!t->l || t->l->y < n->y) {
44                 split(t->l, n->l, n->r, n->x), t->l = n;
45                 break;
46             } else {
47                 t = t->l;
48             }
49         } else {
50             if (!t->r || t->r->y < n->y) {
51                 split(t->r, n->l, n->r, n->x), t->r = n;
52                 break;
53             } else {
54                 t = t->r;
55             }
56         }
57     }
58     return root;
59 }
60
61 node *fast_insert(node *t, int x) {
62     return fast_insert(t, new node(x));
63 }
64
65 int main() {
66     node *t = NULL;
67     forn(i, 1000000) {
68         int x = rand();
69         t = fast_insert(t, x);
70     }
71 }

```