01/1 - Build The Sum

```cpp
#include <iostream>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int m;
        cin >> m;

        // Read sequence and build sum.
        double s = 0;
        for (int i = 0; i < m; ++i)
        {
            double f;
            cin >> f;
            s += f;
        }

        cout << s << endl;
    }
}
```

01/2 - Even Pairs

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read sequence.
        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];

        // Perform a scanline on the sequence.
        int e = 0, o = 0, E = 0;
        for (int i = 0; i < n; ++i)
        {
            if (!xs[i]) ++e;
            else swap(e, o), ++o;
            E += e;
        }

        cout << E << endl;
    }
}
```

01/3 - Dominoes

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read heights.
        vector<int> hs(n);
        for (int i = 0; i < n; ++i)
            cin >> hs[i];

        // Perform a scanline.
        int c = 0, d = 1;
        for (int i = 0; i < n; ++i)
        {
            c = max(hs[i] - 1, c - 1);
            if (c == 0)
                break;
            ++d;
        }

        cout << (d < n ? d : n) << endl;
    }
}
```

01/4 - Even Matrices

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read matrix.
        vector<vector<int>> M(n, vector<int>(n));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                cin >> M[i][j];

        // Apply even-sequences algorithm line-wise.
        int E = 0;
        vector<vector<int>> S(n, vector<int>(n));
        for (int k = 0; k < n; ++k)
        {
            // Consider sub-matrices of height k + 1.
            for (int i = 0; i < n - k; ++i)
                for (int j = 0; j < n; ++j)
                    S[i][j] ^= M[i + k][j];

            // Perform a scanline on each row of S.
            for (int i = 0; i < n - k; ++i)
            {
                int e = 0, o = 0;
                for (int j = 0; j < n; ++j)
                {
                    if (!S[i][j]) ++e;
                    else swap(e, o), ++o;
                    E += e;
                }
            }
        }

        cout << E << endl;
    }
}
```

01/5 - False Coin

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, k;
        cin >> n >> k;

        vector<bool> candidate(n + 1, true), heavier(n + 1), lighter(n + 1);

        for (int i = 0; i < k; ++i)
        {
            int Pi;
            cin >> Pi;

            vector<int> left(Pi), right(Pi);
            for (int j = 0; j < Pi; ++j)
                cin >> left[j];
            for (int j = 0; j < Pi; ++j)
                cin >> right[j];

            char outcome;
            cin >> outcome;

            if (outcome == '=')
                for (int j = 0; j < Pi; ++j)
                    candidate[left[j]] = false, candidate[right[j]] = false;

            else
            {
                vector<bool> bad(n + 1);

                for (int j = 0; j < Pi; ++j)
                {
                    bad[left[j]] = true, bad[right[j]] = true;

                    if (outcome == '<')
                        lighter[left[j]] = true, heavier[right[j]] = true;
                    else // outcome == '>'
                        heavier[left[j]] = true, lighter[right[j]] = true;

                    if (lighter[left[j]] && heavier[left[j]])
                        candidate[left[j]] = false;
                    if (lighter[right[j]] && heavier[right[j]])
                        candidate[right[j]] = false;
                }

                for (int k = 1; k <= n; ++k)
                    if (!bad[k])
                        candidate[k] = false;
            }
        }

        int cnt = 0, sol;
        for (int k = 1; k <= n; ++k)
            if (candidate[k])
                ++cnt, sol = k;
        cout << (cnt == 1 ? sol : 0) << endl;
    }
}
```

01/6 - Deck of Cards

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, k;
        cin >> n >> k;

        vector<int> cs(n);
        for (int i = 0; i < n; ++i)
            cin >> cs[i];

        int i = 0, j = 0, s = cs[0], i_ = 0, j_ = 0, v_ = abs(s - k);
        while (i < n)
        {
            s += s < k && j < n ? cs[++j] : -cs[i++];
            if (abs(s - k) < v_)
                i_ = i, j_ = j, v_ = abs(s - k);
        }
        cout << i_ << " " << j_ << endl;
    }
}
```

02/1 - Search Snippets

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <limits>

using namespace std;

int main()
{
    ios_base::sync_with_stdio(false);

    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read number of occurrences of each word.
        vector<int> ms(n);
        for (int w = 0; w < n; ++w)
            cin >> ms[w];

        // Create ordered list of position/word pairs.
        vector<pair<int, int>> pws;
        for (int w = 0; w < n; ++w)
            for (int i = 0; i < ms[w]; ++i)
                pws.push_back(make_pair([](){ int p; cin >> p; return p; }(), w));
        sort(pws.begin(), pws.end());

        // Position of last occurrence of each word.
        vector<int> ls(n);
        for (int w = 0; w < n; ++w)
            ls[w] = -1;

        // Sliding window on positions-words sequence.
        int a = 0, b = 0, c = 1, l = numeric_limits<int>::max(), l_, N = pws.size();
        vector<int> cs(n);
        cs[pws[0].second] = 1;
        while (a < N)
        {
            // If this interval contains all the words, check whether optimal.
            if (c == n && (l_ = pws[b].first - pws[a].first + 1) < l)
                l = l_;

            // Increase  lower- or upper-bound depending on the number of different words.
            if (c < n && b < N - 1)
            {
                if (cs[pws[++b].second]++ == 0)
                    ++c;
            }
            else
            {
                if (cs[pws[a++].second]-- == 1)
                    --c;
            }
        }

        cout << l << endl;
    }
}
```

02/2 - Boats

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read and sort lengths and positions.
        vector<pair<int, int>> bs(n);
        for (int i = 0; i < n; ++i)
            cin >> bs[i].second >> bs[i].first;
        sort(bs.begin(), bs.end());

        // Choose greedily whether to take each boat.
        int c = 0, pos = -1, pos_;
        for (auto b : bs)
        {
            // First boat.
            if (pos == -1)
                ++c, pos_ = b.first - b.second, pos = b.first;

            else
            {
                // If there is place, take this boat.
                if (b.first >= pos)
                    ++c, pos_ = pos, pos = max(pos_ + b.second, b.first);

                // If taking this boat instead of the last decreases pos, take it.
                else if (b.second < pos - pos_ && max(pos_ + b.second, b.first) < pos)
                    pos = max(pos_ + b.second, b.first);
            }
        }

        cout << c << endl;
    }
}
```

02/3 - Moving Books

```cpp
#include <iostream>
#include <algorithm>
#include <set>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    BEGIN:
    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read strengths.
        int s_max = 0;
        vector<int> ss(n);
        for (int i = 0; i < n; ++i)
        {
            cin >> ss[i];
            if (ss[i] > s_max)
                s_max = ss[i];
        }

        // Read weights.
        multiset<int, greater<int>> ws;
        for (int i = 0; i < m; ++i)
        {
            int w;
            cin >> w;
            if (w > s_max)
            {
                cout << "impossible" << endl;
                goto BEGIN;
            }
            ws.insert(w);
        }

        // Sort strengths.
        sort(ss.begin(), ss.end(), greater<int>());

        // Greedy simulation.
        int r = 0;
        while (!ws.empty())
        {
            for (int i = 0; i < n; ++i)
            {
                auto b = ws.lower_bound(ss[i]);
                if (b != ws.end()) ws.erase(b);
                else break;
            }
            ++r;
        }
        cout << 3 * r - 1 << endl;
    }
}
```

02/4 - Evolution

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <stack>
#include <algorithm>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, q;
        cin >> n >> q;

        // Read species and ages.
        unordered_map<string, int> idx;
        vector<string> species(n);
        vector<int> ages(n);
        for (int i = 0; i < n; ++i)
        {
            cin >> species[i] >> ages[i];
            idx[species[i]] = i;
        }

        // Read offsprings and build graph.
        int root = max_element(ages.begin(), ages.end()) - ages.begin();
        vector<vector<int>> adj(n);
        for (int i = 0; i < n - 1; ++i)
        {
            string s, p;
            cin >> s >> p;
            adj[idx[p]].push_back(idx[s]);
        }

        // Read queries.
        vector<vector<pair<int, int>>> queries(n);
        for (int i = 0; i < q; ++i)
        {
            string s;
            int b;
            cin >> s >> b;
            queries[idx[s]].push_back(make_pair(b, i));
        }

        // Perform modified DFS iteratively on tree.
        vector<int> path, result(q);
        stack<int> s;
        s.push(root);
        while (!s.empty())
        {
            int v = s.top();
            s.pop();
            path.push_back(v);

            // Perform binary search on children.
            for (auto& query : queries[v])
            {
                int l = 0, r = path.size() - 1;
                while (l != r)
                {
                    int m = (l + r) / 2;
                    if (ages[path[m]] > query.first) l = m + 1;
                    else r = m;
                }
                result[query.second] = path[l];
            }

            for (int u : adj[v])
                s.push(u);

            if (adj[v].size() == 0)
                path.pop_back();
        }

        for (int i = 0; i < q; ++i)
            cout << species[result[i]] << " ";
        cout << endl;
    }
}
```

02/5 - Octopussy

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read times.
        vector<int> ts(n);
        for (int i = 0; i < n; ++i)
            cin >> ts[i];

        // Compute maximum deactivation time of each bomb greedily.
        for (int i = 1; i < n; ++i)
        {
            int d = !(i % 2) && ts[(i - 1) / 2] - ts[i - 1] == 1 ? 2 : 1;
            ts[i] = min(ts[i], ts[(i - 1) / 2] - d);
        }

        // Sort maximum deactivation times.
        sort(ts.begin(), ts.end());

        // Check that the sequence of maximum deactivation times is <= i.
        bool success = true;
        for (int i = 0; i < n && success; ++i)
            if (ts[i] <= i)
                success = false;

        cout << (success ? "yes" : "no") << endl;
    }
}
```

03/1 - Hit

```cpp
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef K::Ray_2 R;
typedef K::Segment_2 S;

int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
            break;

        // Read phileas ray.
        R r;
        cin >> r;

        // Read obstacles segments.
        vector<S> os(n);
        for (int i = 0; i < n; ++i)
            cin >> os[i];

        // Check whether ray hits a segment.
        bool hit = false;
        for (int i = 0; i < n && !hit; ++i)
            hit |= do_intersect(r, os[i]);

        cout << (hit ? "yes" : "no") << endl;
    }
}
```

03/2 - First Hit

```
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>

using namespace std;
using namespace boost;
using namespace CGAL;

typedef Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 P;
typedef K::Ray_2 R;
typedef K::Segment_2 S;

double f2d(const K::FT& x)
{
    double a = floor(to_double(x));
    while (a > x) a -= 1;
    while (a + 1 <= x) a += 1;
    return a;
}

int main()
{
    ios_base::sync_with_stdio(false);

    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
            break;

        // Read phileas ray.
        R r;
        cin >> r;

        // Read obstacle segments and randomize their order.
        vector<S> os(n);
        for (int j = 0; j < n; ++j)
        {
            long r, s, t, u;
            cin >> r >> s >> t >> u;
            os[j] = S(P(r, s), P(t, u));
        }
        random_shuffle(os.begin(), os.end());

        // Find closest intersection point p with an obstacle segment on-line.
        P p;
        S s;
        bool hit = false;
        for (auto o : os)
        {
            // Consider a ray.
            if (!hit && do_intersect(r, o))
            {
                P p_;
                auto i = intersection(r, o);

                // Ray and obstacle nonparallel, intersection is a point.
                if (const P* q = get<P>(&*i))
                    p = *q;

                // Ray and obstacle parallel, intersection is a segment;
                // Find which endpoint is closer to the ray's source.
                else if (const S* s = get<S>(&*i))
                    p = squared_distance(r.source(), s->source()) <
                            squared_distance(r.source(), s->target()) ?
                            s->source() : s->target();

                // Reduce the ray to a segment.
                s = S(r.source(), p);
                hit = true;
            }

            // Consider a segment.
            else if (hit && do_intersect(s, o))
            {
                P p_;
                auto i = intersection(r, o);

                // Ray and obstacle nonparallel, intersection is a point.
                if (const P* q = get<P>(&*i))
                    p_ = *q;

                // Ray and obstacle parallel, intersection is a segment;
                // Find which endpoint is closer to the ray's source.
                else if (const S* s = get<S>(&*i))
```

```
                    p_ = squared_distance(r.source(), s->source()) <
                            squared_distance(r.source(), s->target()) ?
                            s->source() : s->target();

                // Update closest intersection.
                if (squared_distance(r.source(), p_) < squared_distance(r.source(), p))
                    p = p_;
            }
        }

        if (hit) cout << (long)f2d(p.x()) << " " << (long)f2d(p.y()) << endl;
        else cout << "no" << endl;
    }
}
```

03/3 - Hiking Maps

```cpp
#include <iostream>
#include <vector>
#include <limits>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 P;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int m, n;
        cin >> m >> n;

        // Read path points.
        vector<P> ps(m);
        for (int i = 0; i < m; ++i)
            cin >> ps[i];

        // Read triangles points and adjust directions of edge points.
        vector<vector<P>> ts(n);
        for (int i = 0; i < n; ++i)
        {
            ts[i] = vector<P>(6);
            for (int j = 0; j < 6; ++j)
                cin >> ts[i][j];

            // Make sure points on edges are clock-wise oriented.
            for (int j = 0; j < 3; ++j)
                for (int k = 2; k <= 5; ++k)
                    if (right_turn(ts[i][2 * j], ts[i][2 * j + 1], ts[i][(2 * j + k) % 6]))
                        swap(ts[i][2 * j], ts[i][2 * j + 1]);
        }

        // Find all segments covered by each triangle.
        vector<vector<int>> cs(n, vector<int>());
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < m - 1; ++j)
            {
                bool c = true;
                for (int k = 0; k < 3; ++k)
                    c &= !right_turn(ts[i][2 * k], ts[i][2 * k + 1], ps[j]) &&
                         !right_turn(ts[i][2 * k], ts[i][2 * k + 1], ps[j + 1]);
                if (c)
                    cs[i].push_back(j);
            }
        }

        // Scanline to find the cheapest covering interval.
        int b, e, min = -1, d = numeric_limits<int>::max();
        vector<int> ls(m - 1, -1);
        for (int i = 0; i < n; ++i)
        {
            for (auto s : cs[i])
            {
                // Update last triangle covering this segment.
                ls[s] = i;

                // Update interval.
                if (s == min || min == -1)
                {
                    auto idx = min_element(ls.begin(), ls.end());
                    min = distance(ls.begin(), idx);
                    b = *idx;
                }
                e = i;

                // Update interval length if better.
                int d_ = e - b + 1;
                if (b != -1 && d_ < d)
                    d = d_;
            }
        }

        cout << d << endl;
    }
}
```

03/4 - Antenna

```cpp
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef Min_circle_2_traits_2<K> T;
typedef Min_circle_2<T> MC;
typedef K::Point_2 P;

int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
            break;

        // Read citizens coordinates.
        vector<P> cs(n);
        for (int i = 0; i < n; ++i)
        {
            long x, y;
            cin >> x >> y;
            cs[i] = P(x, y);
        }

        // Compute minimum covering circle.
        MC mc(&cs[0], &cs[n], true);
        cout << ceil(sqrt(mc.circle().squared_radius())) << endl;
    }
}
```

03/5 - Attack of the Clones

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <set>

using namespace std;

// ABRT.
int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Avoid ABRT.
        if (n > 1000)
            return 0;

        // Read and sort segments covered by each Jedi.
        vector<pair<int, int>> js(n);
        for (int i = 0; i < n; ++i)
            cin >> js[i].second >> js[i].first;

        // Find segment covered by the least Jedis.
        vector<int> qs(m);
        for (auto j : js)
            for (int i = j.second - 1; i < j.first; ++i)
                ++qs[i];
        int k = distance(qs.begin(), min_element(qs.begin(), qs.end()));// + 1;
        //cout<<k<<endl;
        //int k = 0;

        // Fix and sort the Jedi intervals around segment k and find Jedis covering segment k.
        set<pair<int, int>> cs;
        cs.insert(make_pair(-1, -1));
        for (auto& j : js)
        {
            j.first = (m + ((j.first - k) % m)) % m;
            j.second = (m + ((j.second - k) % m)) % m;
            //if ((j.second <= j.first && j.second <= k && k <= j.first) ||
            //        (j.second > j.first && (j.second <= k || k <= j.first)))
            if (j.second > j.first)
                cs.insert(j);
        }
        sort(js.begin(), js.end());
        //for(auto c:cs)cout<<c.first<<","<<c.second<<"; ";cout<<endl;
        //for(auto j:js)cout<<j.second<<","<<j.first<<"; ";cout<<endl;

        // Try to use each one of the Jedis covering segment k separately, or none.
        int min, max, size, max_size = 0;
        for (auto c : cs)
        {
            // Consider (or not) the covering segment.
            if (c.first == -1)
                size = 0, min = 0, max = m;
            else
                size = 1, min = c.first, max = c.second;
            //cout<<min<<" "<<max<<endl;

            // Greedily solve earliest finish time jobs scheduling.
            for (auto j : js)
                if (!cs.count(j) && j.second > min && j.first < max)//{cout<<j.second<<","<<j.first<<"; ";
                    ++size, min = j.first;//}cout<<endl;
            //cout<<size<<endl;
            // Update best achievable size.
            if (size > max_size)
                max_size = size;
        }

        cout << max_size << endl;
    }
}
```

04/1 - First Steps with BGL

```cpp
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/prim_minimum_spanning_tree.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int>> G;
typedef property_map<G, edge_weight_t>::type WM;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read graph.
        G g(n);
        WM wm = get(edge_weight, g);
        for (int i = 0; i < m; ++i)
        {
            int u, v;
            cin >> u >> v;
            cin >> wm[add_edge(u, v, g).first];
        }

        // Compute weight of minimum spanning tree.
        int w = 0;
        vector<int> pred(n);
        prim_minimum_spanning_tree(g, &pred[0]);
        for (int v = 0; v < n; ++v)
        {
            auto e = edge(v, pred[v], g);
            if (e.second)
                w += wm[e.first];
        }

        // Compute distance from 0 to furthest vertex.
        int d = 0;
        vector<int> dist(n);
        dijkstra_shortest_paths(g, 0, distance_map(&dist[0]));
        for (int v = 0; v < n; ++v)
            d = max(d, dist[v]);

        cout << w << " " << d << endl;
    }
}
```

04/2 - Ant Challenge

```cpp
#include <iostream>
#include <vector>
#include <limits>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/prim_minimum_spanning_tree.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int>> G;
typedef property_map<G, edge_weight_t>::type WM;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, s, a, b;
        cin >> n >> m >> s >> a >> b;

        // Read each species graph.
        vector<G> pns(s, G(n));
        vector<WM> wm(s);
        for (int i = 0; i < s; ++i)
            wm[i] = get(edge_weight, pns[i]);
        for (int j = 0; j < m; ++j)
        {
            int t1, t2;
            cin >> t1 >> t2;
            for (int i = 0; i < s; ++i)
                cin >> wm[i][add_edge(t1, t2, pns[i]).first];
        }

        // Read hives (useless, MSTs unique).
        vector<int> hs(n);
        for (int i = 0; i < s; ++i)
            cin >> hs[i];

        // Find each species private network.
        vector<vector<int>> pred(s, vector<int>(n));
        for (int i = 0; i < s; ++i)
            prim_minimum_spanning_tree(pns[i], &pred[i][0]);

        // Create joint minimum weight private networks graph.
        G g(n);
        WM wm2 = get(edge_weight, g);
        for (int i = 0; i < s; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                auto e = edge(j, pred[i][j], pns[i]);
                if (e.second)
                {
                    auto e2 = edge(j, pred[i][j], g);
                    if (!e2.second)
                        wm2[add_edge(j, pred[i][j], g).first] = wm[i][e.first];
                    else if (wm[i][e.first] < wm2[e2.first])
                        wm2[e2.first] = wm[i][e.first];
                }
            }
        }

        // Find shortest a-b path on super-graph.
        vector<int> dist(n);
        dijkstra_shortest_paths(g, a, distance_map(&dist[0]));

        cout << dist[b] << endl;
    }
}
```

04/3 - Important Bridges

```
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/biconnected_components.hpp>

using namespace std;
using namespace boost;

struct EC
{
    enum { num = 555 };
    typedef edge_property_tag kind;
}
edge_component;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property <EC, size_t>> G;
typedef graph_traits<G>::edge_descriptor E;
typedef graph_traits<G>::edge_iterator EI;
typedef property_map<G, EC>::type C;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read graph.
        G g(n);
        for (int i = 0; i < m; ++i)
        {
            int u, v;
            cin >> u >> v;
            add_edge(u, v, g);
        }

        // Compute connected components.
        C c = get(edge_component, g);
        size_t nc = biconnected_components(g, c);

        // Find size-2 connected components (edges) and sort them.
        vector<vector<E>> cnt(nc);
        for (auto e = edges(g); e.first != e.second; ++e.first)
            cnt[c[*e.first]].push_back(*e.first);
        vector<pair<int, int>> es;
        for (auto i : cnt)
            if (i.size() == 1)
                es.push_back(make_pair(
                        min(source(i[0], g), target(i[0], g)),
                        max(source(i[0], g), target(i[0], g))));
        sort(es.begin(), es.end());

        cout << es.size() << endl;
        for (auto e : es)
            cout << e.first << " " << e.second << endl;
    }
}
```

04/4 - Buddy Selection

```cpp
#include <iostream>
#include <vector>
#include <limits>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> G;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, c, f;
        cin >> n >> c >> f;

        // Read and sort characteristics of each student.
        vector<vector<string>> cs(n, vector<string>(c));
        for (int i = 0; i < n; ++i)
        {
            for (int k = 0; k < c; ++k)
                cin >> cs[i][k];
            sort(cs[i].begin(), cs[i].end());
        }

        // Create graph connecting students who share >f characteristics.
        G g(n);
        for (int i = 0; i < n; ++i)
        {
            for (int j = i; j < n; ++j)
            {
                // Search common characteristics in linear time.
                int cnt = 0, l = 0;
                for (int k = 0; k < c; ++k)
                {
                    while (l < c && cs[i][k] > cs[j][l]) ++l;
                    if (l < c && cs[i][k] == cs[j][l]) ++cnt;
                }

                if (cnt > f)
                    add_edge(i, j, g);
            }
        }

        // If the matching is perfect, the solution was not optimal.
        vector<int> mate(n);
        edmonds_maximum_cardinality_matching(g, &mate[0]);
        cout << (matching_size(g, &mate[0]) == (size_t)n / 2 ? "not optimal" : "optimal") << endl;
    }
}
```

04/5 - TheeV

```cpp
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_exact_constructions_kernel K;
typedef Min_circle_2_traits_2<K> T;
typedef Min_circle_2<T> MC;
typedef K::Point_2 P;

double c2d(const K::FT& x)
{
    double a = floor(to_double(x));
    while (a - 1 >= x) --a;
    while (a < x) ++a;
    return a;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read cities coordinates.
        vector<P> cs(n);
        for (int i = 0; i < n; ++i)
            cin >> cs[i];

        // Compute and sort distances from old transmitter to all other cities.
        vector<pair<K::FT, int>> ds(n);
        for (int i = 0; i < n; ++i)
            ds[i] = make_pair(squared_distance(cs[0], cs[i]), i);
        sort(ds.begin(), ds.end());

        // Perform binary search on distances.
        K::FT best = numeric_limits<long>::max();
        int l = 0, r = n - 1;
        while (l != r)
        {
            // Compute minimum enclosing circle on cities not covered by old transmitter.
            int m = (l + r) / 2;
            vector<P> os(n - m - 1);
            for (int i = m + 1; i < n; ++i)
                os[i - m - 1] = cs[ds[i].second];
            MC mc(&os[0], &os[n - m - 1], true);

            // Take best circle.
            best = min(best, max(mc.circle().squared_radius(), ds[m].first));
            if (mc.circle().squared_radius() > ds[m].first) l = m + 1;
            else r = m;
        }

        cout << (long)c2d(best) << endl;
    }
}
```

05/1 - Burning Coins

```cpp
#include <iostream>
#include <vector>

using namespace std;

int solve(int i, int j, bool t, const vector<int>& vs, vector<vector<int>>& m)
{
    // Base case.
    if (i == j)
        return t ? vs[i] : 0;

    // Use memoization.
    if (m[i][j] != -1)
        return m[i][j];

    // Use recursion.
    return m[i][j] = t ? max(vs[i] + solve(i + 1, j, !t, vs, m), vs[j] + solve(i, j - 1, !t, vs, m))
                       : min(solve(i + 1, j, !t, vs, m), solve(i, j - 1, !t, vs, m));
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read coins values.
        vector<int> vs(n);
        for (int i = 0; i < n; ++i)
            cin >> vs[i];

        // Solve recursion using dynamic programming.
        vector<vector<int>> m(n, vector<int>(n, -1));
        cout << solve(0, n - 1, true, vs, m) << endl;
    }
}
```

05/2 - Light Pattern

```cpp
#include <iostream>
#include <vector>
#include <bitset>
#include <cmath>

using namespace std;

int solve(int i, int k, bitset<16> p, bitset<16> r, const vector<bitset<16>>& bs, vector<vector<int>>& m)
{
    // Base case.
    if (i == -1)
        return 0;

    // Use memoization.
    if (m[r[0]][i] != -1)
        return m[r[0]][i];

    // Use recursion.
    int d = ((bs[i] ^ r ^ p) & bitset<16>(pow(2, k) - 1)).count();
    return m[r[0]][i] = min(d + solve(i - 1, k, p, r, bs, m), k - d + 1 + solve(i - 1, k, p, ~r, bs, m));
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, k, x;
        cin >> n >> k >> x;

        // Read bulbs initial states.
        vector<bitset<16>> bs(n / k);
        for (int i = 0; i < n / k; ++i)
            for (int j = 0; j < k; ++j)
                bs[i][k - j - 1] = [](){ bool b; cin >> b; return b; }();

        // Solve recursion using dynamic programming.
        vector<vector<int>> m(2, vector<int>(n / k, -1));
        cout << solve(n / k - 1, k, bitset<16>(x), bitset<16>(), bs, m) << endl;
    }
}
```

05/3 - Light at the Museum

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <bitset>
#include <map>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read target brightnesses.
        vector<int> bs(m);
        for (int j = 0; j < m; ++j)
            cin >> bs[j];

        // Read light-switch connections.
        vector<vector<int>> on(n, vector<int>(m));
        vector<vector<int>> off(n, vector<int>(m));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
                cin >> on[i][j] >> off[i][j];

        // Bruteforce all combinations on half of the switches.
        map<vector<int>, int> list;
        for (int s = 0; s < 1 << n / 2; ++s)
        {
            vector<int> ss(m);
            for (int j = 0; j < m; ++j)
                for (int i = 0; i < n / 2; ++i)
                    ss[j] += s & 1 << i ? off[i][j] : on[i][j];
            int c = bitset<30>(s).count();
            if (!list.count(ss) || (list.count(ss) && c < list[ss]))
                list[ss] = c;
        }

        // Bruteforce on other half.
        int best = 31;
        for (int s = 0; s < 1 << (n / 2 + n % 2); ++s)
        {
            vector<int> ss(bs);
            for (int j = 0; j < m; ++j)
                for (int i = 0; i < n / 2 + n % 2; ++i)
                    ss[j] -= s & 1 << i ? off[n / 2 + i][j] : on[n / 2 + i][j];
            int c = bitset<30>(s).count();
            if (list.count(ss) && list[ss] + c < best)
                best = list[ss] + c;
        }

        cout << (best != 31 ? to_string(best) : "impossible") << endl;
    }
}
```

05/4 - The Great Game

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

int solve(int c, int i, int p, int n, const vector<vector<int>>& ns, vector<vector<int>>& m)
{
    // Base case.
    if (i == n)
        return 0;

    // Use memoization.
    if (m[p][i] != -1)
        return m[p][i];

    // Use recursion.
    int min = numeric_limits<int>::max();
    int max = numeric_limits<int>::min();
    for (auto j : ns[i])
    {
        int s = solve(c, j, !p, n, ns, m);
        min = s < min ? s : min;
        max = s > max ? s : max;
    }

    return m[p][i] = (c == p ? min : max) + 1;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, r, b;
        cin >> n >> m >> r >> b;

        // Read transitions.
        vector<vector<int>> ns(n);
        for (int k = 0; k < m; ++k)
        {
            int i, j;
            cin >> i >> j;
            ns[i].push_back(j);
        }

        // Find optimal number of steps for both meeples.
        vector<vector<int>> m0(2, vector<int>(n, -1));
        vector<vector<int>> m1(2, vector<int>(n, -1));
        int s0 = solve(0, r, 0, n, ns, m0);
        int s1 = solve(1, b, 1, n, ns, m1);

        // If same # of steps, 0 wins iff # of steps even.
        cout << (s0 != s1 ? s0 > s1 : s0 % 2 == 0) << endl;
    }
}
```

05/5 - On Her Majesty Secret Service

```cpp
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_weight_t, int>> G;
typedef property_map<G, edge_weight_t>::type WM;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, a, s, c, d;
        cin >> n >> m >> a >> s >> c >> d;

        // Read graph G.
        G g(n);
        WM wm = get(edge_weight, g);
        for (int i = 0; i < m; ++i)
        {
            char w;
            int x, y, z;
            cin >> w >> x >> y >> z;
            wm[add_edge(x, y, g).first] = z;
            if (w == 'L')
                wm[add_edge(y, x, g).first] = z;
        }

        // Read agents positions.
        vector<int> as(a);
        for (int i = 0; i < a; ++i)
            cin >> as[i];

        // Read shelters positions.
        vector<int> ss(s);
        for (int i = 0; i < s; ++i)
            cin >> ss[i];

        // Compute shortest paths to shelters from each agent.
        vector<vector<int>> dist(a, vector<int>(n));
        for (int i = 0; i < a; ++i)
            dijkstra_shortest_paths(g, as[i], distance_map(&dist[i][0]));

        // Perform binary search on graphs G'_t.
        int l = 0, r = INT_MAX;
        while (l != r)
        {
            // Construct bipartite graph G'_t of distances <= t - d.
            G g_(a + c * s);
            int t = (l + r) / 2;
            for (int i = 0; i < a; ++i)
                for (int j = 0; j < s; ++j)
                    if (dist[i][ss[j]] != INT_MAX)
                        for (int k = 0; k < c; ++k)
                            if (dist[i][ss[j]] <= t - (k + 1) * d)
                                add_edge(i, a + k * s + j, g_);

            // Compute maximum matching on G'_t.
            vector<int> mate(a + c * s);
            edmonds_maximum_cardinality_matching(g_, &mate[0]);

            // Update search interval.
            if (matching_size(g_, &mate[0]) != (size_t)a) l = t + 1;
            else r = t;
        }

        cout << l << endl;
    }
}
```

05/6 - Poker Chips

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <map>

using namespace std;

int solve(vector<int>& ps, const vector<vector<int>>& cs, int n, const vector<int>& ms, map<vector<int>, int
    >& m)
{
    // Use memoization.
    if (m.count(ps))
        return m[ps];

    // Use recursion on monochromatic subsets.
    int max = 0;
    for (int s = 1; s < 1 << n; ++s)
    {
        bool ok = true;
        int k = 0, c = -1, g;
        vector<int> ps_ = ps;
        for (int i = 0; i < n && ok; ++i)
        {
            if (s & 1 << i)
            {
                ++k, --ps_[i];
                if (ps[i] == -1) ok = false;
                else if (c == -1) c = cs[i][ps[i]];
                else if (cs[i][ps[i]] != c) ok = false;
            }
        }
        if (ok && (g = solve(ps_, cs, n, ms, m) + pow(2, k - 2)) > max)
            max = g;
    }

    return m[ps] = max;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read stack sizes.
        vector<int> ms(n);
        for (int i = 0; i < n; ++i)
            cin >> ms[i];

        // Read stacks.
        vector<int> ps = ms;
        vector<vector<int>> cs(n);
        for (int i = 0; i < n; ++i)
        {
            --ps[i];
            cs[i] = vector<int>(ms[i]);
            for (int j = 0; j < ms[i]; ++j)
                cin >> cs[i][j];
        }

        // Solve recursion using dynamic programming.
        map<vector<int>, int> m;
        cout << solve(ps, cs, n, ms, m) << endl;
    }
}
```

06/1 - Coin Tossing

```
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long, property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long w, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = w;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read rounds results and build supply part of the graph.
        G g(m + n + 2);
        ECM ecm = get(edge_capacity, g);
        REM rem = get(edge_reverse, g);
        vector<int> ps(n, 0);
        int miss = 0;
        for (int i = 0; i < m; ++i)
        {
            int a, b, c;
            cin >> a >> b >> c;
            if (c == 0)
            {
                add(i, m + a, 1, g, ecm, rem);
                add(i, m + b, 1, g, ecm, rem);
                add(m + n, i, 1, g, ecm, rem);
                ++miss;
            }
            else ++ps[c == 1 ? a : b];
        }

        // Build demand part of the graph.
        int diff = 0;
        bool sol = true;
        for (int i = 0; i < n; ++i)
        {
            int s;
            cin >> s;
            if (s - ps[i] >= 0)
            {
                add(m + i, m + n + 1, s - ps[i], g, ecm, rem);
                diff += s - ps[i];
            }
            else sol = false;
        }

        // Compute maximum-flow iff data is consistent.
        if (sol &= diff == miss)
            sol = miss == push_relabel_max_flow(g, m + n, m + n + 1);

        cout << (sol ? "yes" : "no") << endl;
    }
}
```

06/2 - Shopping Trip

```
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long, property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long w, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = w;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, s;
        cin >> n >> m >> s;

        // Read shops locations and streets and build graph.
        G g(n + 1);
        ECM ecm = get(edge_capacity, g);
        REM rem = get(edge_reverse, g);
        for (int i = 0; i < s; ++i)
        {
            int l;
            cin >> l;
            add(l, n, 1, g, ecm, rem);
        }
        for (int i = 0; i < m; ++i)
        {
            int u, v;
            cin >> u >> v;
            add(u, v, 1, g, ecm, rem);
            add(v, u, 1, g, ecm, rem);
        }

        // Check whether max-flow equals number of shops.
        cout << (push_relabel_max_flow(g, 0, n) == s ? "yes" : "no") << endl;
    }
}
```

06/3 - Kingdom Defence

```cpp
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long, property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long w, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = w;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int l, p;
        cin >> l >> p;

        // Build graph.
        int tot = 0;
        G g(l + 2);
        ECM ecm = get(edge_capacity, g);
        REM rem = get(edge_reverse, g);
        for (int i = 0; i < l; ++i)
        {
            int s, d;
            cin >> s >> d;
            add(l, i, s, g, ecm, rem);
            add(i, l + 1, d, g, ecm, rem);
            tot += d;
        }
        for (int i = 0; i < p; ++i)
        {
            int f, t, c, C;
            cin >> f >> t >> c >> C;
            add(f, t, C - c, g, ecm, rem);
            add(l, t, c, g, ecm, rem);
            add(f, l + 1, c, g, ecm, rem);
            tot += c;
        }

        // Check whether max-flow equals total demand.
        cout << (push_relabel_max_flow(g, l, l + 1) == tot ? "yes" : "no") << endl;
    }
}
```

06/4 - A New Hope

```
#include <iostream>
#include <vector>
#include <bitset>
#include <map>

using namespace std;

typedef bitset<14> mask;
typedef map<int, mask> subs;
typedef map<int, subs> center;

int solve(vector<center>& cs, int c, int e, int k, int s, vector<vector<int>>& mem)
{
    // Use memoization.
    if (mem[c][e] != -1)
        return mem[c][e];

    // Use recursion on allowed subsets.
    int max = 0;
    for (int i = 0; i < 1 << s; ++i)
    {
        bool ok = !(i & e);
        for (int j = 0; j < s && ok; ++j)
            if (i & 1 << j)
                ok &= !(cs[c][c][j] & mask(i)).count();
        if (ok)
        {
            int tot = mask(i).count();
            for (auto c_ : cs[c])
            {
                if (c_.first != c)
                {
                    mask e_;
                    for (int j = 0; j < s && ok; ++j)
                        if (i & 1 << j)
                            e_ |= cs[c][c_.first][j];
                    tot += solve(cs, c_.first, e_.to_ulong(), k, s, mem);
                }
            }
            if (tot > max)
                max = tot;
        }
    }

    return mem[c][e] = max;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int k, s, m;
        cin >> k >> s >> m;

        // Read supervision network.
        vector<center> cs(k);
        for (int i = 0; i < m; ++i)
        {
            int u, v, h;
            cin >> u >> v >> h;
            cs[u][v] = subs();
            for (int j = 0; j < h; ++j)
            {
                int x, y;
                cin >> x >> y;
                cs[u][v][x][y] = 1;
            }
        }

        // Solve recursion using dynamic programming.
        vector<vector<int>> mem(k, vector<int>(1 << s, -1));
        cout << solve(cs, 0, 0, k, s, mem) << endl;
    }
}
```

07/1 - Maximize It

```
#include <iostream>

#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

using namespace std;
using namespace CGAL;

typedef Gmpz ET;
typedef Quadratic_program<int> QP;
typedef Quadratic_program_solution<ET> S;

int main()
{
    while (true)
    {
        int p, a, b;
        cin >> p;
        if (p == 0)
            break;
        cin >> a >> b;

        // Solve quadratic program.
        if (p == 1)
        {
            QP qp(SMALLER, true, 0, false, 0);
            qp.set_d(0, 0, 2 * a);
            qp.set_c(1, -b);
            qp.set_a(0, 0, 1);
            qp.set_a(1, 0, 1);
            qp.set_a(0, 1, 4);
            qp.set_a(1, 1, 2);
            qp.set_a(0, 2, -1);
            qp.set_a(1, 2, 1);
            qp.set_b(0, 4);
            qp.set_b(1, a * b);
            qp.set_b(2, 1);
            S s = solve_nonnegative_quadratic_program(qp, ET());
            cout << (s.is_infeasible() ? "no" :
                    to_string((int)floor(-to_double(s.objective_value())))) << endl;
        }
        else
        {
            QP qp(LARGER, false, 0, false, 0);
            qp.set_d(0, 0, 2 * a);
            qp.set_d(2, 2, 2);
            qp.set_c(1, b);
            qp.set_a(0, 0, 1);
            qp.set_a(1, 0, 1);
            qp.set_a(0, 1, 4);
            qp.set_a(1, 1, 2);
            qp.set_a(2, 1, 1);
            qp.set_a(0, 2, -1);
            qp.set_a(1, 2, 1);
            qp.set_a(2, 2, 0);
            qp.set_a(0, 3, 1);
            qp.set_a(1, 4, 1);
            qp.set_b(0, -4);
            qp.set_b(1, -a * b);
            qp.set_b(2, -1);
            qp.set_b(3, 0);
            qp.set_b(4, 0);
            qp.set_r(3, SMALLER);
            qp.set_r(4, SMALLER);
            S s = solve_quadratic_program(qp, ET());
            cout << (s.is_infeasible() ? "no" :
                    to_string((int)ceil(to_double(s.objective_value())))) << endl;
        }
    }
}
```

07/2 - Diet

```
#include <iostream>

#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

using namespace std;
using namespace CGAL;

typedef Gmpz ET;
typedef Quadratic_program<int> LP;
typedef Quadratic_program_solution<ET> S;

int main()
{
    while (true)
    {
        int n, m;
        cin >> n >> m;
        if (n == 0 && m == 0)
            break;

        // Read linear program.
        LP lp(SMALLER, true, 0, false, 0);
        for (int i = 0; i < n; ++i)
        {
            int min, max;
            cin >> min >> max;
            lp.set_b(2 * i, max);
            lp.set_b(2 * i + 1, -min);
        }
        for (int j = 0; j < m; ++j)
        {
            int p;
            cin >> p;
            lp.set_c(j, p);
            for (int i = 0; i < n; ++i)
            {
                int C;
                cin >> C;
                lp.set_a(j, 2 * i, C);
                lp.set_a(j, 2 * i + 1, -C);
            }
        }

        // Solve linear program.
        S s = solve_linear_program(lp, ET());
        cout << (s.is_infeasible() ? "No such diet." :
                to_string((int)to_double(s.objective_value()))) << endl;
    }
}
```

07/3 - Portfolios

```
#include <iostream>

#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

using namespace std;
using namespace CGAL;

typedef Gmpz ET;
typedef Quadratic_program<int> QP;
typedef Quadratic_program_solution<ET> S;

int main()
{
    while (true)
    {
        int n, m;
        cin >> n >> m;
        if (n == 0 && m == 0)
            break;

        // Read quadratic program.
        QP qp(SMALLER, true, 0, false, 0);
        for (int i = 0; i < n; ++i)
        {
            int c, r;
            cin >> c >> r;
            qp.set_a(i, 0, c);
            qp.set_a(i, 1, -r);
        }
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                int v;
                cin >> v;
                if (j <= i)
                    qp.set_d(i, j, 2 * v);
            }
        }

        // Solve quadratic program for each person.
        for (int j = 0; j < m; ++j)
        {
            int C, R, V;
            cin >> C >> R >> V;
            qp.set_b(0, C);
            qp.set_b(1, -R);
            S s = solve_nonnegative_quadratic_program(qp, ET());
            cout << (s.is_infeasible() || s.objective_value() > V ? "No." : "Yes.") << endl;
        }
    }
}
```

07/4 - Inball

```
#include <iostream>
#include <cmath>

#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

using namespace std;
using namespace CGAL;

typedef Gmpz ET;
typedef Quadratic_program<int> LP;
typedef Quadratic_program_solution<ET> S;

int main()
{
    while (true)
    {
        int n, d;
        cin >> n;
        if (n == 0)
            break;
        cin >> d;

        // Read linear program.
        LP lp(SMALLER, false, 0, false, 0);
        for (int i = 0; i < n; ++i)
        {
            int A = 0;
            for (int j = 0; j < d; ++j)
            {
                int a;
                cin >> a;
                lp.set_a(j, i, a);
                A += a * a;
            }
            lp.set_a(d, i, sqrt((double)A));
            int b;
            cin >> b;
            lp.set_b(i, b);
        }
        lp.set_a(d, n, -1);
        lp.set_b(n, 0);
        lp.set_c(d, -1);

        // Solve linear program.
        S s = solve_linear_program(lp, ET());
        cout << (s.is_infeasible() ? "none" : (s.is_unbounded() ? "inf" :
                to_string(-(int)to_double(s.objective_value())))) << endl;
    }
}
```

07/5 - Knights

```
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long, property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long w, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = w;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int m, n, k, c;
        cin >> m >> n >> k >> c;

        // Create graph.
        G g(2 * m * n + 2);
        ECM ecm = get(edge_capacity, g);
        REM rem = get(edge_reverse, g);
        int src = 2 * m * n, snk = src + 1;
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < m; ++j)
            {
                int in = 2 * (m * i + j), out = in + 1;
                add(in, out, c, g, ecm, rem);
                add(out, i == 0 ? snk : out - 2 * m - 1, 1, g, ecm, rem);
                add(out, i == n - 1 ? snk : out + 2 * m - 1, 1, g, ecm, rem);
                add(out, j == 0 ? snk : out - 3, 1, g, ecm, rem);
                add(out, j == m - 1 ? snk : out + 1, 1, g, ecm, rem);
            }
        }

        // Read knights initial positions.
        for (int i = 0; i < k; ++i)
        {
            int x, y;
            cin >> x >> y;
            add(src, 2 * (m * y + x), 1, g, ecm, rem);
        }

        // Compute max-flow.
        cout << push_relabel_max_flow(g, src, snk) << endl;
    }
}
```

08/1 - Graypes

```cpp
#include <iostream>
#include <vector>
#include <limits>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef K::Point_2 P;

int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
            break;

        // Read Graypes coordinates.
        vector<P> gs(n);
        for (int i = 0; i < n; ++i)
            cin >> gs[i];

        // Compute Delaunay triangulation and find shortest edge.
        T t(gs.begin(), gs.end());
        K::FT min = numeric_limits<int>::max(), d;
        for (auto e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e)
            if ((d = t.segment(e).squared_length()) < min)
                min = d;

        cout << (int)ceil(to_double(50 * sqrt(min))) << endl;
    }
}
```

08/2 - Bistro

```
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef K::Point_2 P;

int main()
{
    ios_base::sync_with_stdio(false);
    cout << setiosflags(ios::fixed) << setprecision(0);

    while (true)
    {
        int n, m;
        cin >> n;
        if (n == 0)
            break;

        // Read restaurants locations.
        vector<P> rs(n);
        for (int i = 0; i < n; ++i)
            cin >> rs[i];

        // Triangulate restaurants.
        T t(rs.begin(), rs.end());

        // Find for each new restaurant the distance from the nearest existing.
        cin >> m;
        for (int j = 0; j < m; ++j)
        {
            P p;
            cin >> p;
            cout << squared_distance(p, t.nearest_vertex(p)->point()) << endl;
        }
    }
}
```

08/3 - H1N1

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef T::Face_handle FH;
typedef K::Point_2 P;

int main()
{
    ios_base::sync_with_stdio(false);

    while (true)
    {
        int n, m;
        cin >> n;
        if (n == 0)
            break;

        // Read infected people locations and triangulate them.
        vector<P> is(n);
        for (int i = 0; i < n; ++i)
            cin >> is[i];
        T t(is.begin(), is.end());

        // Find whether each healthy person can escape.
        cin >> m;
        for (int j = 0; j < m; ++j)
        {
            P p;
            long d;
            cin >> p >> d;

            // Perform BFS on triangulation faces.
            set<FH> v;
            queue<FH> q;
            bool e = false;
            v.insert(t.locate(p));
            if (squared_distance(t.nearest_vertex(p)->point(), p) >= d)
                q.push(t.locate(p));
            while (!q.empty() && !e)
            {
                FH f = q.front();
                q.pop();
                if (t.is_infinite(f))
                    e = true;
                for (int i = 0; i < 3; ++i)
                    if (!v.count(f->neighbor(i)) &&
                            squared_distance(f->vertex((i + 1) % 3)->point(),
                            f->vertex((i + 2) % 3)->point()) >= 4 * d)
                        q.push(f->neighbor(i)), v.insert(f->neighbor(i));
            }

            cout << (e ? 'y' : 'n');
        }

        cout << endl;
    }
}
```

08/4 - Germs

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef T::Vertex_handle VH;
typedef K::Point_2 P;

inline int irho(K::FT x) { return ceil(sqrt(x - 0.5)); }

int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
            break;

        // Read dish boundaries.
        int l, b, r, t;
        cin >> l >> b >> r >> t;

        // Read germs positions and triangulate them.
        vector<P> gs(n);
        for (int i = 0; i < n; ++i)
            cin >> gs[i];
        T dt(gs.begin(), gs.end());

        // Compute time to first collision for all germs.
        int i = 0;
        vector<int> ts(n);
        for (auto v = dt.finite_vertices_begin(); v != dt.finite_vertices_end(); ++v)
        {
            // Compute minimum time from boundary.
            int m = std::min(
                    std::min(irho(v->point().x() - l), irho(r - v->point().x())),
                    std::min(irho(v->point().y() - b), irho(t - v->point().y()))), m_;

            // Compute minimum distance from other germs.
            if (n > 1)
            {
                auto e = dt.incident_edges(v);
                do
                    if (!dt.is_infinite(e) && (m_ = irho(sqrt(dt.segment(e).squared_length()) / 2)) < m)
                        m = m_;
                while (++e != dt.incident_edges(v));
            }
            ts[i++] = m;
        }

        // Sort times and take first, middle, and last.
        sort(ts.begin(), ts.end());
        cout << ts.front() << ' ' << ts[n / 2] << ' ' << ts.back() << endl;
    }
}
```

08/5 - Stamps

```
#include <iostream>
#include <vector>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Quadratic_program<K::FT> LP;
typedef K::Segment_2 S;
typedef K::Point_2 P;
typedef Gmpzf ET;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int l, s, w;
        cin >> l >> s >> w;

        // Read lamp positions.
        vector<P> ls(l);
        for (int i = 0; i < l; ++i)
            cin >> ls[i];

        // Read stamps positions and maximum lighting.
        vector<P> ss(s);
        vector<int> ms(s);
        for (int j = 0; j < s; ++j)
            cin >> ss[j] >> ms[j];

        // Read walls.
        vector<S> ws(w);
        for (int k = 0; k < w; ++k)
            cin >> ws[k];

        // Set up linear program.
        LP lp(SMALLER, true, 1, true, 1 << 12);
        for (int j = 0; j < s; ++j)
        {
            lp.set_b(j, ms[j]);
            lp.set_b(s + j, -1);
            for (int i = 0; i < l; ++i)
            {
                K::FT I = 1 / squared_distance(ls[i], ss[j]);
                lp.set_a(i, j, I);
                lp.set_a(i, s + j, -I);
            }
        }

        // If a wall is between a lamp and a stamp, then remove constraint.
        for (int i = 0; i < l; ++i)
            for (int j = 0, ok = 1; j < s; ++j, ok = 1)
                for (int k = 0; k < w && ok; ++k)
                    if (do_intersect(ws[k], S(ls[i], ss[j])))
                        lp.set_a(i, j, 0), lp.set_a(i, s + j, 0), ok = 0;

        // Solve linear program.
        cout << (solve_linear_program(lp, ET()).is_infeasible() ? "no" : "yes") << endl;
    }
}
```

09/1 - Real Estate Market

```
#include <iostream>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor, property<edge_weight_t, long>>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_weight_t>::type EWM;
typedef property_map<G, edge_residual_capacity_t>::type RCM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long c, long w, G& g, ECM& ecm, EWM& ewm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ewm[e.first] = w;
    ecm[e_.first] = 0;
    ewm[e_.first] = -w;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, s, B = 100;
        cin >> n >> m >> s;

        // Create graph.
        G g(n + m + s + 2);
        ECM ecm = get(edge_capacity, g);
        EWM ewm = get(edge_weight, g);
        REM rem = get(edge_reverse, g);
        RCM rcm = get(edge_residual_capacity, g);
        int src = n + m + s, snk = src + 1;

        // Read limits on number of sites.
        for (int i = 0; i < s; ++i)
            add(n + m + i, snk, [](){ int l; cin >> l; return l; }(), 0, g, ecm, ewm, rem);

        // Read states-sites relationships.
        for (int j = 0; j < m; ++j)
            add(n + j, n + m + [](){ int s; cin >> s; return s; }() - 1, 1, 0, g, ecm, ewm, rem);

        // Read bids.
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < m; ++j)
                add(i, n + j, 1, B - [](){ int b; cin >> b; return b; }(), g, ecm, ewm, rem);
            add(src, i, 1, 0, g, ecm, ewm, rem);
        }

        // Compute MinCost-MaxFlow.
        successive_shortest_path_nonnegative_weights(g, src, snk);
        int c = find_flow_cost(g), f = 0;
        for (auto e = out_edges(vertex(src, g), g); e.first != e.second; ++e.first)
            f += ecm[*e.first] - rcm[*e.first];
        cout << f << ' ' << f * B - c << endl;
    }
}
```

09/2 - Satellites

```cpp
#include <iostream>
#include <vector>
#include <queue>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_residual_capacity_t>::type RCM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long c, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int g, s, l;
        cin >> g >> s >> l;

        // Build graph and connect source to stations and satellites to sink.
        G gr(g + s + 2);
        ECM ecm = get(edge_capacity, gr);
        REM rem = get(edge_reverse, gr);
        RCM rcm = get(edge_residual_capacity, gr);
        int src = g + s, snk = src + 1;
        for (int i = 0; i < g; ++i)
            add(src, i, 1, gr, ecm, rem);
        for (int i = 0; i < s; ++i)
            add(g + i, snk, 1, gr, ecm, rem);

        // Read station-satellites connections.
        for (int i = 0; i < l; ++i)
        {
            int u, v;
            cin >> u >> v;
            add(u, g + v, 1, gr, ecm, rem);
        }

        // Compute maxflow.
        push_relabel_max_flow(gr, src, snk);

        // Perform BFS on residual graph.
        std::queue<int> q;
        vector<int> vs(g + s + 2);
        q.push(src);
        vs[src] = true;
        while (!q.empty())
        {
            int u = q.front();
            q.pop();
            for (auto e = out_edges(u, gr); e.first != e.second; ++e.first)
            {
                int v = target(*e.first, gr);
                if (rcm[*e.first] == 1 && !vs[v])
                    q.push(v), vs[v] = true;
            }
        }

        // Find vertex cover.
        int g_ = 0, s_ = 0;
        for (int i = 0; i < g; ++i) if (!vs[i]) ++g_;
        for (int i = g; i < g + s; ++i) if (vs[i]) ++s_;
        cout << g_ << ' ' << s_ << endl;
        for (int i = 0; i < g; ++i) if (!vs[i]) cout << i << ' ';
        for (int i = g; i < g + s; ++i) if (vs[i]) cout << i - g << ' ';
        if (!(g_ == 0 && s_ == 0)) cout << endl;
    }
}
```

09/3 - Algocoon

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <queue>
#include <set>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_residual_capacity_t>::type RCM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long c, G& g, ECM& ecm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ecm[e_.first] = 0;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        cin >> n >> m;

        // Read limbs-figures connections and build graph.
        G g(n);
        ECM ecm = get(edge_capacity, g);
        REM rem = get(edge_reverse, g);
        RCM rcm = get(edge_residual_capacity, g);
        for (int i = 0; i < m; ++i)
        {
            int a, b, c;
            cin >> a >> b >> c;
            add(a, b, c, g, ecm, rem);
        }

        // Find the minimum s-t-max-flow.
        int s = 0, t = 0, f, max = numeric_limits<int>::max();
        for (int i = 0; i < n; ++i)
            if ((f = push_relabel_max_flow(g, i, (i + 1) % n)) < max)
                max = f, s = i, t = (i + 1) % n;

        // Recompute optimal residual graph.
        cout << push_relabel_max_flow(g, s, t) << endl;

        // Perform BFS on residual graph.
        std::queue<int> q;
        set<int> vs;
        q.push(s);
        vs.insert(s);
        while (!q.empty())
        {
            int u = q.front();
            q.pop();
            for (auto e = out_edges(u, g); e.first != e.second; ++e.first)
            {
                int v = target(*e.first, g);
                if (rcm[*e.first] >= 1 && !vs.count(v))
                    q.push(v), vs.insert(v);
            }
        }

        // Find visited nodes.
        cout << vs.size() << " ";
        for (auto v : vs)
            cout << v << " ";
        cout << endl;
    }
}
```

09/4 - Canteen

```
#include <iostream>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor, property<edge_weight_t, long>>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_weight_t>::type EWM;
typedef property_map<G, edge_residual_capacity_t>::type RCM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long c, long w, G& g, ECM& ecm, EWM& ewm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ewm[e.first] = w;
    ecm[e_.first] = 0;
    ewm[e_.first] = -w;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Create graph.
        G g(n + 2);
        ECM ecm = get(edge_capacity, g);
        EWM ewm = get(edge_weight, g);
        REM rem = get(edge_reverse, g);
        RCM rcm = get(edge_residual_capacity, g);
        int src = n, snk = n + 1;

        // Read daily amounts of menus and costs.
        for (int i = 0; i < n; ++i)
        {
            int a, c;
            cin >> a >> c;
            add(src, i, a, c, g, ecm, ewm, rem);
        }

        // Read daily amounts of students and prices.
        int S = 0;
        for (int i = 0; i < n; ++i)
        {
            int s, p;
            cin >> s >> p;
            add(i, snk, s, 20 - p, g, ecm, ewm, rem);
            S += s;
        }

        // Read nightly amounts of menus and costs.
        for (int i = 0; i < n - 1; ++i)
        {
            int v, e;
            cin >> v >> e;
            add(i, i + 1, v, e, g, ecm, ewm, rem);
        }

        // Compute MinCost-MaxFlow.
        successive_shortest_path_nonnegative_weights(g, src, snk);
        int c = find_flow_cost(g), f = 0;
        for (auto e = out_edges(vertex(src, g), g); e.first != e.second; ++e.first)
            f += ecm[*e.first] - rcm[*e.first];
        cout << (f == S ? "possible " : "impossible ") << f << ' ' << 20 * f - c << endl;
    }
}
```

09/5 - Casino Royale

```cpp
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor, property<edge_weight_t, long>>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_weight_t>::type EWM;
typedef property_map<G, edge_residual_capacity_t>::type RCM;
typedef property_map<G, edge_reverse_t>::type REM;
typedef graph_traits<G>::edge_descriptor E;

E add(int u, int v, long c, long w, G& g, ECM& ecm, EWM& ewm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ewm[e.first] = w;
    ecm[e_.first] = 0;
    ewm[e_.first] = -w;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
    return e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m, l, Q = 1 << 7;
        cin >> n >> m >> l;

        // Build graph.
        G g(n + 1);
        ECM ecm = get(edge_capacity, g);
        EWM ewm = get(edge_weight, g);
        REM rem = get(edge_reverse, g);
        RCM rcm = get(edge_residual_capacity, g);
        add(n, 0, l, 0, g, ecm, ewm, rem);
        for (int i = 0; i < n - 1; ++i)
            add(i, i + 1, l, Q, g, ecm, ewm, rem);

        // Read missions.
        vector<E> ms(m);
        for (int i = 0; i < m; ++i)
        {
            int x, y, q;
            cin >> x >> y >> q;
            ms[i] = add(x, y, 1, (y - x) * Q - q, g, ecm, ewm, rem);
        }

        // Compute MinCost-MaxFlow.
        successive_shortest_path_nonnegative_weights(g, n, n - 1);
        int p = 0;
        for (int i = 0; i < m; ++i)
        {
            int s = source(ms[i], g), t = target(ms[i], g);
            if (rcm[ms[i]] == 0)
                p += Q * (t - s) - ewm[ms[i]];
        }

        cout << p << endl;
    }
}
```

10/1 - Odd Route

```
#include <iostream>
#include <vector>
#include <limits>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_weight_t, int>> G;
typedef property_map<G, edge_weight_t>::type WM;

int main()
{
    int T;
    cin >> T;

    while (T--)
    {
        int n, m, s, t;
        cin >> n >> m >> s >> t;

        // Create graph.
        G g(4 * n);
        WM wm = get(edge_weight, g);
        for (int i = 0; i < m; ++i)
        {
            int u, v, w;
            cin >> u >> v >> w;
            wm[add_edge(u, (2 + (w % 2)) * n + v, g).first] = w;
            wm[add_edge(n + u, (2 + !(w % 2)) * n + v, g).first] = w;
            wm[add_edge(2 * n + u, (w % 2) * n + v, g).first] = w;
            wm[add_edge(3 * n + u, !(w % 2) * n + v, g).first] = w;
        }

        // Compute shortest s'-t'-path.
        vector<int> dist(4 * n);
        dijkstra_shortest_paths(g, s, distance_map(&dist[0]));
        cout << (dist[3 * n + t] != numeric_limits<int>::max() ?
                to_string(dist[3 * n + t]) : "no") << endl;
    }
}
```

10/2 - Light the Stage

```cpp
#include <iostream>
#include <vector>
#include <map>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef K::Point_2 P;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int m, n, h;
        cin >> m >> n;

        // Read participants and lamps postions.
        vector<P> ps(m), ls(n);
        vector<int> rs(m);
        map<P, int> imap;
        for (int j = 0; j < m; ++j)
            cin >> ps[j] >> rs[j];
        cin >> h;
        for (int i = 0; i < n; ++i)
        {
            cin >> ls[i];
            imap[ls[i]] = i;
        }

        // Triangulate lamps.
        T t(ls.begin(), ls.end());

        // Find for each partecipant which (if any) lamp hits him first.
        int last = -1;
        bool nw = true;
        vector<int> ws;
        vector<bool> hs(m);
        for (int j = 0; j < m; ++j)
        {
            P l = t.nearest_vertex(ps[j])->point();
            if (sqrt(to_double(squared_distance(ps[j], l))) >= double(h + rs[j]))
            {
                cout << j << ' ';
                nw = false;
            }

            // Find the first lamp to hit this participant, if there are still no winners.
            else if (nw)
            {
                for (int i = 0; imap[l] >= last && i <= imap[l]; ++i)
                {
                    if (sqrt(to_double(squared_distance(ps[j], ls[i]))) < double(h + rs[j]))
                    {
                        if (i > last) last = i, ws.clear();
                        if (i == last) ws.push_back(j);
                        break;
                    }
                }
            }
        }

        if (nw)
            for (int w : ws)
                cout << w << ' ';
        cout << endl;
    }
}
```

10/3 - Bonus Level

```cpp
#include <iostream>
#include <vector>

using namespace std;

int solve(int i1, int j1, int i2, int j2, int n, vector<vector<int>>& as, vector<vector<vector<int>>>& m)
{
    // Base cases.
    if (i1 < 0 || i1 >= n || j1 < 0 || j1 >= n || i2 < 0 || i2 >= n || j2 < 0 || j2 >= n)
        return 0;

    // Paths cannot intersect.
    if (i1 == i2 && j1 == j2)
        return 0;

    // Use memoization.
    if (m[i1][j1][i2] != -1)
        return m[i1][j1][i2];

    // Use recursion.
    return m[i1][j1][i2] = max(
            max(solve(i1 + 1, j1, i2 + 1, j2, n, as, m), solve(i1, j1 + 1, i2, j2 + 1, n, as, m)),
            max(solve(i1 + 1, j1, i2, j2 + 1, n, as, m), solve(i1, j1 + 1, i2 + 1, j2, n, as, m)))
            + as[i1][j1] + as[i2][j2];
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read map.
        vector<vector<int>> as(n, vector<int>(n));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                cin >> as[i][j];

        // Solve recursion using dynamic programming.
        vector<vector<vector<int>>> m(n, vector<vector<int>>(n, vector<int>(n, -1)));
        cout << solve(0, 1, 1, 0, n, as, m) + as[0][0] + as[n - 1][n - 1] << endl;
    }
}
```

10/4 - Sith

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace boost;
using namespace CGAL;

typedef adjacency_list<vecS, vecS, undirectedS> G;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Triangulation_face_base_2<K> FB;
typedef Triangulation_vertex_base_with_info_2<int, K> VB;
typedef Triangulation_data_structure_2<VB, FB> TDS;
typedef Delaunay_triangulation_2<K, TDS> T;
typedef K::Point_2 P;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, r;
        cin >> n >> r;
        long r2 = pow(r, 2);

        // Read planets positions.
        vector<P> ps(n);
        for (int i = 0; i < n; ++i)
            cin >> ps[i];

        // Find optimal k using binary search.
        int lk = 0, rk = n - 1, mk;
        while (lk != rk)
        {
            // Triangulate remaining planets and label them.
            int i = 0;
            mk = (lk + rk) / 2;
            T t(ps.begin() + mk, ps.end());
            for (auto v = t.finite_vertices_begin(); v != t.finite_vertices_end(); ++v)
                v->info() = i++;

            // Build distance-induced graph.
            G g(n - mk);
            for (auto e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e)
                if (t.segment(e).squared_length() <= r2)
                    add_edge(e->first->vertex((e->second + 1) % 3)->info(),
                             e->first->vertex((e->second + 2) % 3)->info(), g);

            // Find largest connected component.
            vector<int> cs(n - mk);
            int c = connected_components(g, &cs[0]);
            vector<int> ss(c);
            for (size_t i = 0; i < cs.size(); ++i)
                ++ss[cs[i]];

            // Update search interval.
            if (mk <= *max_element(ss.begin(), ss.end())) lk = mk + 1;
            else rk = mk;
        }

        cout << lk - 1 << endl;
    }

    return 0;
}
```

11/1 - Clues

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

11/2 - Punch

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

11/3 - Carsharing

```
#include <iostream>
#include <vector>
#include <map>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> T;
typedef adjacency_list<vecS, vecS, directedS, no_property,
        property<edge_capacity_t, long, property<edge_residual_capacity_t, long,
        property<edge_reverse_t, T::edge_descriptor, property<edge_weight_t, long>>>>> G;
typedef property_map<G, edge_capacity_t>::type ECM;
typedef property_map<G, edge_weight_t>::type EWM;
typedef property_map<G, edge_reverse_t>::type REM;

void add(int u, int v, long c, long w, G& g, ECM& ecm, EWM& ewm, REM& rem)
{
    auto e = add_edge(u, v, g);
    auto e_ = add_edge(v, u, g);
    ecm[e.first] = c;
    ewm[e.first] = w;
    ecm[e_.first] = 0;
    ewm[e_.first] = -w;
    rem[e.first] = e_.first;
    rem[e_.first] = e.first;
}

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, s, tmax = 100000, pmax = 100;
        cin >> n >> s;

        // Create graph.
        G g;
        ECM ecm = get(edge_capacity, g);
        EWM ewm = get(edge_weight, g);
        REM rem = get(edge_reverse, g);
        int src = add_vertex(g), snk = add_vertex(g);

        // Read initial number of cars at each station.
        int tot = 0;
        vector<int> ls(s);
        for (int i = 0; i < s; ++i)
            tot += [&ls, i](){ cin >> ls[i]; return ls[i]; }();

        // Add begin and end node for each station.
        vector<map<int, int>> ts(s);
        for (int i = 0; i < s; ++i)
            add(src, ts[i][0] = add_vertex(g), ls[i], 0, g, ecm, ewm, rem),
            add(ts[i][tmax] = add_vertex(g), snk, tot, 0, g, ecm, ewm, rem);

        // Read queries and add corresponding edges to the graph.
        for (int i = 0; i < n; ++i)
        {
            int s, t, d, a, p;
            cin >> s >> t >> d >> a >> p;
            --s, --t;

            // Add time vertices if not yet there and relative edge.
            if (!ts[s].count(d)) ts[s][d] = add_vertex(g);
            if (!ts[t].count(a)) ts[t][a] = add_vertex(g);
            add(ts[s][d], ts[t][a], 1, (a - d) * pmax - p, g, ecm, ewm, rem);
        }

        // Add parking edges between time nodes of each station.
        for (int i = 0; i < s; ++i)
            for (auto it = ts[i].begin(); it != prev(ts[i].end(), 1); ++it)
                add(it->second, next(it, 1)->second, tot,
                        (next(it, 1)->first - it->first) * pmax, g, ecm, ewm, rem);

        // Compute MinCost-MaxFlow.
        successive_shortest_path_nonnegative_weights(g, src, snk);
        cout << (long)tmax * pmax * tot - find_flow_cost(g) << endl;
    }
}
```

11/4 - Planks

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

12/1 - New Tiles

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

12/2 - GoldenEye

```
#include <iostream>
#include <vector>
#include <map>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

using namespace std;
using namespace boost;
using namespace CGAL;

typedef adjacency_list<vecS, vecS, undirectedS> G;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef Delaunay_triangulation_2<K> T;
typedef K::Point_2 P;

// Passing only first test set.
int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, m;
        long p;
        cin >> n >> m >> p;

        // Read and triangulate jammers.
        vector<P> js(n);
        map<P, int> p2i;
        for (int i = 0; i < n; ++i)
            p2i[([&js, i](){ cin >> js[i]; return js[i]; }())] = i;
        T dt(js.begin(), js.end());

        // Build intersection graph and find its components.
        G g(n);
        for (int i = 0; i < n; ++i)
            for (int j = i; j < n; ++j)
                if (squared_distance(js[i], js[j]) <= p)
                    add_edge(i, j, g);
        vector<int> comp(n);
        connected_components(g, &comp[0]);

        // Process missions.
        for (int j = 0; j < m; ++j)
        {
            P s, t, s_, t_;
            cin >> s >> t;
            s_ = dt.nearest_vertex(s)->point(), t_ = dt.nearest_vertex(t)->point();
            cout << (squared_distance(s, s_) <= p / 4 && squared_distance(t, t_) <= p / 4
                    && comp[p2i[s_]] == comp[p2i[t_]] ? 'y' : 'n');
        }

        cout << endl << 4 * p << endl << p << endl;
    }
}
```

12/3 - Corbusier

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n, i, k;
        cin >> n >> i >> k;

        // Read disks heights.
        vector<int> hs(n);
        for (int i = 0; i < n; ++i)
            cin >> hs[i];

        // All possible remainders.
        vector<bool> rs(k);

        // Try to add each disk.
        for (int l = 0; l < n; ++l)
        {
            // Find new possible remainders by adding this disk to the existing.
            vector<bool> ts(k);
            for (int m = 0; m < k; ++m)
                if (rs[m])
                    ts[(m + hs[l]) % k] = true;

            // Merge new remainders.
            for (int m = 0; m < k; ++m)
                rs[m] = rs[m] || ts[m];

            // Add this disk as a singleton set.
            rs[hs[l] % k] = true;
        }

        cout << (rs[i] ? "yes" : "no") << endl;
    }
}
```

12/4 - Placing Knights

```
#include <iostream>
#include <vector>
#include <map>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> G;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        int n;
        cin >> n;

        // Read chessboard and count missing cells.
        int m = 0;
        vector<vector<int>> xs(n, vector<int>(n));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (![&xs, i, j](){ cin >> xs[i][j]; return xs[i][j]; }())
                    ++m;

        // Build bipartite graph.
        G g(n * n);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (xs[i][j])
                    for (auto k : map<int, int>{ {i - 1, j - 2}, {i - 1, j + 2},
                            {i + 1, j - 2}, {i + 1, j + 2},
                            {i - 2, j - 1}, {i - 2, j + 1},
                            {i + 2, j - 1}, {i + 2, j + 1} })
                        if (k.first >= 0 && k.first < n && k.second >= 0 && k.second < n
                                && xs[k.first][k.second])
                            add_edge(n * i + j, n * k.first + k.second, g);

        // Compute |max. match.| = |min. vtx. cov.| = |G| - |max. ind. set|.
        vector<int> mate(n * n);
        edmonds_maximum_cardinality_matching(g, &mate[0]);
        cout << n * n - m - matching_size(g, &mate[0]) << endl;
    }
}
```

12/5 - Radiation

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

12/6 - The Empire Strikes Back

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

13/1 - Bob's Burden

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

13/2 - DHL

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

13/3 - Sweepers

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

13/4 - Portfolios Revisited

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

13/5 - The Phantom Menace

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```

14/1 - Cantonal Courier

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int t;
    cin >> t;

    while (t--)
    {
        /*int n;
        cin >> n;

        vector<int> xs(n);
        for (int i = 0; i < n; ++i)
            cin >> xs[i];*/

        cout << 0 << endl;
    }
}
```