Algorithms Lab

Alberto Montes
malberto@student.ethz.ch

January 22, 2017

Contents

Chapter 1

Fundamentals

1.1 Introduction

Build the sum

***Keywords*—**

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

using namespace std;

void do_sum(){
    int n; cin>> n;
    float sum = 0.0;
    float sumant;
    for (int i =0; i < n; i++) {
        cin >> sumant;
        sum += sumant;
    }

    cout << sum << endl;
}

int main() {
    int T; cin >> T;
    for (int t=0; t < T; t++){
        do_sum();
    }
}
```

Even Pairs

Keywords— Precompute

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

using namespace std;

void do_even_count(){
    int n; cin >> n;
    vector<int> bits(n);
    for (int i =0; i < n; i++)
        cin >> bits [i];

    // Precompute Si's
    vector<int> partial_sum(n);
    partial_sum[0] = bits[0];
    for (int i = 1; i < n; i++) {
        partial_sum[i] = partial_sum[i-1] + bits[i];
    }

    vector<int> evens(n), odds(n);
    evens[0] = 0; odds[0] = 0;
    if (bits[0] == 0)
        evens[0] = 1;
    else
        odds[0] = 1;
    for (int i = 1; i < n; i++) {
        evens[i] = evens[i-1];
        odds[i] = odds[i-1];
        if (partial_sum[i] % 2 == 0)
            evens[i]++;
        else
            odds[i]++;
    }

    // Calculate the result
    int counter = 0;
    if (bits[0] == 0)
        counter++;
    for (int i =1; i < n; i++){
        if (partial_sum[i] % 2 == 0)
            counter += evens[i-1] + 1;
        else
            counter += odds[i-1];

    }
    cout << counter << endl;
}

int main() {
    int T; cin >> T;
    for (int t=0; t < T; t++){
        do_even_count();
    }
}
```

Dominoes

***Keywords*—**

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

using namespace std;

void compute_dominoes_falling(){
    int n; cin >> n;
    vector<unsigned long int> dominoes_height(n);
    for (int i =0; i < n; i++)
        cin >> dominoes_height[i];

    int max = 1;

    for (int i = 0; i < n; i++) {
        if (i + dominoes_height[i] > max)
            max = i + dominoes_height[i];
        else if (max == i + 1)
            break;
    }
    if (max > n)
        max = n;

    cout << max << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        compute_dominoes_falling();
    }
}
```

Even Matrices

Keywords— Precompute

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

using namespace std;


// Implementation in O(n^3) time
void do_even_count_matrices(){
    int n; cin >> n;
    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];

    // Precompute pMi's
    vector<vector<int> > pM(n+1, vector<int>(n+1));
    for (int i  = 0; i <= n; i++) {
        pM[i][0] = 0;
        pM[0][i] = 0;
     }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            pM[i][j] = pM[i-1][j] + pM[i][j-1] - pM[i-1][j-1] + M[i-1][j-1];
        }
    }

    int counter = 0;
    for (int i1 = 1; i1 <= n; i1++) {
        for (int i2 = i1; i2 <= n; i2++) {
            // We reduce the probelm to one dimension
            vector<int> S(n+1);     // Do even pairs on array S
            vector<int> pS(n+1);    // pS contains partial sums of S
            pS[0] = 0;
            for (int k = 1; k <= n; k++) {
                S[k] = pM[i2][k] - pM[i2][k-1] - pM[i1-1][k] + pM[i1-1][k-1];
                pS[k] = pS[k-1] + S[k];
            }

            // Do even pairs O(n) algorithm on array S
            int onedim_sol = 0;
            int even = 0, odd = 0;
            for (int j = 1; j <=n; j++) {
                // even = # of partial sums of array (S[1], ..., S[j-1]) that are even.
                // odd = # of partial sums of array (S[1], ..., S[j-1]) that are odd.
                if (pS[j] % 2 == 0){
                    onedim_sol +=  even + 1;
                    even++;
                }
                else {
                    onedim_sol += odd;
                    odd++;
                }
            }
            counter += onedim_sol;
        }
    }

    cout << counter << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        do_even_count_matrices();
    }
}
```

False Coin

Keywords— Tree Search

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

using namespace std;

void test_false_coin() {
    int n; cin >> n;

    vector<int> candidates(n, 0);
    vector<int> lw_coins(n, 0), hw_coins(n, 0);

    int num_trials; cin >> num_trials;

    for (int i = 0; i < num_trials; i++) {
        int num_per_side; cin >> num_per_side;

        vector<int> left_coins(n, false), right_coins(n, false);
        int left_pos, right_pos;
        for (int j = 0; j < num_per_side; j++) {
            cin >> left_pos;
            left_coins[left_pos-1] = true;
        }
        for (int j = 0; j < num_per_side; j++) {
            cin >> right_pos;
            right_coins[right_pos-1] = true;
        }

        char comparison; cin >> comparison;

        // For each trial eliminate all the possible candidates depending where on the balance
        // appear
        for (int j = 0; j < n; j++) {
            switch (comparison) {
                case '=': {
                    if (left_coins[j] || right_coins[j])
                        candidates[j] = -1;
                    else if (candidates[j] == 0)
                        candidates[j] = 1;
                    break;
                }
                case '<': {
                    if (left_coins[j]) {
                        lw_coins[j]++;
                        if (hw_coins[j] > 0) {
                            candidates[j] = -1;
                        } else if (candidates[j] == 0) {
                            candidates[j] = 1;
                        }
                    } else if (right_coins[j]) {
                        hw_coins[j]++;
                        if (lw_coins[j] > 0) {
                            candidates[j] = -1;
                        } else if (candidates[j] == 0) {
                            candidates[j] = 1;
                        }
                    } else {
                        candidates[j] = -1;
                    }
                    break;
                }
                case '>': {
                    if (right_coins[j]) {
                        lw_coins[j]++;
                        if (hw_coins[j] > 0) {
                            candidates[j] = -1;
                        } else if (candidates[j] == 0) {
                            candidates[j] = 1;
                        }
                    } else if (left_coins[j]) {
                        hw_coins[j]++;
                        if (lw_coins[j] > 0) {
                            candidates[j] = -1;
                        } else if (candidates[j] == 0) {
                            candidates[j] = 1;
                        }
                    } else {
                        candidates[j] = -1;
```

```
                    }
                    break;
                }
                default: break;
            }
        }
    }

    int count_candidates = 0;
    int coin_id = 0;
    for (int i = 0; i < n; i++) {
        if (candidates[i] == 1) {
            count_candidates++;
            coin_id = i + 1;
        }
    }
    // If there is more than one candidate there is not an unique solution
    if (count_candidates > 1) {
        coin_id = 0;
    }
    cout << coin_id << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        test_false_coin();
    }
}
```

Deck of Cards

Keywords— Sliding Window

```cpp

#include <vector>
#include <iostream>
#include <limits>
#include <stdlib.h>

using namespace std;

// Implementation in O(n)
void deck_of_cards(){
    long n; cin >> n;
    long k; cin >> k;

    vector<long> values(n);
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }

    long i_r = 0, j_r = 0;
    long start = 0, end = 0;
    long sum = values[0];
    long min_value = abs(k-sum);

    // Slide a window and keep track of the minimum difference between k and the sum of values
    while (start < n) {
        if (sum <= k) {
            if (end < n-1) {
                sum += values[++end];
            } else { break; }
        } else {
            sum -= values[start++];
        }
        if (abs(k-sum) < min_value) {
            min_value = abs(k-sum);
            i_r = start;
            j_r = end;
        }
    }

    cout << i_r << ' ' << j_r << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        deck_of_cards();
    }
}
```

1.2 Binary Search, Sliding Window, Graph Traversals, Greedy

Search Snippets

Keywords— Sliding Window

```cpp
/*
Implementation sliding window.
Hint: to improve the efficiency, instead of loop along all the appearance vector to see if all the
words appear, keep up a counter and changing it with conditions. It is much more efficient and
    clean
*/
#include <vector>
#include <iostream>
#include <algorithm>
#include <math.h>
#include <limits.h>

using namespace std;

struct word {
    int id;
    int position;
};

void search_snippets() {
    int n; cin >> n;

    vector<int> m(n);
    int t = 0;              // total word appearance
    for (int i = 0; i < n; i++) {
        cin >> m[i];
        t += m[i];
    }

    vector<word > w; w.reserve(t);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m[i]; j++) {
            word wo; wo.id = i;
            cin >> wo.position;
            w.push_back(wo);
        }
    }
    sort(w.begin(), w.end(), [](word a, word b) {
        return a.position < b.position;
    });

    vector<int> a(n, 0);
    int min_d = w[t-1].position - w[0].position + 1;
    int s = 0, e = 0, u = n; // start, end, and uncovered counter
    a[w[s].id]++; u--;
    // Sliding Window
    while (e != t) {
        while (e < t - 1 && u > 0) {
            e++;
            if (a[w[e].id] == 0) u--;
            a[w[e].id]++;
        }
        if (u != 0) break;
        do {
            a[w[s].id]--;
            if (a[w[s].id] == 0) ++u;
        } while (++s != e && u==0);     // Even you go out of the while loop, s will have been
                                        // incresed undiredbly
        min_d = min(min_d, w[e].position - w[s-1].position + 1);
    }

    cout << min_d << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        search_snippets();
    }
}
```

Boats

Keywords— Interval Scheduling

```cpp
/*
Implementation using two fors to iterate

Correct
*/
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

void boats() {
    long n; cin >> n;

    vector<pair<long, long> > boats(n);
    for (int i = 0; i < n; i++) {
        cin >> boats[i].second;
        cin >> boats[i].first;
    }

    sort(boats.begin(), boats.end(), [](pair<long, long> a, pair<long, long> b) {
        return a.first < b.first;
    });

    long current_last_position = boats[0].first;
    long number_boats = 1;
    long i = 1;
    while (i < n) {
        if (current_last_position > boats[i].first) {
            i++;
        } else if (boats[i].first - current_last_position > boats[i].second) {
            number_boats++;
            current_last_position = boats[i].first;
            i++;
        } else {
            long first_end = boats[i].second + current_last_position;
            long min_position = i;
            for (long j = i+1; j < n; j++) {
                long end_position;
                if (boats[j].first - current_last_position > boats[j].second) {
                    end_position = boats[j].first;
                } else {
                    end_position = current_last_position + boats[j].second;
                }
                if (end_position < first_end) {
                    first_end = end_position;
                    min_position = j;
                }
            }
            number_boats++;
            current_last_position = first_end;
            i = min_position + 1;
        }
    }

    cout << number_boats << endl;


}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        boats();
    }
}
```

Moving Books

Keywords— Greedy

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

using namespace std;


void moving_books() {
    long n; cin >> n;
    long m; cin >> m;

    vector<long> s(n), w(m);
    for (int i = 0; i < n; i++) {
        cin >> s[i];
    }
    for (int i = 0; i < m; i++) {
        cin >> w[i];
    }

    // Check if the heviest box can not be bringed by nobody
    int max_s = *max_element(s.begin(), s.end());
    int max_w = *max_element(w.begin(), w.end());
    if (max_w > max_s) {
        cout << "impossible\n";
        return;
    }

    multiset<int, greater<int> > ws;
    sort(s.begin(), s.end(), greater<int>());

    for (int i = 0; i < m; i++) ws.insert(w[i]);
    int r = 0;
    while (!ws.empty()) {
        r++;
        for (int i = 0; i < n; i++) {
            auto j = ws.lower_bound(s[i]);
            if (j != ws.end()) {
                ws.erase(j);
            } else {
                break;
            }
        }
    }

    cout << 3*r-1 << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        moving_books();
    }
}
```

Evolution

Keywords— DFS, Binary Search

```cpp
#include <vector>
#include <map>
#include <iostream>
#include <string>
#include <algorithm>
#include <unordered_map>

using namespace std;


// Binary Search
int binary(int b, vector<int>& path, vector<int>& age) {
    int l = 0; int r = path.size()-1;
    while(l != r){
        int m = (l+r)/2;
        if (age[path[m]] > b) l = m+1; else r = m;
    }
    return path[l];
}

// DFS
void dfs(int u, vector<vector<int> >& tree, vector<int>& path, vector<vector<pair<int,int> > >&
    query, vector<int>& result, vector<int>& age) {
    // process queries
    for (int i = 0; i < query[u].size(); i++) {
        result[query[u][i].second] = binary(query[u][i].first,path,age);
    }

    // continue search
    for (int i = 0; i < tree[u].size(); ++i){
        int v = tree[u][i];
        path.push_back(v);
        dfs(v,tree,path,query,result,age);
    }
    path.pop_back();
}

void evolution() {
    int n, q; cin >> n >> q;

    // Store the names and respective ages
    unordered_map<string, int> species_to_index;
    vector<string> species(n);
    vector<int> age(n);
    string name;
    for (int i = 0; i < n; i++) {
        cin >> name;
        species_to_index[name] = i;
        species[i] = name;
        cin >> age[i];
    }

    // Find root
    int root = max_element(age.begin(), age.end()) - age.begin();

    // Read tree
    vector<vector<int> > tree(n);
    string child, parent;
    for (int i = 0; i < n - 1; i++) {
        cin >> child >> parent;
        tree[species_to_index[parent]].push_back(species_to_index[child]);
    }

    // Read queries: for each species store a vector of queries consisting of the age b and the
    // index of the query i
    vector<vector<pair<int, int> > > query(n);
    for (int i = 0; i < q; i++) {
        cin >> name;
        int b; cin >> b;
        query[species_to_index[name]].push_back(make_pair(b, i));
    }

    // Process queries in one tree transversal
    vector<int> path; path.push_back(root);
    vector<int> result(q);
    dfs(root, tree, path, query, result, age);

```

```
77      // Output result
78      for (int i = 0; i < q; i++) {
79          cout << species[result[i]];
80          if ( i < q - 1 ) cout << " ";
81      }
82      cout << endl;
83  }
84
85  int main() {
86      ios_base::sync_with_stdio(false);
87      int T; cin >> T;
88      for (int t=0; t < T; t++){
89          evolution();
90      }
91  }
```

Octopussy

Keywords— Greedy

```cpp

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

long find_minimum_time(vector<long> &t, long pos) {
    if (pos == 0) {
        return t[pos];
    }
    long upper_pos = (pos % 2 == 1) ? (pos - 1) / 2 : (pos - 2) / 2;
    long next_pos = (pos % 2 == 1) ? pos + 1 : pos - 1;
    long min_upper_pos = find_minimum_time(t, upper_pos);
    // Check if the time limit of the pair bomb is more limiting or not.
    if (t[next_pos] <= min_upper_pos - 2) {
        return min(t[pos], min_upper_pos - 1);
    } else {
        return min(t[pos], min_upper_pos - 2);
    }
}

void octopussy() {
    int n; cin >> n;

    vector<long> t(n);
    for (int i = 0; i < n; i++) {
        cin >> t[i];
    }

    vector<bool> disarmed(n, false);

    auto cmp = [&t](pair<long, long> left, pair<long, long> right) {
        if ( left.second > right.second ) { return true; }
        else if ( left.second == right.second ) { return t[left.first] > t[right.first]; }
        else { return false; }
    };
    priority_queue<pair<long, long>, vector<pair<long, long> >, decltype(cmp)> bombs_queue(cmp);

    for (long j = (n-3)/2+1; j < n; j++) {
        long min_time = find_minimum_time(t, j);
        bombs_queue.push(pair<long, long>(j, min_time));
    }

    long t_time = 0;
    while( !bombs_queue.empty() ) {
        auto p_bomb = bombs_queue.top();
        bombs_queue.pop();
        // The most priority bomb has expired its time...BOOOM!
        long b_index = p_bomb.first;
        if (t[b_index] <= t_time) { break; }
        // Marked the bomb as disarmed
        disarmed[b_index] = true;
        t_time++;
        /* Now see if the bomb next to it has been desarmed to let the bomb standing over them to
        enter the priority queue. If not...too late...BOOOOOM!! */
        if (b_index == 0) { continue; } // The bomb at index 0 has no bomb standing over it.
        long next_b_index = (b_index % 2 == 1) ? b_index + 1 : b_index - 1;
        if ( disarmed[next_b_index] ) {
            long std_b_index = (b_index % 2 == 1) ? (b_index - 1) / 2 : (b_index - 2) / 2;
            bombs_queue.push(pair<long, long>(std_b_index, find_minimum_time(t, std_b_index)));
        }
    }

    bool all_disarmed = all_of(disarmed.begin(), disarmed.end(), [](bool v){return v;});
    string response = (all_disarmed) ? "yes" : "no";
    cout << response << endl;
}


int main() {
    int t; cin >> t;
    for (int i = 0; i < t; i++) {
        octopussy();
    }
}
```

1.3 CGAL

Hit

Keywords— Intersection

```
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;

using namespace std;


void compute_hit(int n) {
    long x, y, a, b;
    cin >> x >> y >> a >> b;
    K::Point_2 xy(x, y), ab(a, b);
    K::Ray_2 ray(xy, ab);

    bool hit = false;
    for (int i = 0; i < n; i++) {
        long r, s, t, u;
        cin >> r >> s >> t >> u;
        if (hit) {
            continue;
        }
        K::Point_2 rs(r, s), tu(t, u);
        K::Segment_2 seg(rs, tu);
        if (CGAL::do_intersect(ray, seg)) {
            hit = true;
        }
    }

    string result = (hit) ? "yes" : "no";
    cout << result << endl;

}


int main() {
    ios_base::sync_with_stdio(false);
    int nb_obstacles; cin >> nb_obstacles;

    while(nb_obstacles > 0) {
        compute_hit(nb_obstacles);
        cin >> nb_obstacles;
    }
    return 0;
}
```

First Hit

Keywords— Intersection

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <vector>
#include <algorithm>
#include <type_traits>
#include <stdexcept>
#include <limits>

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel       K;
typedef result_of<K::Intersect_2(K::Ray_2,K::Segment_2)>::type  IT;

// round down to next double
double floor_to_double(const K::FT& x) {
    double a = floor(CGAL::to_double(x));
    while (a > x) a -= 1;
    while (a+1 <= x) a += 1;
    return a;
}

// read segment cin; as each coordinate can be represented as a
// long, this is significantly faster than K::Segment_2 s; cin >> s;
inline K::Segment_2 read_segment() {
    long x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    return K::Segment_2(K::Point_2(x1,y1), K::Point_2(x2,y2));
}

// clip/set target of s to o
void shorten_segment(K::Segment_2& s, const IT& o) {
    if (const K::Point_2* p = boost::get<K::Point_2>(&*o))
        s = K::Segment_2(s.source(), *p);
    else if (const K::Segment_2* t = boost::get<K::Segment_2>(&*o))
        // select endpoint of *t closer to s.source()
        if (CGAL::collinear_are_ordered_along_line (s.source(), t->source(), t->target()))
            s = K::Segment_2(s.source(), t->source());
        else
            s = K::Segment_2(s.source(), t->target());
    else
        throw runtime_error("Strange segment intersection.");
}

void find_hit(size_t n) {
    // read input
    K::Ray_2 r;
    cin >> r;
    vector<K::Segment_2> segs;
    segs.reserve(n);
    for (size_t i = 0; i < n; ++i) segs.push_back(read_segment());
    random_shuffle(segs.begin(), segs.end());

    // clip the ray at each segment hit (cuts down on the number of
    // intersection points to be constructed: for a uniformly random
    // order of segments, the expected number of constructions is
    // logarithmic in the number of segments that intersect the initial // ray.)
    K::Segment_2 rc(r.source(), r.source());

    // find some segment hit by r
    for (size_t i; i < n; ++i)
        if (CGAL::do_intersect(segs[i], r)) {
            shorten_segment(rc, CGAL::intersection(segs[i], r));
            break;
        }

    if (i == n) { cout << "no\n"; return; }
    // check remaining segments against rc
    while (++i < n)
        if (CGAL::do_intersect(segs[i], rc))
            shorten_segment(rc, CGAL::intersection(segs[i], r)); // not rc!

    cout << floor_to_double(rc.target().x()) << " " << floor_to_double(rc.target().y()) << "\n";

}

int main() {
    // sanity check
    if (numeric_limits<long>::digits < 51)
```

```
78          throw runtime_error("long has <51 bits mantissa");
79      ios_base::sync_with_stdio(false);
80      cout << setiosflags(ios::fixed) << setprecision(0);
81      for (size_t n; cin >> n && n > 0;)
82          find_hit(n);
83      return 0;
84  }
```

Hiking Maps

Keywords— Triangle Contains, Sliding Window

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
#include <vector>
#include <algorithm>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                                          P;
typedef std::vector<P>                                      Pts;
typedef std::vector<int>                                    Covered;
typedef std::vector<Covered>                                Trs;
typedef Covered::const_iterator                             Iterator;

using namespace std;

// Function to check whether a point is inside a triangle
inline bool contains(const Pts& t, const P& p) {
    return !CGAL::right_turn(t[0],t[1],p) && !CGAL::right_turn(t[2],t[3],p) && !CGAL::right_turn(t
        [4],t[5],p);
}


void hiking_maps() {
    int m, n; cin >> m >> n;

    // Read the path
    Pts path(m);
    long x, y;
    for (int i = 0; i < m; i++) {
        cin >> x >> y;
        path[i] = P(x, y);
    }

    // Read the triangles
    Trs triangles(n);
    for (int i = 0; i < n; ++i) {
        Pts t;
        for (std::size_t j = 0; j < 6; ++j) {
            P p;
            std::cin >> p;
            t.push_back(p);
        }
        // Ensure correct order for orientation tests
        for (int j = 0; j < 6; j+=2)
            if (CGAL::right_turn(t[j],t[j+1],t[(j+2)%6])) std::swap(t[j],t[j+1]);
        // Store which path segments are covered
        bool prev = contains(t,path[0]);
        for (int j = 1; j < m; ++j) {
            if (contains(t,path[j])) {
                if (prev) triangles[i].push_back(j-1);
                else prev = true;
            } else
                prev = false;
        }
    }

    // search for the cover by scanning through the sequence of triangles
    Covered covered(m-1,0); // #times i,i+1 is covered
    int uncovered = m-1; // #uncovered segments (covered[i]==0)
    int best = n; // size of best range so far
    for (int tb = 0, te = 0; tb != n;) {
        // Ensure covering
        for (; uncovered > 0 && te != n; te++) {
            for (Iterator j = triangles[te].begin(); j != triangles[te].end(); j++)
                if (++covered[*j] == 1) --uncovered;
        }
        if (uncovered != 0) break;
        // Can we remove tb?
        do {
            for (Iterator j = triangles[tb].begin(); j != triangles[tb].end(); j++)
                if (--covered[*j] == 0) uncovered++;
        } while (++tb != te && uncovered == 0);
        best = min(best, te - tb + 1);
    }
    cout << best << endl;

}

```

```
77 int main() {
78     ios_base::sync_with_stdio(false);
79     int T; cin >> T;
80     for (int t=0; t < T; t++){
81         hiking_maps();
82     }
83 }
```

Antenna

Keywords— Minimum Circle

```

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

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

using namespace std;


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

void compute_antenna_radius(int n) {
    vector<K::Point_2> P(n);
    long x, y;
    for (int i = 0; i < n; i++) {
        cin >> x >> y;
        P[i] = K::Point_2(x, y);
    }

    Min_circle mc(P.begin(), P.end(), true);
    Traits::Circle c = mc.circle();
    cout << ceil_to_double(sqrt(c.squared_radius())) << endl;
}


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

    cout << fixed << setprecision(0);
    while(nb_population > 0) {
        compute_antenna_radius(nb_population);
        cin >> nb_population;
    }
    return 0;
}
```

Attack of the Clones

Keywords— Interval Scheduling

```cpp
/*
The algorithm to solve this problem consist in using the Interval Scheduling to place the maximum number of jedis along the perimeter. The problem begins when the range is continuosly cyncling so there is no reference point to start. The best point to use it, is the one with less jedis covering it (for the first two testsets there is ensure at least one segment without covering and on the last
testset there is at least one with at most 10 jedis). This segment with minimum jedis covering should be found and then apply from there the Interval Scheduling algorithm for all combinations of
starting points (none segment is cutting the starting point, or picking each of the segments that is cutting the starting point and take the best result).
*/
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <climits>

using namespace std;


void compute_jedis() {
    long N, M; cin >> N >> M;

    vector<pair<int, int> > jedis(N);
    // vector<pair<int,int> > nn_jedis, sum_jedis;
    map<int, int> nn_jedis;
    map<int, int> sum_jedis;

    int a, b;
    for (int i = 0; i < N; i++) {
        cin >> a >> b;
        a--; b--;                                       // Re-index to 0-based index
        jedis[i] = make_pair(a, b);
        nn_jedis[a]++;
        if (b < a) nn_jedis[0]++;
        if (b < M - 1) nn_jedis[b+1]--;
    }

    // Find the position which no interval overlaps, or the one that less intervals overlap (at
    // most 10 by statement)
    int prev_sum = 0;
    int min_jedis = INT_MAX, min_jedis_pos = -1;
    for (auto nn_jedi : nn_jedis) {
        int pos = nn_jedi.first;
        sum_jedis[pos] = prev_sum + nn_jedi.second;
        prev_sum += nn_jedi.second;
        // cout << pos << "\t" << prev_sum << endl;
        if (prev_sum < min_jedis) {
            min_jedis = prev_sum;
            min_jedis_pos = pos;
        }
    }

    // Separate the intervals between the ones that cut k_0 and the ones that not.
    vector<pair<int,int> > jedis_cutting, jedis_not_cutting;
    int ai, bi;
    int k0 = min_jedis_pos;
    for (int i = 0; i < N; i++) {
        a = jedis[i].first;
        b = jedis[i].second;
        ai = (a - k0 + M) % M;
        bi = (b - k0 + M) % M;

        if (bi < ai) jedis_cutting.push_back(make_pair(bi, ai));
        else jedis_not_cutting.push_back(make_pair(bi, ai));
    }

    // Sort by ending interval position
    sort(jedis_not_cutting.begin(), jedis_not_cutting.end());

    // Interval Scheduling for the non-cutting-min position intervals
    int result = 0, lastb = -1;
    for (int i = 0; i < jedis_not_cutting.size(); i++) {
        if (jedis_not_cutting[i].second > lastb) {
            result++;
            lastb = jedis_not_cutting[i].first;
```

```
        }
    }

    // Now the same algorithm of Interval Scheduling but for each of the interval that cut the min
    // jedis position
    int result_cutting;
    for (int j = 0; j < jedis_cutting.size(); j++) {
        result_cutting = 1;
        lastb = jedis_cutting[j].first;
        for (int i = 0; i < jedis_not_cutting.size(); i++) {
            if (jedis_not_cutting[i].second > lastb && jedis_not_cutting[i].first < jedis_cutting[
                j].second) {
                result_cutting++;
                lastb = jedis_not_cutting[i].first;
            }
        }
        result = max(result, result_cutting);
    }

    cout << result << endl;
}


int main() {
    ios_base::sync_with_stdio(false);
    int t; cin >> t;

    for (int i = 0; i < t; i++) {
        compute_jedis();
    }
    return 0;
}
```

1.4 BGL

First Steps BGL

Keywords— Minimum Spanning Tree, Dijkstra Shortest Path

```
1
2  // STL includes
3  #include <vector>
4  #include <iostream>
5  #include <algorithm>
6  #include <climits>
7  // BGL includes
8  #include <boost/graph/adjacency_list.hpp>
9  #include <boost/graph/strong_components.hpp>
10 #include <boost/graph/dijkstra_shortest_paths.hpp>
11 #include <boost/graph/prim_minimum_spanning_tree.hpp>
12
13 // Namespaces
14 using namespace std;
15 using namespace boost;
16
17 // Directed graph with integer weights on edges.
18 typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int> > Graph;
19 typedef graph_traits<Graph>::vertex_descriptor  Vertex;
20 typedef graph_traits<Graph>::edge_descriptor    Edge;
21 typedef graph_traits<Graph>::edge_iterator      EdgeIt;
22 // Property map edge -> weight
23 typedef property_map<Graph, edge_weight_t>::type    WeightMap;
24
25
26 void graphs() {
27     int n, m;
28     cin >> n >> m;
29
30     // Initialize graph and weightmap
31     Graph G(n);
32     WeightMap weightmap = get(edge_weight, G);
33
34     // Store the graph
35     for (int i = 0; i < m; i++) {
36         int u, v, w;
37         cin >> u >> v >> w;
38         Edge e; bool success;
39         tie(e, success) = add_edge(v, u, G);
40         weightmap[e] = w;
41     }
42
43     // Find the Minimum Spanning Tree with Prim algorithm
44     vector<int> predmap(n);
45     Vertex start = 0;
46     prim_minimum_spanning_tree(G, make_iterator_property_map(
47         predmap.begin(), get(vertex_index, G)
48     ), root_vertex(start));
49     int w = 0;
50     for (int j = 0; j < n; j++) {
51         Edge e; bool success;
52         tie(e, success) = edge(j, predmap[j], G);
53         if ( success ) {
54             w += weightmap[e];
55         }
56     }
57
58     // Compute shortest t-u path in G
59     vector<int> distmap(n);
60     vector<Vertex> predmap_d(n);
61     dijkstra_shortest_paths(G, start,
62         predecessor_map(make_iterator_property_map(
63             predmap_d.begin(), get(vertex_index, G)
64         )).
65         distance_map(make_iterator_property_map(
66             distmap.begin(), get(vertex_index, G)
67         ))
68     );
69
70     // Find the furthest vertex
71     int max_distance = 0;
72     for (int j = 1; j < n; j++) {
73         if (distmap[j] < INT_MAX) {
74             max_distance = max(max_distance, distmap[j]);
75         }
```

```
    }

    cout << w << " " << max_distance << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        graphs();
    }
}
```

Ant Challenge

Keywords— Minimum Spanning Tree, Dijkstra Shortest Path

```
1
2  // STL includes
3  #include <vector>
4  #include <iostream>
5  #include <algorithm>
6  #include <climits>
7  // BGL includes
8  #include <boost/graph/adjacency_list.hpp>
9  #include <boost/graph/strong_components.hpp>
10 #include <boost/graph/dijkstra_shortest_paths.hpp>
11 #include <boost/graph/prim_minimum_spanning_tree.hpp>
12 #include <boost/graph/graph_utility.hpp>
13
14 // Namespaces
15 using namespace std;
16 using namespace boost;
17
18 // Directed graph with integer weights on edges.
19 typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int> > Graph;
20 typedef graph_traits<Graph>::vertex_descriptor      Vertex;
21 typedef graph_traits<Graph>::edge_descriptor        Edge;
22 typedef graph_traits<Graph>::edge_iterator          EdgeIt;
23 // Property map edge -> weight
24 typedef property_map<Graph, edge_weight_t>::type    WeightMap;
25
26
27 void graphs() {
28     int n, e, s, a, b;
29     cin >> n >> e >> s >> a >> b;
30
31     vector<Graph> Gs(s);
32     vector<WeightMap> weightmaps(s);
33     for (int i = 0; i < s; i++) {
34         weightmaps[i] = get(edge_weight, Gs[i]);
35     }
36
37     for (int i = 0; i < e; i++) {
38         int u, v;
39         cin >> u >> v;
40
41         for (int j = 0; j < s; j++) {
42             int w; cin >> w;
43             Edge edge_to_add; bool success;
44             tie(edge_to_add, success) = add_edge(u, v, Gs[j]);
45             weightmaps[j][edge_to_add] = w;
46         }
47     }
48
49     Graph G;
50     WeightMap weightmap = get(edge_weight, G);
51
52     // For every specie compute the Prim Minimum Spanning Tree and then add the edges into a
       global
53     // graph
54     for (int i = 0; i < s; i++) {
55         int v; cin >> v;
56         vector<Vertex> primpredmap(n);
57         prim_minimum_spanning_tree(Gs[i], make_iterator_property_map(primpredmap.begin(), get(
               vertex_index, Gs[i])));
58
59         Edge edge_s; bool success;
60         Edge new_edge; bool new_success;
61         for (int j = 0; j < n; j++) {
62             if (j == primpredmap[j]) { continue; }
63
64             Vertex source = j, target = primpredmap[j];
65             tie(edge_s, success) = edge(source, target, Gs[i]);
66             int weight = weightmaps[i][edge_s];
67             tie(new_edge, new_success) = add_edge(j, primpredmap[j], G);
68             weightmap[new_edge] = weight;
69
70         }
71     }
72
73     // Compute the dijkstra shortest path over the global graph
74     vector<int> distmap(n);
75     vector<Vertex> predmap(n);
```

```
    Vertex start = a;
    Vertex end = b;
    dijkstra_shortest_paths(G, start,
        predecessor_map(make_iterator_property_map(
            predmap.begin(), get(vertex_index, G)
        )).
        distance_map(make_iterator_property_map(
            distmap.begin(), get(vertex_index, G)
        ))
    );

    cout << distmap[end] << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        graphs();
    }
}
```

Important Bridges

Keywords— Biconnected Components

```
1
2  // STL includes
3  #include <vector>
4  #include <iostream>
5  #include <algorithm>
6  #include <climits>
7  // BGL includes
8  #include <boost/graph/adjacency_list.hpp>
9  #include <boost/graph/strong_components.hpp>
10 #include <boost/graph/dijkstra_shortest_paths.hpp>
11 #include <boost/graph/prim_minimum_spanning_tree.hpp>
12 #include <boost/graph/biconnected_components.hpp>
13
14
15 // Namespaces
16 using namespace std;
17 namespace boost {
18     struct edge_component_t {
19         enum { num = 555 };
20         typedef edge_property_tag kind;
21     } edge_component;
22 }
23
24 using namespace boost;
25
26 // Directed graph with integer weights on edges.
27 typedef adjacency_list<vecS, vecS, undirectedS, no_property, property <edge_component_t, size_t> >
       Graph;
28 typedef graph_traits<Graph>::vertex_descriptor      Vertex;
29 typedef graph_traits<Graph>::edge_descriptor        Edge;
30 typedef graph_traits<Graph>::edge_iterator          EdgeIt;
31
32
33 void important_bridges() {
34     int n, m;
35     cin >> n >> m;
36
37     Graph G;
38     vector<pair<int, int> > bridges(m), critical_bridges;
39     for (int i = 0; i < m; i++) {
40         int u, v;
41         cin >> u >> v;
42         bridges[i] = pair<int, int>(u, v);
43         Edge e; bool success;
44         tie(e, success) = add_edge(u, v, G);
45     }
46
47     property_map<Graph, edge_component_t>::type
48     component = get(edge_component, G);
49
50     size_t num_comps = biconnected_components(G, component);
51
52     // Agrupate all the Edges depending the component ID
53     EdgeIt ei, ei_end;
54     vector<vector<Edge> > edge_components(num_comps, vector<Edge>(0));
55     for (tie(ei, ei_end) = edges(G); ei != ei_end; ++ei) {
56         int comp = component[*ei];
57         edge_components[comp].push_back(*ei);
58     }
59
60     // Store as result all the edges that are alone at its component
61     for (int i = 0; i < num_comps; i++) {
62         if (edge_components[i].size() > 1) {
63             continue;
64         }
65         Vertex s, t;
66         s = source(edge_components[i][0], G);
67         t = target(edge_components[i][0], G);
68         if (s < t) {
69             critical_bridges.push_back(pair<int, int>(s, t));
70         } else {
71             critical_bridges.push_back(pair<int, int>(t, s));
72         }
73     }
74
75     // Print number of critical bridges
76     sort(critical_bridges.begin(), critical_bridges.end());
```

```
    cout << critical_bridges.size() << endl;
    for (int i = 0; i < critical_bridges.size(); i++) {
        cout << critical_bridges[i].first << " " << critical_bridges[i].second << endl;
    }

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        important_bridges();
    }
}
```

Buddy Selection

Keywords— Maximum Cardinality Matching

```
1
2  // STL includes
3  #include <vector>
4  #include <string>
5  #include <iostream>
6  #include <unordered_map>
7  // BGL includes
8  #include <boost/graph/adjacency_list.hpp>
9  #include <boost/graph/max_cardinality_matching.hpp>
10
11 // Namespaces
12 using namespace std;
13 using namespace boost;
14
15 // Directed graph with integer weights on edges.
16 typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int> > Graph;
17 typedef graph_traits<Graph>::vertex_descriptor      Vertex;
18 typedef graph_traits<Graph>::edge_descriptor        Edge;
19 typedef graph_traits<Graph>::edge_iterator          EdgeIt;
20 // Property map edge -> weight
21 typedef property_map<Graph, edge_weight_t>::type    WeightMap;
22
23
24 void buddy_selection() {
25     int n, c, f;
26     cin >> n >> c >> f;
27
28     vector<vector<int> > adjacency_list(n, vector<int>(n, 0));
29
30     /* Create the graph with all the edges that have higher weight than f and perform maximum
31     cardinality matching to find a more optimal solution of the problem. If cardinality matching
           is
32     equal to n/2 it means that the solution propoesed ‘f‘ is not the optimal */
33     std::unordered_map<string, vector<int> > characteristics;
34     string name;
35     for (int i = 0; i < n; i++) {
36         for (int j = 0; j < c; j++) {
37             cin >> name;
38             characteristics[name].push_back(i);
39         }
40     }
41
42     // Iterate over all characteristics and create vertex with weight
43     for (auto it = characteristics.begin(); it != characteristics.end(); it++) {
44         vector<int> people = it->second;
45         int np = people.size(); // Number of people that share the same caracteristic
46         if (np <= 1) continue;
47
48         // Lets iterate over all people that share the same characteristic
49         for (int i = 0; i < np; i++) {
50             for (int j = i+1; j < np; j++) {
51                 int p_i = people[i], p_j = people[j];
52                 adjacency_list[p_i][p_j]++;
53                 adjacency_list[p_j][p_i]++;
54             }
55         }
56     }
57
58     // Create final graph
59     Graph G(n);
60     WeightMap weightmap = get(edge_weight, G);
61
62     // Now iterate over all edges and substract the edges with a weight less than ‘f‘
63     for (int i = 0; i < n; i++) {
64         for (int j = i + 1; j < n; j++) {
65             int weight = adjacency_list[i][j];
66             if (weight > f) {
67                 Edge edg; bool success;
68                 tie(edg, success) = add_edge(i, j, G);
69                 weightmap[edg] = weight - f;
70             }
71         }
72     }
73
74     // Compute Matching
75     vector<Vertex> matemap(n); // Use as an Exterior Property Map: Vertex -> Matc
76     edmonds_maximum_cardinality_matching (
```

```
        G,
        make_iterator_property_map(matemap.begin(), get(vertex_index , G))
    );
    // Look at the matching size
    int matchingsize = matching_size (
        G,
        make_iterator_property_map (matemap.begin(), get(vertex_index , G))
    );

    if (matchingsize == n/2) {
        cout << "not optimal" << endl;
    } else {
        cout << "optimal" << endl;
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        buddy_selection();
    }
}
```

TheeV

Keywords— Binary Search, Minimum Circle

```

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

typedef CGAL::Exact_predicates_exact_constructions_kernel   K;
typedef K::Point_2                                          P;
typedef CGAL::Min_circle_2_traits_2<K>                      Traits;
typedef CGAL::Min_circle_2<Traits>                          Min_circle;

using namespace std;


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


pair<bool, K::FT> evaluate(int index, vector<K::FT>& distances, vector<P>& cities) {

    // Initialize the external circle with the furthest cities from 0 to index.
    Min_circle min_ex_circle(cities.begin(), cities.begin() + index + 1, true);
    Traits::Circle circle = min_ex_circle.circle();

    K::FT ex_radius = circle.squared_radius();
    K::FT tc_radius;
    if (index >= cities.size() - 1) tc_radius = 0;
    else tc_radius = distances[index + 1];

    bool eval = ex_radius >= tc_radius;
    K::FT max_radius = max(ex_radius, tc_radius);

    return make_pair(eval, max_radius);
}

int binary_search(int left, int right, vector<K::FT>& distances, vector<P>& cities) {
    if (left == right - 1) return left;
    int m = (right + left) / 2;

    K::FT max_rad; bool eval;
    tie(eval, max_rad) = evaluate(m, distances, cities);
    if (!eval) {
        return binary_search(m, right, distances, cities);
    } else {
        return binary_search(left, m, distances, cities);
    }
}

void theev() {
    int n; cin >> n;

    // Read cities locations
    long x, y; cin >> x >> y;
    P their_city(x, y);
    P e_their_city(x, y);

    vector<P> cities(n-1);
    for (int i = 0; i < n-1; i++) {
        cin >> x >> y;
        cities[i] = P(x, y);
    }

    // For the cas we have one or two cities where the min radius will be 0
    if (n <= 2) {
        cout << 0 << endl;
        return;
    }

    // Sort all the cities depending on the distance to the capital with decreasing order
    sort(cities.begin(), cities.end(),
        [&their_city](P a, P b) {
            return CGAL::squared_distance(a, their_city) > CGAL::squared_distance(b, their_city);
```

```
        }
    );

    // Compute and store the distances to the capital
    vector<K::FT> distances(n-1);
    for (int i = 0; i < n - 1; i++) {
        distances[i] = squared_distance(cities[i], their_city);
    }

    // Perform binary search
    int index = binary_search(0, n-1, distances, cities);

    K::FT rad_1 = evaluate(index, distances, cities).second;
    K::FT rad_2 = evaluate(index + 1, distances, cities).second;

    K::FT result = min(rad_1, rad_2);

    cout << ceil_to_double(result) << endl;
}

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

    cout << fixed << setprecision(0);
    for (int i = 0; i < t; i++) {
        theev();
    }
    return 0;
}
```

Chapter 2

Advanced Algorithms

2.1 Dynamic Programming, Brute Force, Split and List

Burning Coins

Keywords— Dynamic Programming

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

using namespace std;


int play(vector<int>& coins_values, int left_idx, int right_idx, bool turn, vector<vector<int> >& mem) {
    /*  turn = true  -> Your turn
        turn = false -> Friend's turn */

    // Final case if no coins left
    if (left_idx == right_idx) {
        return 0;
    }

    if (mem[left_idx][right_idx-1] > -1) {
        return mem[left_idx][right_idx-1];
    }

    // First take the left coins then the right one
    int left_result, right_result, result;
    if (turn) {
        left_result = coins_values[left_idx];
        left_result += play(coins_values, left_idx+1, right_idx, false, mem);
        right_result = coins_values[right_idx-1];
        right_result += play(coins_values, left_idx, right_idx-1, false, mem);
        result = (left_result > right_result) ? left_result : right_result;
        mem[left_idx][right_idx-1] = result;
        return result;
    } else {
        left_result = play(coins_values, left_idx+1, right_idx, true, mem);
        right_result = play(coins_values, left_idx, right_idx-1, true, mem);
        result = (left_result < right_result) ? left_result: right_result;
        mem[left_idx][right_idx-1] = result;
        return result;
    }
}


void burning_coins() {
    int n; cin >> n;

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

    // Add some memory storing the result for every pair of left and right indexes possible (speedup)
    vector<vector<int> > mem(n, vector<int>(n, -1));
    cout << play(coins_values, 0, n, true, mem) << endl;


}

```

```
int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        burning_coins();
    }
}
```

Light Pattern

Keywords— Dynamic Programming

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

using namespace std;


void light_pattern() {
    int n, k, x;
    cin >> n >> k >> x;


    int p, s, b;            // p: number of patterns,
    uint16_t q = 0x00;      // q: desired pattern
    p = n / k;

    // Compute the desired pattern
    for (int i = 1; i <= k; i++) {
        q |= (x & 1)<<(k-i);
        x >>= 1;
    }

    vector<uint16_t> bulbs_state(p);
    for (int i = 0; i < p; i++) {
        s = 0;              // s to store the bulb state
        for (int j = 0; j < k; j++) {
            cin >> b;
            if (b == 1) s |= 1 << j;
        }
        bulbs_state[i] = s;
    }

    // Compute the diff state of each pattern with the desired one. The int represent the number
        of
    // different bulbs at each of the patterns.
    vector<int> diff_state(p, 0);
    for (int i = 0; i < p; i++) {
        int diff = bulbs_state[i] ^ q;
        for (int j = 0; j < k; j++) {
            if ((diff & 1<<j) != 0) diff_state[i]++;
        }
    }

    // Compute the accumulated in case the bulbs have been changed one by one or by block all the
    // previous ones
    vector<int> values_individual(p), values_block(p);
    values_individual[0] = diff_state[0];
    values_block[0] = k - diff_state[0] + 1;
    for (int i = 1; i < p; i++) {
        values_individual[i] = min(values_individual[i-1], values_block[i-1]) + diff_state[i];
        values_block[i] = min(values_block[i-1] + k - diff_state[i],
                              values_individual[i-1] + k - diff_state[i] + 2);
    }

    cout << min(values_individual[p-1], values_block[p-1]) << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        light_pattern();
    }
}
```

Light at the Museum

Keywords— Dynamic Programming, Split and List

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

using namespace std;

typedef pair<int, int>                      pair_int;
typedef vector<vector<int> >                vecvec;
typedef vector<pair<vector<int>,int> >      vecpairvec;


void create_subset(vecvec& state_on, vecvec& state_off, vecpairvec& F, int dbound, int ubound) {
    int N = ubound - dbound;
    int M = state_on[0].size();
    for (int k = 0; k < 1<<N; k++) {
        vector<int> tuple(M, 0);
        int count = 0;

        for (int i = 0; i < N; i++) {
            bool changed_state = (k & 1<<i) != 0;
            if (changed_state) count++;

            for (int j = 0; j < M; j++)
                tuple[j] += (changed_state) ? state_off[i+dbound][j] : state_on[i+dbound][j];
        }
        F.push_back(make_pair(tuple, count));
    }
}

struct Comp {
    bool operator() (pair<vector<int>, int> P, vector<int> V) {
        return P.first < V;
    }
    bool operator() (vector<int> V, pair<vector<int>, int> P) {
        return V < P.first;
    }
};

void light_at_the_museum() {
    int N, M; cin >> N >> M;

    vector<int> brightness(M);
    for (int i = 0; i < M; i++) cin >> brightness[i];

    vector<vector<int> > state_on(N, vector<int>(M)), state_off(N, vector<int>(M));
    for (int j = 0; j < N; j++)
        for (int i = 0; i < M; i++) cin >> state_on[j][i] >> state_off[j][i];


    // Create all combinations for two subsets (divide and list)
    vecpairvec F1, F2;
    create_subset(state_on, state_off, F1, 0, N/2);
    create_subset(state_on, state_off, F2, N/2, N);
    sort(F2.begin(), F2.end());

    int min_changes = INT_MAX;
    for (int idx = 0; idx < F1.size(); idx++) {
        vector<int> missing = F1[idx].first;
        for (int i = 0; i < M; i++) {
            missing[i] = brightness[i] - missing[i];
        }

        pair<vecpairvec::iterator, vecpairvec::iterator>  it_pair;
        it_pair = equal_range(F2.begin(), F2.end(), missing, Comp());

        for (auto it = it_pair.first; it != it_pair.second; it++) {
            int count = it->second + F1[idx].second;
            min_changes = min(min_changes, count);
        }
    }

    if (min_changes == INT_MAX) cout << "impossible\n";
    else cout << min_changes << endl;
}

int main() {
```

```
78      ios_base::sync_with_stdio(false);
79      int T; cin >> T;
80      for (int t=0; t < T; t++){
81          light_at_the_museum();
82      }
83  }
```

The Great Game

Keywords— Dynamic Programming

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

using namespace std;


int max_(vector<vector<int> > &transitions, int start, int end,
    vector<int> &min_mem, vector<int> &max_mem);

int min_(vector<vector<int> > &transitions, int start, int end,
        vector<int> &min_mem, vector<int> &max_mem) {

    if (start == end) {
        min_mem[start] = 0;
        return 0;
    }

    int min_dist = INT_MAX;
    for (int v : transitions[start]) {
        if (max_mem[v] == -1) {
            max_(transitions, v, end, min_mem, max_mem);
        }
        min_dist = min(min_dist, max_mem[v]);
    }
    min_mem[start] = min_dist + 1;
    return min_mem[start];
}


int max_(vector<vector<int> > &transitions, int start, int end,
    vector<int> &min_mem, vector<int> &max_mem) {

    if (start == end) {
        max_mem[start] = 0;
        return 0;
    }

    int max_dist = 0;
    for (int v : transitions[start]) {
        if (min_mem[v] == -1) {
            min_(transitions, v, end, min_mem, max_mem);
        }
        max_dist = max(max_dist, min_mem[v]);
    }
    max_mem[start] = max_dist + 1;
    return max_mem[start];
}

void great_game() {
    int n, m;
    cin >> n >> m;
    int r, b;
    cin >> r >> b;
    // set to 0-index
    r--; b--;

    vector<vector<int> > transitions(n);
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        // set to 0-index
        u--; v--;
        transitions[u].push_back(v);
    }

    // Create memory vectors
    vector<int> min_mem(n, -1), max_mem(n, -1);

    // For each of the starting possitions, check which one has minimum movements to win the game.
    int red_moves = min_(transitions, r, n-1, min_mem, max_mem);
    int black_moves = min_(transitions, b, n-1, min_mem, max_mem);

    cout << (black_moves < red_moves || (black_moves == red_moves && black_moves % 2 == 0)) <<
        endl;
}

```

```
77 int main() {
78     ios_base::sync_with_stdio(false);
79     int T; cin >> T;
80     for (int t=0; t < T; t++){
81         great_game();
82     }
83 }
```

On Her Majesty's Secret Service

Keywords— Maximum Cardinality Matching, Dijkstra Shortest Path

```

// STL includes
#include <vector>
#include <string>
#include <iostream>
#include <unordered_map>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

// Namespaces
using namespace std;
using namespace boost;

// Directed graph with integer weights on edges.
typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int> > Graph;
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_weight_t, int> > DiGraph;
typedef graph_traits<DiGraph>::vertex_descriptor      Vertex;
typedef graph_traits<DiGraph>::edge_descriptor        Edge;
typedef graph_traits<DiGraph>::edge_iterator          EdgeIt;
// Property map edge -> weight
typedef property_map<DiGraph, edge_weight_t>::type    WeightMap;


void oh_her_majestys_secret_service() {
    int n, m, a, s, c, d;
    cin >> n >> m >> a >> s >> c >> d;

    DiGraph G(n);
    WeightMap weightmap = get(edge_weight, G);

    char w; int x, y, z;
    Edge edg; bool success;
    for (int i = 0; i < m; i++) {
        cin >> w >> x >> y >> z;

        tie(edg, success) = add_edge(x, y, G);
        weightmap[edg] = z;
        if (w == 'L') {
            tie(edg, success) = add_edge(y, x, G);
            weightmap[edg] = z;
        }
    }

    vector<int> agents(a), shelters(s);
    for (int i = 0; i < a; i++) cin >> agents[i];
    for (int i = 0; i < s; i++) cin >> shelters[i];

    // Compute one distance map per agent
    vector<vector<int> > distmap(a, vector<int>(n));
    for (int i = 0; i < a; i++) {
        dijkstra_shortest_paths(G, agents[i],
            distance_map(make_iterator_property_map(distmap[i].begin(), get(vertex_index, G))));
    }
    // Represent G' as the pairwise distance matrix T from agents to shelters.
    vector<vector<int> > T(a, vector<int>(s, INT_MAX));
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < s; j++) {
            T[i][j] = distmap[i][shelters[j]];
        }
    }

    // Binary search for the smallest t
    int low = 0, high = INT_MAX;
    while (low < high) {
        int mid = low + (high-low)/2;
        // Regresent the model as a bipartite graph with all the agents as nodes in one side, and
        // all the shelters duplicated for each possible capacity and then compute the maximum
        // cardinality matching
        Graph GG(a + s*c);
        for (int i = 0; i < a; i++) {
            for (int j = 0; j < s; j++) {
                if (T[i][j] == INT_MAX) continue;
                for (int k = 0; k < c; k++) {
                    if (T[i][j] + (k + 1) * d <= mid) {
                        add_edge(i, a + k * s + j, GG);
```

```
                    }
                }
            }
        }
        // Compute maximum cardinality
        vector<Vertex> matemap(a + s * c); // Use as an Exterior Property Map: Vertex -> Matc
        edmonds_maximum_cardinality_matching (
            GG,
            make_iterator_property_map(matemap.begin(), get(vertex_index , G))
        );
        const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
        int matchingsize = 0;
        for (int i = 0; i < a; i++) {
            matchingsize += (matemap[i] != NULL_VERTEX);
        }

        if (matchingsize == a) high = mid;
        else low = mid + 1;

    }
    cout << low << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        oh_her_majestys_secret_service();
    }
}
```

Poker Chips

Keywords— Dynamic Programming

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

using namespace std;

double compute_award(double k) {
    if (k > 1) {
        return exp2(k-2);
    } else if (k == 1) {
        return 0;
    } else {
        throw runtime_error("k can not be less than 1.");
    }
}

int map_positions_into_vector(vector<int>& stack_heigh, vector<int>& stack_top) {
    int nb_stacks = stack_heigh.size();

    int accum = 1, pos = 0;
    for (int i = nb_stacks - 1; i >= 0; i--) {
        pos += accum * stack_heigh[i];
        accum *= stack_top[i] + 1;
    }
    return pos;
}

int play(vector<vector<int> >& stacks, vector<int>& nb_chips, vector<int>& turn_positions, vector<
    int>& memory) {

    // Check memory first
    int mem_pos = map_positions_into_vector(turn_positions, nb_chips);
    if (memory[mem_pos] != -1) return memory[mem_pos];

    // Check if there is not more chips at the stack (it is not necessary as memory[0] = 0)

    int nb_stacks = stacks.size();
    map<int, vector<int> > color_map;
    // Check at each of the stacks and see the next poker chips and group them by color
    for (int i = 0; i < nb_stacks; i++) {
        int next_position = turn_positions[i] - 1;
        if (next_position < 0) continue;    // In case a stack is empty
        color_map[stacks[i][next_position]].push_back(i);
    }

    // Iterate over all groups of colors, and compute the maximum award in DP achievable from this
    // stack positions
    int max_award = 0;
    for (auto it = color_map.begin(); it != color_map.end(); it++) {
        vector<int> chips_positions = it->second;
        int nb = chips_positions.size();

        /* Now compute all the possible combinations of extracting these chips (extract all the
        ones with the same colos is not allways the best strategy)
        Example:
        3 3 3 2 2 2 2
        2 2 2 X X 3 X

        If use the strategy to extract all the chips from the same color award = 10
        but if first is removed the 2 with a 3 at the bottom, then the award = 20
        */
        for (int k = 1; k < 1<<nb; k++) {
            int count = 0;

            // Selecting the chips from the same color, see which is the award with the next
                position
            vector<int> new_positions(turn_positions);

            for (int i = 0; i < nb; i++) {
                if (k & (1<<i)) {
                    count++;
                    new_positions[chips_positions[i]]--;
                }
            }
```

```
76
77              int new_award = compute_award(count);
78
79              int award = play(stacks, nb_chips, new_positions, memory);
80              max_award = max(max_award, award + new_award);
81          }
82      }
83
84      // Store result into memory
85      memory[mem_pos] = max_award;
86
87      return max_award;
88  }
89
90  void poker_chips() {
91      int n; cin >> n;
92
93      vector<int> nb_chips(n);
94      vector<vector<int> > stacks(n);
95      vector<int> initial_positions(n);
96
97      for (int i = 0; i < n; i++) {
98          cin >> nb_chips[i];
99      }
100     for (int i = 0; i < n; i++) {
101         for (int j = 0; j < nb_chips[i]; j++) {
102             int c; cin >> c;
103             stacks[i].push_back(c);
104         }
105         initial_positions[i] = nb_chips[i];
106     }
107
108     int N_memory = map_positions_into_vector(nb_chips, nb_chips) + 1;
109     vector<int> memory(N_memory, -1);
110     memory[0] = 0;
111
112     int max_award = play(stacks, nb_chips, initial_positions, memory);
113     cout << max_award << endl;
114
115 }
116
117 int main() {
118     ios_base::sync_with_stdio(false);
119     int T; cin >> T;
120     for (int t=0; t < T; t++){
121         poker_chips();
122     }
123 }
```

2.2 BGL Flows

Coin Tossing

Keywords— Maximum Flow

```
#include <iostream>
#include <vector>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};


void coin_tossing() {
    int n, m;
    cin >> n >> m;

    Graph G;
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;

        if (c == 1) {
            eaG.addEdge(i, m+a, 1);
        } else if (c == 2) {
            eaG.addEdge(i, m+b, 1);
        } else {
```

```
            eaG.addEdge(i, m+a, 1);
            eaG.addEdge(i, m+b, 1);
        }

    }

    Vertex src = add_vertex(G);
    for (int i = 0; i < m; i++) {
        eaG.addEdge(src, i, 1);
    }
    Vertex sink = add_vertex(G);
    long sum = 0;
    for (int j = m; j < m+n; j++) {
        int s; cin >> s;
        sum += s;
        eaG.addEdge(j, sink, s);
    }

    long flow = push_relabel_max_flow(G, src, sink);

    if (flow != m || flow != sum) {
        cout << "no" << endl;
    } else {
        cout << "yes" << endl;
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        coin_tossing();
    }
}
```

Shopping Trip

Keywords— Maximum Flow

```

#include <iostream>
#include <vector>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};


void shopping() {
    int n, m, s;
    cin >> n >> m >> s;

    Graph G;
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    vector<int> stores_locations(s);
    for (int i = 0; i < s; i++) {
        cin >> stores_locations[i];
    }

    // Add street edges
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        eaG.addEdge(a, b, 1);
        eaG.addEdge(b, a, 1);
    }

```

```
    // Add edges from stores to sink
    // Vertex src = add_vertex(G);
    Vertex sink = add_vertex(G);
    // eaG.addEdge(src, 0, s);
    for (int i = 0; i < s; i++) {
        eaG.addEdge(stores_locations[i], sink, 1);
    }

    long flow = push_relabel_max_flow(G, 0, sink);

    if (flow == s) {
        cout << "yes" << endl;
    } else {
        cout << "no" << endl;
    }

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        shopping();
    }
}
```

Kingdom Defence

Keywords— Maximum Flow

```

#include <iostream>
#include <map>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::edge_iterator              EdgeIt;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    Edge addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
        return e;
    }
};


void kingdom_defence() {
    int l, p;
    cin >> l >> p;

    Graph G(l+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    // ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    Vertex src = vertex(l, G);
    Vertex sink = vertex(l+1, G);

    long total_defenders = 0;
    long total_available = 0;
    for (int i = 0; i < l; i++) {
        long g, d;
        cin >> g >> d;
        Vertex location = vertex(i, G);
        eaG.addEdge(src, location, g);
```

```
        eaG.addEdge(location, sink, d);
        total_defenders += d;
        total_available += g;
    }

    for (int i = 0; i < p; i++) {
        int f, t;
        long c, C;
        cin >> f >> t >> c >> C;
        eaG.addEdge(f, t, C - c);
        if (c > 0) {
            eaG.addEdge(f, sink, c);
            total_defenders += c;
            eaG.addEdge(src, t, c);
            total_available += c;
        }
    }

    long flow = push_relabel_max_flow(G, src, sink);

    bool result = flow == total_defenders;

    if (result) {
        cout << "yes" << endl;
    } else {
        cout << "no" << endl;
    }

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        kingdom_defence();
    }
}
```

A New Hope

Keywords— Dynamic Programming

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

using namespace std;

struct center {
    center (int n) {
        sup = vector<uint16_t>(n, 0);
    }
    vector<uint16_t> sup;
    map<uint16_t, map<int, int> > extsup;
    map<int, int> bestmap;
};


int best_stormtroopers(vector<center>& centers, int c, int mask, const int S) {
    auto bestit = centers[c].bestmap.find(mask);
    if (bestit != centers[c].bestmap.end())
        return bestit->second;

    int best = 0;
    for (int k = 0; k < (0x1<<S); k++) {
        // Mask refers to the stormtroopers that can not be choosen because are being supervised
        if ((k & mask) != 0) continue;

        bool valid = true;
        // Check the validity looking if any of the stormtroopers supervise each other in this
        // configurations
        for (int i = 0; i < S && valid; i++) {
            if (((k & (0x1<<i)) != 0) && ((k & centers[c].sup[i]) != 0)) valid = false;
        }
        if (!valid) continue;

        int count = 0;
        // Count the strompers being sabotaged
        for (int i = 0; i < S; i++) {
            if (k & (0x1<<i)) count++;
        }
        // Now compute the mask (the stormtroopers that on the children centers are being
        // supervised by this configuration) and with DP count the maximum number of stormtroopers
        // that are not being supervised by each other.
        for (auto& esup : centers[c].extsup) {
            int nmask = 0;
            for (auto& p : esup.second) {
                if (k & 0x1<<p.first) nmask |= p.second;
            }
            count += best_stormtroopers(centers, esup.first, nmask, S);
        }
        best = max(best, count);
    }
    centers[c].bestmap[mask] = best;
    return best;
}

void a_new_hope() {
    int K, S, M;
    cin >> K >> S >> M;

    vector<center> centers(K, center(S));

    int u, v, h, x, y;
    for (int i = 0; i < M; i++) {
        cin >> u >> v >> h;
        if (u == v) {
            for (int j = 0; j < h; j++) {
                cin >> x >> y;
                centers[u].sup[x] |= 0x1 << y;
            }
        }
        else {
            map<int, int> esup;
            for (int j = 0; j < h; j++) {
                cin >> x >> y;
                esup[x] |= 0x1 << y;
```

```
            }
            centers[u].extsup[v] = esup;
        }

    }

    cout << best_stormtroopers(centers, 0, 0, S) << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        a_new_hope();
    }
}
```

2.3 Linear/Quadratic Programing

What is the maximum?

Keywords— LP/QP

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// choose exact integral type
typedef CGAL::Gmpz ET;

// program and solution types
typedef CGAL::Quadratic_program<long> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

void solve_1(int a, int b){
    Program qp (CGAL::SMALLER, true, 0, false, 0);
    const int X = 0;
    const int Y = 1;
    qp.set_a(X, 0, 1); qp.set_a(Y, 0, 1); qp.set_b(0, 4);
    qp.set_a(X, 1, 4); qp.set_a(Y, 1, 2); qp.set_b(1, a*b);
    qp.set_a(X, 2, -1); qp.set_a(Y, 2, 1); qp.set_b(2, 1);
    qp.set_c(Y, -b);
    qp.set_d(X, X, 2*a);

    Solution s = CGAL::solve_quadratic_program(qp, ET());
    assert (s.solves_quadratic_program(qp));
    // cout << s << endl;

    if (s.is_infeasible()) {
        cout << "no" << endl;
    } else if (s.is_unbounded()) {
        cout << "unbounded" << endl;
    } else {
        assert (s.is_optimal());
        long result = floor(-CGAL::to_double(s.objective_value()));
        cout << result << endl;
    }
}

void solve_2(int a, int b){
    Program qp (CGAL::LARGER, false, 0, true, 0);
    const int X = 0;
    const int Y = 1;
    const int Z2 = 2;
    qp.set_a(X, 0, 1); qp.set_a(Y, 0, 1); qp.set_b(0, -4);
    qp.set_a(X, 1, 4); qp.set_a(Y, 1, 2); qp.set_a(Z2, 1, 1); qp.set_b(1, -a*b);
    qp.set_a(X, 2, -1); qp.set_a(Y, 2, 1); qp.set_b(2, -1);
    qp.set_u(Z2, false, 0);
    qp.set_d(X, X, 2*a);
    qp.set_c(Y, b);
    qp.set_d(Z2, Z2, 2*1);

    Solution s = CGAL::solve_quadratic_program(qp, ET());
    assert (s.solves_quadratic_program(qp));
    // cout << s << endl;

    if (s.is_infeasible()) {
        cout << "no" << endl;
    } else if (s.is_unbounded()) {
        cout << "unbounded" << endl;
    } else if (s.is_optimal()){
        long result = ceil(CGAL::to_double(s.objective_value()));
        cout << result << endl;
    } else {
        cout << "no" << endl;
    }
}


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

```

```
76      while (p > 0) {
77          int a, b;
78          cin >> a >> b;
79          if (p == 1) solve_1(a, b);
80          else if (p == 2) solve_2(a, b);
81          cin >> p;
82      }
83  }
```

Diet

Keywords— LP/QP

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpzf.h>

// choose exact integral type
typedef CGAL::Gmpzf ET;

// program and solution types
typedef CGAL::Quadratic_program<long> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

void diet(int n, int m){
    Program lp (CGAL::LARGER, true, 0, false, 0);


    for (int i = 0; i < n; i++) {
        long c_min, c_max;
        cin >> c_min >> c_max;

        lp.set_b(i*2, c_min);
        lp.set_b(i*2+1, c_max);
        lp.set_r(i*2+1, CGAL::SMALLER);
    }

    for (int j = 0; j < m; j++) {
        long p; cin >> p;
        lp.set_c(j, p);
        for (int i = 0; i < n; i++) {
            long C; cin >> C;
            lp.set_a(j, i*2, C);
            lp.set_a(j, i*2+1, C);
        }
    }

    Solution s = CGAL::solve_linear_program(lp, ET());
    assert (s.solves_linear_program(lp));

    if (s.is_infeasible() || s.is_unbounded()) {
        cout << "No such diet." << endl;
    } else {
        long result = floor(CGAL::to_double(s.objective_value()));
        cout << result << endl;
    }
}


int main() {
    int n, m;
    cin >> n >> m;
    while (!(n==0 && m==0)) {
        diet(n, m);
        cin >> n >> m;
    }
}
```

Portfolios

Keywords— LP/QP

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpzf.h>

// choose exact integral type
typedef CGAL::Gmpzf ET;

// program and solution types
typedef CGAL::Quadratic_program<long> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

bool solve_case(vector<long>& c, vector<long>& r, vector<vector<long> >& v, long C, long R, long V
    , int n) {
    Program qp (CGAL::LARGER, true, 0, false, 0);
    for (int i = 0; i < n; i++) {
        qp.set_a(i, 0, r[i]);
        qp.set_a(i, 1, c[i]);
    }
    qp.set_r(1, CGAL::SMALLER);
    qp.set_b(0, R);
    qp.set_b(1, C);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            qp.set_d(i, j, 2*v[i][j]);
        }
    }

    Solution s = CGAL::solve_quadratic_program(qp, ET());
    assert (s.solves_quadratic_program(qp));

    if (s.is_infeasible() || s.is_unbounded()) {
        return false;
    } else if (CGAL::to_double(s.objective_value()) <= V) {
        return true;
    } else {
        return false;
    }

}


void portfolios(int n, int m){
    vector<long> c(n), r(n);
    vector<vector<long> > v(n, vector<long>(n));

    for (int i = 0; i < n; i++) {
        cin >> c[i] >> r[i];
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> v[i][j];
        }
    }
    for (int i = 0; i < m; i++) {
        long C, V, R;
        cin >> C >> R >> V;
        bool result = solve_case(c, r, v, C, R, V, n);
        if (result) {
            cout << "Yes." << endl;
        } else {
            cout << "No." << endl;
        }
    }


}


int main() {
    int n, m;
```

```
77      while (true) {
78          cin >> n >> m;
79          if (n == 0 && m == 0) { break; }
80          portfolios(n, m);
81      }
82  }
```

Inball

Keywords— LP/QP

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// choose exact integral type
typedef CGAL::Gmpz ET;

// program and solution types
typedef CGAL::Quadratic_program<long> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;


void inball(int n){
    int d; cin >> d;

    Program lp (CGAL::SMALLER, false, 0, false, 0);
    lp.set_c(d, -1);
    lp.set_l(d, true, 0);
    for (int i = 0; i < n; i++) {
        long norm = 0;
        for (int j = 0; j < d; j++) {
            long a; cin >> a;
            lp.set_a(j, i, a);
            norm += a * a;
        }
        lp.set_a(d, i, sqrt(norm));
        long b; cin >> b;
        lp.set_b(i, b);
    }

    Solution s = CGAL::solve_linear_program(lp, ET());
    assert (s.solves_linear_program(lp));

    if (s.is_infeasible()) {
        cout << "none" << endl;
    } else if (s.is_unbounded()) {
        cout << "inf" << endl;
    } else {
        long result = -CGAL::to_double(s.objective_value());
        cout << result << endl;
    }
}


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

Knights

Keywords— Maximum Flow

```

#include <iostream>
#include <map>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::edge_iterator              EdgeIt;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    Edge addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
        return e;
    }
};


void knights() {
    int m, n, k, c;
    cin >> m >> n >> k >> c;

    Graph G(2*m*n+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    // Create graph representing the cave
    Vertex src = vertex(2*n*m, G);
    Vertex sink = vertex(2*n*m+1, G);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            int pos_out = i*m + j;
            int pos_in = pos_out + n*m;
            int pos_out_right = pos_out + 1;
            int pos_in_right = pos_out_right + n*m;
            int pos_out_down = pos_out + m;
            int pos_in_down = pos_out_down + n*m;
```

```
            // Conect inner node with outer node
            eaG.addEdge(pos_in, pos_out, c);
            if (j < m-1) {
                // Connecting node with the one on the right
                eaG.addEdge(pos_out, pos_in_right, 1);
                eaG.addEdge(pos_out_right, pos_in, 1);
            }
            if (i < n-1) {
                // Connecting node with the one on the bottom
                eaG.addEdge(pos_out, pos_in_down, 1);
                eaG.addEdge(pos_out_down, pos_in, 1);
            }
            if (i == 0 || i == n-1) {
                // Connecting top and bottom nodes to sink
                eaG.addEdge(pos_out, sink, 1);
            }
            if (j == 0 || j == m-1) {
                // Connecting left and right nodes to sink
                eaG.addEdge(pos_out, sink, 1);
            }
        }
    }

    // Add edge from the source to the intersection representing the starting point of the knight
    for (int i = 0; i < k; i++) {
        int x, y; cin >> x >> y;
        int pos = y*m+x + n*m;
        eaG.addEdge(src, pos, 1);
    }

    // Compute the maximum number of knights as maximum flow
    long flow = push_relabel_max_flow(G, src, sink);
    cout << flow << endl;

}

int main() {
    int T; cin >> T;
    for (int t=0; t < T; t++){
        knights();
    }
}
```

2.4 Proximity Structures in CGAL

Graypes

Keywords— Delaunay Triangulation

```cpp
#include <vector>

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>                   Triangulation;
typedef Triangulation::Finite_faces_iterator                Face_iterator;
typedef Triangulation::Edge_iterator                        EdgeIt;

using namespace std;


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

void graypes(int n) {
    vector<K::Point_2> p(n);
    Triangulation t;

    for (int i = 0; i < n; i++) {
        long x, y;
        cin >> x >> y;
        p[i] = K::Point_2(x, y);
    }
    t.insert(p.begin(), p.end());

    K::FT min_dist = t.segment(t.finite_edges_begin()).squared_length();
    for (EdgeIt e = t.finite_edges_begin(); e != t.finite_edges_end(); e++) {
        min_dist = min(min_dist, t.segment(e).squared_length());
    }

    K::FT d = sqrt(min_dist * (100 * 100) / (2 * 2));

    cout << ceil_to_double(d) << endl;
}


int main() {
    int n;
    cout << fixed << setprecision(0);
    while (true) {
        cin >> n;
        if (n == 0) { break; }
        graypes(n);
    }
}
```

Bistro

Keywords— Delaunay Triangulation

```cpp
#include <vector>

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>                   Triangulation;
typedef Triangulation::Finite_faces_iterator                Face_iterator;
typedef Triangulation::Edge_iterator                        EdgeIt;
typedef K::Point_2                                          Point;

using namespace std;


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

void bistro(int n) {
    int m;
    vector<Point> r(n);
    Triangulation t;

    for (int i = 0; i < n; i++) {
        long x, y;
        cin >> x >> y;
        r[i] = Point(x, y);
    }
    t.insert(r.begin(), r.end());

    cin >> m;

    for (int i = 0; i < m; i++) {
        long x, y;
        cin >> x >> y;
        Point p = Point(x,y);
        K::FT dist = CGAL::squared_distance(p, t.nearest_vertex(p)->point());
        cout << ceil_to_double(dist) << endl;
    }

}


int main() {
    int n;
    cout << fixed << setprecision(0);
    while (true) {
        cin >> n;
        if (n == 0) { break; }
        bistro(n);
    }
}
```

H1N1

Keywords— Delaunay Triangulation

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

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>                   Triangulation;
typedef Triangulation::Finite_faces_iterator                FaceIt;
typedef Triangulation::Edge_iterator                        EdgeIt;
typedef Triangulation::Face_handle                          FaceH;
typedef Triangulation::Vertex_handle                        VertexH;
typedef K::Point_2                                          Point;
typedef K::Segment_2                                        Segment;

using namespace std;


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

void h1n1(int n) {

    vector<Point> infected(n);
    Triangulation t;
    int m;

    for (int i = 0; i < n; i++) {
        long x, y;
        cin >> x >> y;
        infected[i] = Point(x, y);
    }
    t.insert(infected.begin(), infected.end());

    cin >> m;
    for (int i = 0; i < m; i++) {
        long x, y, d;
        cin >> x >> y >> d;
        Point q(x, y);

        bool result = false;

        if (CGAL::squared_distance(q, t.nearest_vertex(q)->point()) < d) {
            cout << "n";
            continue;
        }

        FaceH face_1 = t.locate(q);

        map<FaceH, bool> face_seen;
        stack<FaceH> face_stack;

        face_stack.push(face_1);
        while(!face_stack.empty()) {
            FaceH face = face_stack.top();
            face_stack.pop();

            // If you are placed on an infinite face, as you are not closer from d to the nearest
            // point, you can escape from the infected
            if (t.is_infinite(face)) {
                result = true;
                break;
            }

            face_seen[face] = true;

            for (int j = 0; j < 3; j++) {
                Segment s = t.segment(face, j);

                // The point can not escape from this face through this edge of the triangulation
                if (s.squared_length() < d*4) { continue; }

```

```
                // If can scape through, take the neighbour face
                FaceH neighbour = face->neighbor(j);
                // If it has been seen continue
                if (face_seen[neighbour]) { continue; }
                // Put to the queue of faces to study if scapatory
                face_stack.push(neighbour);
            }
        }

        if (result) {
            cout << "y";
        } else {
            cout << "n";
        }

    }
    cout << endl;
}


int main() {
    int n;
    cout << fixed << setprecision(0);
    while (true) {
        cin >> n;
        if (n == 0) { break; }
        h1n1(n);
    }
}
```

Germs

Keywords— Delaunay Triangulation, Nearest Neighbor

```
#include <vector>
#include <map>
#include <stack>
#include <algorithm>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/nearest_neighbor_delaunay_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>                   Triangulation;
typedef Triangulation::Finite_faces_iterator                FaceIt;
typedef Triangulation::Edge_iterator                        EdgeIt;
typedef Triangulation::Finite_vertices_iterator             VertexIt;
typedef Triangulation::Face_handle                          FaceH;
typedef Triangulation::Vertex_handle                        VertexH;
typedef K::Point_2                                          Point;
typedef K::Segment_2                                        Segment;

using namespace std;


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

void germs(int n) {
    long l, b, r, t;
    cin >> l >> b >> r >> t;

    vector<Point> bacteries(n);
    vector<K::FT> distances(n);
    for (int i = 0; i < n; i++) {
        long x, y;
        cin >> x >> y;
        bacteries[i] = Point(x, y);
    }

    Triangulation tri;
    tri.insert(bacteries.begin(), bacteries.end());

    int i = 0;
    for (VertexIt vi = tri.finite_vertices_begin(); vi != tri.finite_vertices_end(); vi++) {
        VertexH vh = vi;
        Point p = vh->point();

        // Take the min distance to the neirest neighbour
        K::FT min_distance;
        if (tri.number_of_vertices() > 1) {
            VertexH vn = CGAL::nearest_neighbor(tri, vh);
            min_distance = CGAL::squared_distance(vh->point(), vn->point()) / 4;
        } else {
            // In the case the germ is alone, initialize min distance with the distance to one
            // corner (it will allways be greater than the distance to any of the boundaries)
            min_distance = CGAL::squared_distance(vh->point(), Point(l, b));
        }

        // check with every boundary if the distance is less than the minimum
        min_distance = min(min_distance, (p.x() - l) * (p.x() - l));
        min_distance = min(min_distance, (p.x() - r) * (p.x() - r));
        min_distance = min(min_distance, (p.y() - t) * (p.y() - t));
        min_distance = min(min_distance, (p.y() - b) * (p.y() - b));

        distances[i++] = min_distance;
    }

    sort(distances.begin(), distances.end());
    vector<double> dead_time(n);
    for (int i = 0; i < n; i++) {
        K::FT d = distances[i];
        long ti = ceil_to_double(sqrt(sqrt(d) - .5));
        dead_time[i] = ti;
    }

```

```
    // Print results
    cout << dead_time[0] << " " << dead_time[n/2] << " " << dead_time[n-1] << endl;

}


int main() {
    int n;
    cout << fixed << setprecision(0);
    while (true) {
        cin >> n;
        if (n == 0) { break; }
        germs(n);
    }
}
```

Stamps

Keywords— LP/QP

```
#include <vector>
#include <cmath>

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// Choose exact integral type
typedef CGAL::Gmpzf ET;
// Program and solution types
typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
// Geometric types
typedef CGAL::Exact_predicates_exact_constructions_kernel   K;
typedef K::Point_2                                          P;
typedef K::Segment_2                                        S;

using namespace std;


void stamps() {
    int l, s, w; cin >> l >> s >> w;

    vector<P> lamps(l), stamps(s);
    vector<long> M(s);
    vector<S> walls(w);

    // Read data
    for (int i = 0; i < l; i++) {
        long x, y; cin >> x >> y;
        lamps[i] = P(x, y);
    }
    for (int i = 0; i < s; i++) {
        long x, y;
        cin >> x >> y; stamps[i] = P(x, y);
        cin >> M[i];
    }
    for (int i = 0; i < w; i++) {
        long x_a, y_a, x_b, y_b;
        cin >> x_a >> y_a >> x_b >> y_b;
        walls[i] = S(P(x_a, y_a), P(x_b, y_b));
    }

    // Define quadratic program
    Program lp (CGAL::SMALLER, true, 1, true, 1<<12);
    for (int i = 0; i < s; i++) {                // stamp index
        for (int j = 0; j < l; j++) {            // lamp index
            S l_s(lamps[j], stamps[i]);          // lamp to stamp segment

            bool blocked = false;
            for (int k = 0; k < w; k++) {
                if (CGAL::do_intersect(walls[k], l_s)) {
                    blocked = true;
                    break;
                }
            }

            if (!blocked) {
                double d = CGAL::to_double(1 / l_s.squared_length());
                lp.set_a(j, i, d);
                lp.set_a(j, i+s, d);
            }
        }
        lp.set_b(i, M[i]);
        lp.set_r(i, CGAL::SMALLER);
        lp.set_b(s+i, 1);
        lp.set_r(s+i, CGAL::LARGER);
    }

    // Get solution
    Solution sol = CGAL::solve_linear_program(lp, ET());
    assert (sol.solves_linear_program(lp));

    if (sol.is_infeasible()) {
        cout << "no" << endl;
```

```
    } else {
        cout << "yes" << endl;
    }

}


int main() {
    int t; cin >> t;
    for (int i = 0; i < t; i++) {
        stamps();
    }
}
```

2.5 Advanced Flows

Real Estate Market

Keywords— Max Flow Min Cost

```

#include <iostream>
#include <vector>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt; // Iterator


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap  &revedgemap;

public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap, ReverseEdgeMap &
        revedgemap)
        : G(G), capacitymap(capacitymap), weightmap(weightmap), revedgemap(revedgemap) {}

    void addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};



void real_estate() {
    int N, M, S;
    cin >> N >> M >> S;

    const int MAX = 100;

    Graph G(N+M+S+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);

    Vertex src = vertex(N+M+S, G);
```

```
    Vertex sink = vertex(N+M+S+1, G);

    // Store the graph
    for (int i = 0; i < N; i++) { eaG.addEdge(src, i, 1, 0); }
    for (int i = 0; i < S; i++) {
        long l; cin >> l;
        eaG.addEdge(N+M+i, sink, l, 0);
    }
    for (int i = 0; i < M; i++) {
        int s; cin >> s;
        eaG.addEdge(N+i, N+M+s-1, 1, 0);
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            long b; cin >> b;
            eaG.addEdge(i, N+j, 1, MAX-b);
        }
    }

    successive_shortest_path_nonnegative_weights(G, src, sink);
    int cost = find_flow_cost(G);
    int flow = 0;
    // Iterate over all edges leaving the source to sum up the flow values.
    OutEdgeIt e, eend;
    for(tie(e, eend) = out_edges(vertex(src,G), G); e != eend; ++e) {
        flow += capacitymap[*e] - rescapacitymap[*e];
    }
    cost = flow*MAX - cost;

    cout << flow << " " << cost << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        real_estate();
    }
}
```

Satellites

Keywords— Maximum Flow, Bipartite Matchings, DFS

```

#include <iostream>
#include <vector>
#include <queue>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};


void satellites() {
    int g, s, l;
    cin >> g >> s >> l;

    Graph G(g+s+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    Vertex src = vertex(g+s, G);
    Vertex sink = vertex(g+s+1, G);

    int from, to;
    for (int i = 0; i < l; i++) {
        cin >> from >> to;
        eaG.addEdge(from, g + to, 1);
    }
    for (int i = 0; i < g; i++) {
        eaG.addEdge(src, i, 1);
    }
```

```
    for (int i = g; i < g + s; i++) {
        eaG.addEdge(i, sink, 1);
    }

    push_relabel_max_flow(G, src, sink);

    // BFS to find vertex set S
    vector<int> vis(g+s+2, false); // visited flags
    std::queue<int> Q; // BFS queue (from std:: not boost::)
    vis[src] = true; // Mark the source as visited
    Q.push(src);
    while (!Q.empty()) {
        const int u = Q.front();
        Q.pop();
        OutEdgeIt ebeg, eend;
        for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
            const int v = target(*ebeg, G);
            // Only follow edges with spare capacity
            if (rescapacitymap[*ebeg] == 0 || vis[v]) continue;
            vis[v] = true;
            Q.push(v);
        }
    }

    int count_g = 0, count_s = 0;
    vector<int> result;
    for (int i = 0; i < g; i++) {
        if (!vis[i]) {
            count_g++;
            result.push_back(i);
        }
    }
    for (int i = g; i < g + s; i++) {
        if (vis[i]) {
            count_s++;
            result.push_back(i - g);
        }
    }

    cout << count_g << " " << count_s << endl;
    for (int i = 0; i < result.size(); i++) {
        cout << result[i] << " ";
        if (i == result.size() - 1) cout << endl;
    }

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        satellites();
    }
}
```

Algocoön

Keywords— Max Flow Min Cut, DFS

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <queue>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/cycle_canceling.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt;

// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};


void algocoon() {
    int n, m; cin >> n >> m;

    Graph G(n);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    for (int i = 0; i < m; i++) {
        int from, to, cost;
        cin >> from >> to >> cost;
        eaG.addEdge(from, to, cost);
    }

    int best_src = -1, best_sink = -1, best_value = numeric_limits<int>::max();
    int flow;
```

```
    for (int i = 1; i < n; i++) {
        flow = push_relabel_max_flow(G, 0, i);
        if (flow < best_value) {
            best_value = flow; best_src = 0; best_sink = i;
        }
        flow = push_relabel_max_flow(G, i, 0);
        if (flow < best_value) {
            best_value = flow; best_src = i; best_sink = 0;
        }
    }

    flow = push_relabel_max_flow(G, best_src, best_sink);

    // BFS to find vertex set S
    vector<bool> vis(n, false); // visited flags
    std::queue<int> Q; // BFS queue (from std:: not boost::)
    vis[best_src] = true; // Mark the source as visited
    Q.push(best_src);
    while (!Q.empty()) {
        const int u = Q.front();
        Q.pop();
        OutEdgeIt ebeg, eend;
        for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
            const int v = target(*ebeg, G);
            // Only follow edges with spare capacity
            if (rescapacitymap[*ebeg] == 0 || vis[v]) continue;
            vis[v] = true;
            Q.push(v);
        }
    }

    cout << flow << endl;
    cout << count(vis.begin(), vis.end(), true);
    for (int i = 0; i < n; ++i) {
        if (vis[i]) cout << " " << i;
    }
    cout << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        algocoon();
    }
}
```

Canteen

Keywords— Max Flow Min Cost

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt; // Iterator


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap  &revedgemap;

public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap, ReverseEdgeMap &
        revedgemap)
        : G(G), capacitymap(capacitymap), weightmap(weightmap), revedgemap(revedgemap) {}

    void addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};



void canteen() {
    int n; cin >> n;

    const int MAX_PRICE = 20;

    vector<int> a(n), c(n), s(n), p(n), v(n-1), e(n-1);
    for (int i = 0; i < n; i++) { cin >> a[i] >> c[i]; }
    for (int i = 0; i < n; i++) { cin >> s[i] >> p[i]; }
    for (int i = 0; i < n-1; i++) { cin >> v[i] >> e[i]; }

    // Create Graph and Maps
    Graph G(n+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
```

```
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);

    Vertex src = n, sink = n+1;
    for (int i = 0; i < n; i++) {
        eaG.addEdge(src, i, a[i], c[i]);
        eaG.addEdge(i, sink, s[i], MAX_PRICE-p[i]);
        if (i < n-1) { eaG.addEdge(i, i+1, v[i], e[i]); }
    }

    // Compute flow and cost
    successive_shortest_path_nonnegative_weights(G, src, sink);
    int cost = find_flow_cost(G);
    int flow = 0;

    for (int i = 0; i < n; i++) {
        Edge ee; bool success;
        tie(ee, success) = edge(i, sink, G);
        int f = capacitymap[ee] - rescapacitymap[ee];
        flow += f;
        cost -= f * MAX_PRICE;
    }
    cost = -cost;

    int total_s = accumulate(s.begin(), s.end(), 0);
    string text = (total_s == flow) ? "possible " : "impossible ";
    cout << text << flow << " " << cost << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        canteen();
    }
}
```

Casino Royale

Keywords— Max Flow Min Cost

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt; // Iterator


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap &revedgemap;

public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap, ReverseEdgeMap &
        revedgemap)
        : G(G), capacitymap(capacitymap), weightmap(weightmap), revedgemap(revedgemap) {}

    Edge addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
        return e;
    }
};

void casino_royale() {
    int n, m, l;
    cin >> n >> m >> l;

    const int MAX_PRIORITY = 1<<7;

    // Create Graph and Maps
    Graph G(n+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);
    Vertex v_source = n, v_target = n + 1;

    for (int i = 0; i < n-1; i++) { eaG.addEdge(i, i+1, l, MAX_PRIORITY); }
```

```
    eaG.addEdge(v_source, 0, 1, 0);
    eaG.addEdge(n-1, v_target, 1, 0);

    // Add the missions edges
    int x, y, q;
    for (int i = 0; i < m; i++) {
        cin >> x >> y >> q;
        eaG.addEdge(x, y, 1, (MAX_PRIORITY*(y-x) - q));
    }

    // Find MaxFlowMinCost
    successive_shortest_path_nonnegative_weights(G, v_source, v_target);
    int cost = find_flow_cost(G), flow = 0;
    OutEdgeIt e, eend;
    for(tie(e, eend) = out_edges(vertex(v_source,G), G); e != eend; ++e) {
        flow += capacitymap[*e] - rescapacitymap[*e];
    }
    cost -= MAX_PRIORITY * (n-1) * flow;
    cout << -cost << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        casino_royale();
    }
}
```

Chapter 3

Exam Preparation

Odd Route

Keywords— Dijkstra Shortest Path

```
1
2  // STL includes
3  #include <vector>
4  #include <iostream>
5  // BGL includes
6  #include <boost/graph/adjacency_list.hpp>
7  #include <boost/graph/dijkstra_shortest_paths.hpp>
8
9  // Namespaces
10 using namespace std;
11 using namespace boost;
12
13 // Directed graph with integer weights on edges.
14 typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_weight_t, int> > Graph;
15 typedef graph_traits<Graph>::vertex_descriptor      Vertex;
16 typedef graph_traits<Graph>::edge_descriptor        Edge;
17 // Property map edge -> weight
18 typedef property_map<Graph, edge_weight_t>::type    WeightMap;
19
20
21 void odd_route() {
22     int n, m; cin >> n >> m;
23     int s, t; cin >> s >> t;
24
25     Graph G(4*n);
26     WeightMap weightmap = get(edge_weight, G);
27
28     /*
29     Split all vertices u, 0 <= u < V(G) into 4 parts
30     4*u (even length , even weight)
31     4*u+1 (even length, odd weight)
32     4*u+2 (odd length, even weight)
33     4*u+3 (odd length, odd weight)
34     */
35
36     Edge e;
37     int u, v, w;
38     for (int i = 0; i < m; i++) {
39         cin >> u >> v >> w;
40         tie(e, tuples::ignore) = add_edge(4*u,     4*v + 2 + w%2, G); weightmap[e] = w;
41         tie(e, tuples::ignore) = add_edge(4*u + 1, 4*v + 3 - w%2, G); weightmap[e] = w;
42         tie(e, tuples::ignore) = add_edge(4*u + 2, 4*v + w%2,     G); weightmap[e] = w;
43         tie(e, tuples::ignore) = add_edge(4*u + 3, 4*v + 1 - w%2, G); weightmap[e] = w;
44     }
45
46     vector<int> dist(4 * n), pred(4 * n);
47     dijkstra_shortest_paths(G, 4*s,
48                             predecessor_map(make_iterator_property_map(pred.begin(),
49                             get(vertex_index, G)))
50                               .distance_map(make_iterator_property_map(dist.begin(),
51                                 get(vertex_index, G))
52                               )
53                            );
54     if (dist[4*t + 3] == INT_MAX)
55       cout << "no" << endl;
56     else
57       cout << dist[4*t + 3] << endl;
58
59 }
```

```
60
61 int main() {
62     ios_base::sync_with_stdio(false);
63     int T; cin >> T;
64     for (int t=0; t < T; t++){
65         odd_route();
66     }
67 }
```

Light the Stage

Keywords— Delaunay Triangulation, Nearest Neighbor

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

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel     K;
typedef CGAL::Delaunay_triangulation_2<K>                       Triangulation;
typedef K::Point_2                                              P;

using namespace std;


void light_the_stage() {
    int m, n, h;
    cin >> m >> n;

    vector<P> l(n), p(m);
    vector<int> r(m);
    vector<int> dead_time(m, INT_MAX);

    // Read input
    int x, y;
    for (int i = 0; i < m; i++) {
        cin >> x >> y;
        p[i] = P(x, y);
        cin >> r[i];
    }
    cin >> h;
    for (int i = 0; i < n; i++) {
        cin >> x >> y;
        l[i] = P(x, y);
    }

    // Create triangulation for the lamps points
    Triangulation tri;
    tri.insert(l.begin(), l.end());

    // Iterate over all participants and look for the closest lamp
    for (int i = 0; i < m; i++) {
        P v = tri.nearest_vertex(p[i])->point();
        double d_2 = CGAL::squared_distance(v, p[i]);
        double d = (r[i] + h);
        double d_max = d * d;

        if (d_max <= d_2) continue;

        for (int j = 0; j < n; j++) {
            double dist = CGAL::squared_distance(p[i], l[j]);
            if (d_max > dist) {
                dead_time[i] = j;
                break;
            }
        }
    }

    int winner_time = *std::max_element(dead_time.begin(), dead_time.end());

    // Print result
    for (int i = 0; i < m; i++) {
        if (dead_time[i] == winner_time) cout << i << " ";
    }
    cout << endl;
}


int main() {
    std::ios_base::sync_with_stdio(false);
    int t; cin >> t;
    for (int i = 0; i < t; i++) {
        light_the_stage();
    }
}
```

Bonus Level

Keywords— Max Flow Min Cost

```
// Includes
// ========
// STL includes
#include <iostream>
#include <cstdlib>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
// Namespaces
using namespace boost;
using namespace std;

// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator  OutEdgeIt; // Iterator

// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap  &revedgemap;

public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap, ReverseEdgeMap &
        revedgemap)
        : G(G), capacitymap(capacitymap), weightmap(weightmap), revedgemap(revedgemap) {}

    void addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};

void bonus_level() {
    int n; cin >> n;

    const int MAX_COST = 100;
    // Create Graph and Maps
    Graph G(3 * n * n);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);

    Vertex src = add_vertex(G);
    Vertex sink = add_vertex(G);

    int a, pos;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> a;
            pos = 3 * j + 3 * n * i;
```

```
            eaG.addEdge(pos, pos + 1, 1, MAX_COST - a);
            eaG.addEdge(pos, pos + 2, 1, MAX_COST);
            if (i != n - 1) {
                eaG.addEdge(pos + 1, pos + 3 * n, 1, 0);
                eaG.addEdge(pos + 2, pos + 3 * n, 1, 0);
            }
            if (j != n -1) {
                eaG.addEdge(pos + 1, pos + 3, 1, 0);
                eaG.addEdge(pos + 2, pos + 3, 1, 0);
            }
        }
    }
    eaG.addEdge(src, 0, 2, 0);
    eaG.addEdge(3 * n * n - 1, sink, 1, 0);
    eaG.addEdge(3 * n * n - 2, sink, 1, 0);

    successive_shortest_path_nonnegative_weights(G, src, sink);
    int cost = find_flow_cost(G);
    cout << - cost + 2 * (2 * n - 1) * MAX_COST << endl;

}

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; t++){
        bonus_level();
    }
}
```

Sith

Keywords— Binary Search, Delaunay Triangulation, Connected Components

```cpp
// Includes
// ========
// STL includes
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
// CGAL
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>

// Namespaces
using namespace std;
using namespace boost;

// Directed graph with integer weights on edges.
typedef adjacency_list<vecS, vecS, undirectedS,
    no_property, property<edge_weight_t, int> >         Graph;
typedef graph_traits<Graph>::vertex_descriptor          Vertex; // Vertex type
typedef graph_traits<Graph>::edge_descriptor            Edge;   // Edge type
typedef graph_traits<Graph>::edge_iterator              EdgeIt; // Edge iterator
// Property map edge -> weight
typedef property_map<Graph, edge_weight_t>::type        WeightMap;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                                          P;

typedef CGAL::Triangulation_vertex_base_with_info_2<int,K>  Vb;
typedef CGAL::Triangulation_face_base_2<K>                  Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>         Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>               Triangulation;
typedef Triangulation::Edge_iterator                        Edge_iterator;
typedef Triangulation::Vertex_handle                        VertexH;

int get_k_planets(int t, vector<pair<P, int> >& planets, const int r, const int n) {
    assert(t > 0);
    assert(t < n);
    int n_not_conquered = n - t;

    Graph G(n_not_conquered);

    Triangulation tr;
    tr.insert(planets.begin(), planets.begin() + n_not_conquered);

    for (Edge_iterator e = tr.finite_edges_begin(); e != tr.finite_edges_end(); e++) {
        VertexH v1 = e->first->vertex((e->second + 1) % 3);
        VertexH v2 = e->first->vertex((e->second + 2) % 3);
        double dist = CGAL::squared_distance(v1->point(), v2->point());
        double rr = r;
        double min_dist = rr * rr;
        if (dist <= min_dist) {
            add_edge(v1->info(), v2->info(), G);
        }
    }

    // Analyze components
    vector<int> component(n_not_conquered);
    int n_components = connected_components(G, &component[0]);
    vector<int> sizes(n_components, 0);
    int largest = -1;
    for (int i = 0; i < n_not_conquered; i++) {
        int c = component[i];
        sizes[c]++;
        largest = max(largest, sizes[c]);
    }

    return min(t, largest);
}

int search_t(int begin, int end, vector<pair<P, int> >& planets, const int r, const int n) {
    if (begin + 1 == end)
        return max(get_k_planets(begin, planets, r, n), get_k_planets(end, planets, r, n));
    else {
```

```
        int middle = (begin + end) / 2;
        int b = get_k_planets(middle, planets, r, n);
        if (b == middle) return search_t(middle, end, planets, r, n);
        else return search_t(begin, middle, planets, r, n);
    }
}

void sith() {
    int n, r; cin >> n >> r;

    vector<pair<P, int> > p(n);
    long x, y;
    for (int i = n-1; i >= 0; i--) {
        cin >> x >> y;
        p[i] = make_pair(P(x, y), i);
    }

    int t = search_t(0, n/2, p, r, n);
    cout << t << endl;

}

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; t++){
        sith();
    }
}
```

Clues

Keywords— Delaunay Triangulation, Nearest Neighbor

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

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

using namespace std;

// CGAL typedefs
typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef pair<int, bool>                                             info_t;
typedef CGAL::Triangulation_vertex_base_with_info_2<info_t, K>      Vb;
typedef CGAL::Triangulation_data_structure_2<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds>                      Triangulation;
typedef Triangulation::Finite_faces_iterator                        FaceIt;
typedef Triangulation::Edge_iterator                                EdgeIt;
typedef Triangulation::Face_handle                                  FaceH;
typedef Triangulation::Vertex_handle                                VertexH;
typedef Triangulation::Vertex_circulator                            VertexCirculator;
typedef K::Point_2                                                  Point;
typedef K::Segment_2                                                Segment;
typedef K::Triangle_2                                               Triangle;


bool has_interference(Triangulation const & trg, K::FT const & rr) {
    for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); e++)
        if (trg.segment(*e).squared_length() <= rr) return true;
    return false;
}

bool try_two_colors(Triangulation & trg, K::FT const & rr) {
    for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); v++)
        v->info() = { 0, false };

    int components = 0;
    Triangulation trg0, trg1;
    for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); v++) {
        if (v->info().first == 0) {
            v->info().first = ++components;
            vector<VertexH> stack(1, v);
            do {
                VertexH h = stack.back();
                stack.pop_back();
                VertexCirculator c = trg.incident_vertices(h);
                do if (!trg.is_infinite(c) && CGAL::squared_distance(h->point(), c->point()) <= rr
                    ) {
                    if (c->info() == h->info()) return false;
                    if (c->info().first == 0) {
                        stack.push_back(c);
                        c->info() = { components, !h->info().second };
                    }
                } while (++c != trg.incident_vertices(h));
            } while (!stack.empty());
        }

        if (v->info().second) trg1.insert(v->point());
        else trg0.insert(v->point());
    }

    return !has_interference(trg0, rr) && !has_interference(trg1, rr);
}

void clues() {
    int n, m;
    long r;
    cin >> n >> m >> r;
    K::FT rr(r*r);

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

    Triangulation trg;
    trg.insert(stations.begin(), stations.end());
    bool success = try_two_colors(trg, rr);

```

```
    for (int i = 0; i < m; i++) {
        Point holmes, watson;
        cin >> holmes >> watson;

        if (success) {
            if (CGAL::squared_distance(holmes, watson) <= rr) {
                cout << "y"; continue;
            }

            auto holmes_station = trg.nearest_vertex(holmes);
            auto watson_station = trg.nearest_vertex(watson);
            if (holmes_station->info().first == watson_station->info().first &&
                CGAL::squared_distance(holmes_station->point(), holmes) <= rr &&
                CGAL::squared_distance(watson_station->point(), watson) <= rr)
            {
                cout << "y"; continue;
            }
        }
        cout << "n";
    }
    cout << endl;

}


int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        clues();
    }
}
```

Punch

***Keywords*—**

```cpp
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// choose exact integral type
typedef CGAL::Gmpz ET;

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

void punch(){
    int n, k; cin >> n >> k;

    Program qp (CGAL::EQUAL, true, 0, false, 0);
    int c, v;
    for (int i = 0; i < n; i++) {
        cin >> c >> v;
        qp.set_c(i, c);
        qp.set_a(i, 0, v);
        // qp.set_d(i, i, 1);
    }
    qp.set_b(0, k);

    Solution s = CGAL::solve_linear_program(qp, ET());
    assert(s.solves_linear_program(qp));
    assert(s.is_optimal());

    // CGAL::Quadratic_program_solution<ET>::Variable_value_iterator opt = s.variable_values_begin
        ();
    // int value = 0;
    // for (; opt != s.variable_values_end(); opt++) {
    //     int tmp = int(CGAL::to_double(*(opt)));
    //     if (tmp > 0) value++;
    // }
    // cout << value << endl;
    cout << CGAL::to_double(s.objective_value()) << endl;

}


int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;

    for (int i = 0; i < T; i++) {
        punch();
    }
}
```

Carsharing

Keywords— Max Flow Min Cost

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <map>
// BGL include
#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;

// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt; // Iterator

// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap  &revedgemap;

public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap, ReverseEdgeMap &
        revedgemap)
        : G(G), capacitymap(capacitymap), weightmap(weightmap), revedgemap(revedgemap) {}

    void addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};

struct Booking {
    int s, t, d, a, p;
};

void carsharing() {
    int N, S;
    cin >> N >> S;

    const int MAXL = 100, MAXT = 100000, MAXP = 100;
    const int INF = MAXL*S;

    // Define variables to store data
    vector<int> L(S);
    vector<Booking> B;
    vector<set<int> > times(S);
    vector<map<int, int> > M(S);
    vector<int> psum(S+1);

    // Read data
    for (int i = 0; i < S; i++) {
```

```
        cin >> L[i];
        times[i].insert(0); times[i].insert(MAXT);
    }

    int s, t, d, a, p;
    for (int i = 0; i < N; i++) {
        cin >> s >> t >> d >> a >> p;
        s--; t--;
        times[s].insert(d); times[t].insert(a);
        B.push_back({s,t,d,a,p});
    }

    for (int s = 0; s < S; s++) {
        int i = 0;
        for (auto &t : times[s]) {
            M[s][t] = i;
            i++;
        }
        psum[s+1] = psum[s] + M[s].size();
    }

    int T = psum.back();
    int v_source = T, v_target = T + 1;
    Graph G(T+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);

    for (int i = 0; i < S; i++) {
        eaG.addEdge(v_source, psum[i], L[i], 0);
        eaG.addEdge(psum[i+1]-1, v_target, INF, 0);
        int it = -1, lastt = 0;
        for (auto &t : times[i]) {
            if (it != -1) {
                eaG.addEdge(psum[i]+it, psum[i]+it+1, INF, MAXP*(t-lastt));
            }
            it++; lastt = t;
        }
    }

    for (int i = 0; i < N; i++) {
        eaG.addEdge(psum[B[i].s] + M[B[i].s][B[i].d],
                    psum[B[i].t] + M[B[i].t][B[i].a],
                    1, ((B[i].a - B[i].d) * MAXP) - B[i].p);
    }

    successive_shortest_path_nonnegative_weights(G, v_source, v_target);
    int flow = 0;
    // Iterate over all edges leaving the source to sum up the flow values.
    OutEdgeIt e, eend;
    for(tie(e, eend) = out_edges(vertex(v_source,G), G); e != eend; ++e) {
        flow += capacitymap[*e] - rescapacitymap[*e];
    }
    int cost = MAXP * MAXT * flow - find_flow_cost(G);
    cout << cost << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        carsharing();
    }
}
```

Planks

Keywords— Backtracking, Split and List

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

using namespace std;

typedef vector<vector<int> > vecvec;

void back_track(vector<int>& planks, vecvec& assigment, vecvec& F, int id, int ubound) {
    if (id >= ubound) {
        vector<int> tuple(4, 0);
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < assigment[i].size(); j++)
                tuple[i] += planks[assigment[i][j]];
        }
        F.push_back(tuple);
        return;
    }

    for (int side = 0; side < 4; side++) {
        assigment[side].push_back(id);
        back_track(planks, assigment, F, id+1, ubound);
        assigment[side].pop_back();
    }
}

void planks() {
    int n; cin >> n;

    vector<int> planks(n);
    int sum = 0;
    long long n_sol = 0;

    for (int i = 0; i < n; i++) { cin >> planks[i]; sum += planks[i]; }

    if (sum % 4 != 0) {
        cout << 0 << endl;
        return;
    }

    vecvec F1, F2, assigment1(4), assigment2(4);
    back_track(planks, assigment1, F1, 0, n/2);
    back_track(planks, assigment2, F2, n/2, n);
    sort(F2.begin(), F2.end());

    for (int idx = 0; idx < F1.size(); idx++) {
        vector<int> member = F1[idx];
        for (int i = 0; i < 4; i++)
            member[i] = sum/4 - member[i];

        pair<vecvec::iterator, vecvec::iterator> bounds;
        bounds = equal_range(F2.begin(), F2.end(), member);
        n_sol += distance(bounds.first, bounds.second);
    }

    cout << n_sol / 24 << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        planks();
    }
}
```

New Tiles

***Keywords*—**

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


void new_tiles() {
    int h, w;
    cin >> h >> w;

    vector<uint32_t> floor_plan(h, 0);
    int p;
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            cin >> p;
            if (p == 1) floor_plan[i] |= 0x01<<j;
        }
    }



}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        new_tiles();
    }
}
```

Golden Eye

Keywords— Delaunay Triangulation, Disjoint Sets

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Triangulation_face_base_2.h>
#include <vector>
#include <boost/pending/disjoint_sets.hpp>

typedef CGAL::Exact_predicates_inexact_constructions_kernel     K;
typedef K::Point_2                                              P;
typedef CGAL::Triangulation_vertex_base_with_info_2<int,K>      Vb;
typedef CGAL::Triangulation_face_base_2<K>                      Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>             Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>                   Delaunay;
typedef Delaunay::Vertex_handle                                 VertexH;
typedef Delaunay::Finite_edges_iterator                         EI;
typedef std::pair<P,int>                                        IPoint;
typedef boost::disjoint_sets_with_storage<>                     Uf;

using namespace std;


struct Edge {
    Edge(int u_, int v_, K::FT sql_) : u(u_), v(v_), sql(sql_) {}
    int u, v;       // endpoints
    K::FT sql;      // squared length
};

inline bool operator<(const Edge& e, const Edge& f) { return e.sql < f.sql; }

void goldeneye() {
    int n, m; double p;
    cin >> n >> m >> p;

    // Read jammers and build Delaunay
    vector<IPoint> jammers(n);
    long x, y;
    for (int i = 0; i < n; i++) {
        cin >> x >> y;
        jammers[i] = IPoint(P(x, y), i);
    }
    Delaunay t;
    t.insert(jammers.begin(), jammers.end());

    // Extract edges and sort by length
    vector<Edge> edges;
    edges.reserve(3*n);
    for (EI e = t.finite_edges_begin(); e != t.finite_edges_end(); e++) {
        edges.push_back(Edge(e->first->vertex((e->second+1)%3)->info(),
                             e->first->vertex((e->second+2)%3)->info(),
                             t.segment(e).squared_length()));
    }
    sort(edges.begin(), edges.end());

    // Compute components with power consumption p
    Uf ufp(n);
    typedef vector<Edge>::const_iterator ECI;
    for (ECI e = edges.begin(); e != edges.end() && e->sql <= p; e++)
        ufp.union_set(e->u, e->v);

    // Handle missions
    K::FT a = 0, b = 0;
    Uf ufa(n), ufb(n);
    ECI ai = edges.begin(), bi = edges.begin();
    int x0, y0, x1, y1;
    for (int i = 0; i < m; i++) {
        cin >> x0 >> y0 >> x1 >> y1;
        P p0(x0, y0), p1(x1, y1);
        VertexH v0 = t.nearest_vertex(p0), v1 = t.nearest_vertex(p1);
        K::FT d = 4 * max(CGAL::squared_distance(p0, v0->point()),
                          CGAL::squared_distance(p1, v1->point()));

        if (d <= p && ufp.find_set(v0->info()) == ufp.find_set(v1->info())) {
            // mission possible with power p -> also with b
            cout << "y";
            if (d > b) b = d;
            for (; bi != edges.end() &&
                    ufb.find_set(v0->info()) != ufb.find_set(v1->info()); bi++)
```

```
                ufb.union_set(bi->u, bi->v);
        } else cout << "n";
        // ensure it is possible at power a
        if (d > a) a = d;
        for (; ai != edges.end() &&
                ufa.find_set(v0->info()) != ufa.find_set(v1->info()); ai++)
            ufa.union_set(ai->u, ai->v);

    }
    if (ai != edges.begin() && (ai-1)->sql > a) a = (ai-1)->sql;
    if (bi != edges.begin() && (bi-1)->sql > b) b = (bi-1)->sql;
    cout << endl << a << endl << b << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    cout << setiosflags(ios::fixed) << setprecision(0);
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        goldeneye();
    }
}
```

Corbusier

Keywords— Dynamic Programming

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

using namespace std;

void corbusier() {
    int n, i, k;
    cin >> n >> i >> k;

    vector<int> disks(n);
    vector<vector<bool> > T(n, vector<bool>(k, false));

    for (int j = 0; j < n; j++) cin >> disks[j];

    int h;
    for (int d = 0; d < n; d++) {
        h = disks[d];

        T[d][h%k] = true;
        if (d > 0) {
            for (int j = 0; j < k; j++) {
                if (T[d-1][j]) {
                    T[d][j] = true;
                    T[d][(j+h)%k] = true;
                }
            }
        }

        if (T[d][i]) {
            cout << "yes\n";
            return;
        }
    }
    cout << "no\n";
}


int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        corbusier();
    }
}
```

Placing Knights

Keywords— Maximum Cardinality Matching

```
#include <iostream>
#include <map>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
typedef adjacency_list<vecS, vecS, undirectedS> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;


void add_connection(Graph& G, vector<vector<bool> >& chessboard, int n, int i, int j, int new_i,
    int new_j) {
    if (new_i >= n || new_i < 0) return;
    if (new_j >= n || new_j < 0) return;
    if ( !chessboard[new_i][new_j] ) return;

    add_edge(i*n + j, new_i*n + new_j, G);
    return;
}


void placing_knights() {
    int n; cin >> n;
    vector<vector<bool> > chessboard(n, vector<bool>(n));

    // Read chessboardx
    int c = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            bool p; cin >> p;
            chessboard[i][j] = p;
            if ( p ) c++;
        }
    }

    Graph G(n*n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (chessboard[i][j]) {
                add_connection(G, chessboard, n, i, j, i-1, j-2);
                add_connection(G, chessboard, n, i, j, i-1, j+2);
                add_connection(G, chessboard, n, i, j, i+1, j-2);
                add_connection(G, chessboard, n, i, j, i+1, j+2);
                add_connection(G, chessboard, n, i, j, i-2, j-1);
                add_connection(G, chessboard, n, i, j, i-2, j+1);
                add_connection(G, chessboard, n, i, j, i+2, j-1);
                add_connection(G, chessboard, n, i, j, i+2, j+1);
            }
        }
    }

    vector<Vertex> mate(n*n);
    checked_edmonds_maximum_cardinality_matching(G, &mate[0]);
    int matches = matching_size(G, &mate[0]);

    cout << c - matches << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        placing_knights();
    }
}
```

Radiation

Keywords— Binary Search, LP/QP

```cpp
#include <iostream>
#include <vector>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// choose exact integral type
typedef CGAL::Gmpz ET;

// program and solution types
typedef CGAL::Quadratic_program<long> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;


bool solve(vector<vector<vector<long> > >& coefficients, const int h, const int t, const int d) {

    int nb_coeff = coefficients[0][d].size();

    Program lp (CGAL::SMALLER, false, 0, false, 0);
    CGAL::Quadratic_program_options options;
    options.set_pricing_strategy(CGAL::QP_BLAND);     // Bland's rule

    for (int i = 0; i < (h + t); i++) {
        for (int k = 0; k < nb_coeff; k++) {
            lp.set_a(k, i, coefficients[i][d][k]);
        }
        lp.set_b(i, 0);

        if (i >= h) lp.set_a(nb_coeff, i, 1); // Epsilon
    }
    lp.set_c(nb_coeff, -1);
    lp.set_l(nb_coeff, true, 0);

    Solution s = CGAL::solve_linear_program(lp, ET(), options);
    return s.is_unbounded();
}


void radiation(){
    int h, t;
    cin >> h >> t;

    const int D = 30;

    vector<vector<long> > health(h, vector<long>(3)), tumor(t,vector<long>(3));

    for (int i = 0; i < h; i++) cin >> health[i][0] >> health[i][1] >> health[i][2];
    for (int i = 0; i < t; i++) cin >> tumor[i][0] >> tumor[i][1] >> tumor[i][2];

    vector<vector<vector<long> > >coefficients(h+t, vector<vector<long> >(D+1));

    for (int i = 0; i < (h + t); i++) {
        vector<long> cell = (i < h) ? health[i]: tumor[i-h];

        for (int d = 1; d <= D; d++) {
            long coef_x = 1;
            for (int x = 0; x <= d; x++) {
                long coef_y = 1;
                for (int y = 0; y <= d; y++) {
                    long coef_z = 1;
                    for (int z = 0; z <= d; z++) {
                        if (x + y + z > d) continue;
                        if (i >=h) coefficients[i][d].push_back(coef_x * coef_y * coef_z);
                        else coefficients[i][d].push_back(-coef_x * coef_y * coef_z);
                        coef_z *= cell[2];
                    }
                    coef_y *= cell[1];
                }
                coef_x *= cell[0];
            }
        }
    }
```

```
78
79      int upper = 1;
80      while (!solve(coefficients, h, t, upper) && upper <= D) upper *= 2;
81
82      if (upper > D) cout << "impossible" << endl;
83
84      int low = upper / 2;
85      while (low + 1 != upper) {
86          int mid = low + (upper - low) / 2;
87          if (solve(coefficients, h, t, mid)) {
88              upper = mid;
89          } else low = mid;
90      }
91
92      cout << upper << endl;
93  }
94
95
96  int main() {
97      int T; cin >> T;
98      for (int t = 0; t < T; t++) {
99          radiation();
100     }
101 }
```

The Empire Strikes Back

Keywords— ***Keywords***— Delaunay Triangulation, LP/QP

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
#include <limits.h>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpzf.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

// LP
typedef CGAL::Gmpzf ET;
typedef CGAL::Quadratic_program<double> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
// Delaunay
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>                   Triangulation;
typedef K::Point_2                                          Point;

using namespace std;

void strikes_back() {
    int a, s, b, e;
    cin >> a >> s >> b >> e;

    vector<Point> asteroid(a), shooting_points(s), bounty_hunters(b);
    vector<int> asteroid_densities(a);
    long x, y;
    // Read data
    for (int i = 0; i < a; i++) {
        cin >> x >> y;
        asteroid[i] = Point(x, y);
        cin >> asteroid_densities[i];
    }
    for (int i = 0; i < s; i++) {
        cin >> x >> y;
        shooting_points[i] = Point(x, y);
    }
    for (int i = 0; i < b; i++) {
        cin >> x >> y;
        bounty_hunters[i] = Point(x, y);
    }

    // Create a triangulation for the bounty hunters to find the closest one at each shooting
        point
    Triangulation bh;
    bh.insert(bounty_hunters.begin(), bounty_hunters.end());

    // Initialize linear program
    Program lp (CGAL::LARGER, true, 0, false, 0);

    double max_d = LONG_MAX;    // Distance to the closest bounty hunters
    double d;
    for (int i = 0; i < s; i++) {
        if (b > 0) {            // In case the set ob bounty hunters are empty
            Point cl_bh = bh.nearest_vertex(shooting_points[i])->point();
            max_d = CGAL::to_double(CGAL::squared_distance(cl_bh, shooting_points[i]));
        }

        for (int j = 0; j < a; j++) {
            // Compute the distance of every shooting point to each asteroid
            d = CGAL::to_double(CGAL::squared_distance(shooting_points[i], asteroid[j]));
            // If the closests bounty hunter is nearer than the asteroid, then to this no energy
            // will arrive
            if(d >= max_d) continue;

            // If arrives compute the coefficient
            double coef = 1 / ((double) max(1., d));
            lp.set_a(i, j, coef);
        }
        lp.set_c(i, 1);         // Minimize the sum of energy at each shot
        lp.set_a(i, a, 1);      // (*) Set a last inequality upbounding the total amount of energy
                                // used
    }
    lp.set_b(a, e);                 // (*)
    lp.set_r(a, CGAL::SMALLER);     // (*)
```

```
    // Set a lower limit to the amount of energy that must arrive to each asteroid
    for (int j = 0; j < a; j++) {
        lp.set_b(j, asteroid_densities[j]);
    }

    // Solve LP
    Solution sol = CGAL::solve_linear_program(lp, ET());
    assert (sol.solves_linear_program(lp));

    // If exists solution then its possible to do it
    if (sol.is_optimal()) cout << "y" << endl;
    else cout << "n" << endl;

}

int main() {
    int T; cin >> T;
    for (int i = 0; i < T; i++) {
        strikes_back();
    }
}
```

Bob's Burden

Keywords— Dijkstra Shortest Path

```
// Includes
// ========
// STL includes
#include <iostream>
#include <vector>

// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
// Namespaces
using namespace std;
using namespace boost;

// BGL Graph definitions
// =====================
// Graph Type, OutEdgeList Type, VertexList Type, (un)directedS
typedef adjacency_list<vecS, vecS, directedS,       // Use vecS for the VertexList! Choosing setS for the OutEdgeList disallows parallel edges.
        no_property,                // interior properties of vertices
        property<edge_weight_t, int>        // interior properties of edges
        >                   Graph;
typedef graph_traits<Graph>::edge_descriptor        Edge;       // Edge Descriptor: an object that represents a single edge.
typedef graph_traits<Graph>::vertex_descriptor      Vertex;     // Vertex Descriptor: with vecS vertex list, this is really just an int in the range [0, num_vertices(G)).
typedef graph_traits<Graph>::edge_iterator      EdgeIt;     // to iterate over all edges
typedef graph_traits<Graph>::out_edge_iterator      OutEdgeIt;  // to iterate over all outgoing edges of a vertex
typedef property_map<Graph, edge_weight_t>::type    WeightMap;  // property map to access the interior property edge_weight_t


void bobs_burden() {
    int k; cin >> k;

    int N = k * (k+1) / 2;
    Graph G(2 * N);
    WeightMap weightmap = get(edge_weight, G);

    int pos_in, pos_out, v;
    Edge e; bool success;
    for (int i = 0; i < k; i++) {
        for (int j = 0; j <= i; j++) {
            pos_in = i * (i + 1) / 2 + j;
            pos_out = pos_in + N;
            cin >> v;
            tie(e, success) = add_edge(pos_in, pos_out, G);
            weightmap[e] = v;
            if (j > 0) {
                int pos_left_in = pos_in - 1, pos_left_out = pos_out - 1;
                add_edge(pos_left_out, pos_in, G);
                add_edge(pos_out, pos_left_in, G);
            }
            if (i < k - 1) {
                int pos_down_left_in = pos_in + (i + 1);
                int pos_down_left_out = pos_down_left_in + N;
                int pos_down_right_in = pos_down_left_in + 1;
                int pos_down_right_out = pos_down_right_in + N;
                add_edge(pos_out, pos_down_left_in, G);
                add_edge(pos_down_left_out, pos_in, G);
                add_edge(pos_out, pos_down_right_in, G);
                add_edge(pos_down_right_out, pos_in, G);
            }
        }
    }

    vector<vector<int> > distmap(3, vector<int>(N*2));
    // Compute the distance map for the three point (0,0), (k,0), (k,k)
    dijkstra_shortest_paths(G, N,
        distance_map(make_iterator_property_map(distmap[0].begin(), get(vertex_index, G))));
    dijkstra_shortest_paths(G, 2*N - k,
        distance_map(make_iterator_property_map(distmap[1].begin(), get(vertex_index, G))));
    dijkstra_shortest_paths(G, 2*N - 1,
        distance_map(make_iterator_property_map(distmap[2].begin(), get(vertex_index, G))));

    int minimum_burden = INT_MAX;
    for (int i = 0; i < k; i++) {
```

```
        for (int j = 0; j <= i; j++) {
            pos_in = i * (i + 1) / 2 + j;
            if (pos_in == 0 || pos_in == N - k || pos_in == N -1)
                continue;
            tie(e, success) = edge(pos_in, pos_in+N, G);
            minimum_burden = min(minimum_burden,
                distmap[0][pos_in] + distmap[1][pos_in] + distmap[2][pos_in] + weightmap[e]);
        }
    }

    cout << minimum_burden << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        bobs_burden();
    }
}
```

DHL

Keywords— Dynamic Programming

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

using namespace std;


int dp(vector<vector<int> >& sum, int pos_a, int pos_b, vector<vector<int> >& memory) {

    // Check memory first
    if (memory[pos_a][pos_b] != -1) return memory[pos_a][pos_b];

    // Look for the best combination to reduce the cost given an specific position of the two stacks
    int pick_a, pick_b, new_cost, cost = INT_MAX;

    // Separate the iteration, first look at the first stack and take the first volume
    pick_a = sum[0][pos_a] - sum[0][pos_a-1];
    for(int i = pos_b - 1; i >= 0; i--) {
        if ((pos_a-1 == 0 && i > 0) || (pos_a-1 > 0 && i == 0)) continue;
        pick_b = sum[1][pos_b] - sum[1][i];

        new_cost = (memory[pos_a-1][i] != -1) ? memory[pos_a-1][i] : dp(sum, pos_a-1, i, memory);
        cost = min(cost, new_cost + pick_a*pick_b);
    }

    pick_b = sum[1][pos_b] - sum[1][pos_b-1];
    for(int i = pos_a - 1; i >= 0; i--) {
        if ((i == 0 && pos_b-1 > 0) || (i > 0 && pos_b-1 == 0)) continue;
        pick_a = sum[0][pos_a] - sum[0][i];

        new_cost = (memory[i][pos_b-1] != -1) ? memory[i][pos_b-1] : dp(sum, i, pos_b-1, memory);
        cost = min(cost, new_cost + pick_a*pick_b);
    }

    memory[pos_a][pos_b] = cost;
    return cost;

}

void dhl() {
    int n; cin >> n;

    vector<vector<int> > sum(2, vector<int>(n+1));
    sum[0][0] = 0; sum[1][0] = 0;

    int v;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < n; j++) {
            cin >> v; v--;
            sum[i][j+1] = sum[i][j] + v;
        }
    }

    int pos_a = n, pos_b = n;

    // Build memory matrix
    vector<vector<int> > memory(n+1, vector<int>(n+1, -1));
    memory[0][0] = 0;

    int cost = dp(sum, pos_a, pos_b, memory);
    cout << cost << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        dhl();
    }
}
```

Sweepers

Keywords— Maximum Flow, Connected Components

```

#include <iostream>
#include <map>
#include <algorithm>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/connected_components.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::edge_iterator              EdgeIt;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    Edge addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
        return e;
    }
};


void sweepers() {
    int n, m, s, p;
    cin >> n >> m >> s;

    Graph G(n+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);
    Vertex v_source = n, v_target = n + 1;

    vector<int> degree(n, 0);
    vector<bool> has_edges(n, false);
    has_edges[n] = true; has_edges[n+1];

    for (int i = 0; i < s; i++) {
        cin >> p;
        eaG.addEdge(v_source, p, 1);
        degree[p]++;
        has_edges[p] = true;
```

```
    }
    for (int i = 0; i < s; i++) {
        cin >> p;
        eaG.addEdge(p, v_target, 1);
        degree[p]++;
        has_edges[p] = true;
    }

    int x, y;
    for (int i = 0; i < m; i++) {
        cin >> x >> y;
        eaG.addEdge(x, y, 1);
        eaG.addEdge(y, x, 1);
        degree[x]++; degree[y]++;
        has_edges[x] = true; has_edges[y] = true;
    }

    int flow = push_relabel_max_flow(G, v_source, v_target);

    vector<Vertex> componentmap(n+2);
    connected_components(G, make_iterator_property_map(componentmap.begin(), get(vertex_index, G))
        );

    bool result = flow == s;        // First check if the flow is equal the number of sweepers
    // Now check the degree of all the nodes
    for (int i = 0; i < n; i++)
        if (degree[i] % 2 != 0) result = false;
    // Now check that all the nodes from the same component that source, are the ones that have
        the
    // edges -> the edges are reachable.
    int ref_component = componentmap[n];
    for (int i = 0; i < n; i++) {
        if (has_edges[i] && componentmap[i] != ref_component) result = false;
    }

    if (result) cout << "yes" << endl;
    else cout << "no" << endl;

}

int main() {
    ios_base::sync_with_stdio(false);
    int T; cin >> T;
    for (int t=0; t < T; t++){
        sweepers();
    }
}
```

Portfolios revisited

Keywords— LP/QP

```
#include <iostream>
#include <cassert>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpz.h>

// choose exact integral type
typedef CGAL::Gmpz ET;

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;


void portfolios(int n, int m){

    // Quadratic Programming
    Program qp(CGAL::SMALLER, true, 0, false, 0);
    const int COST = 0;
    const int RETURN = 1;
    int c, r, v;
    vector<int> returns(n);

    qp.set_r(RETURN, CGAL::LARGER);
    for (int i = 0; i < n; i++) {
        cin >> c >> r;
        qp.set_a(i, COST, c);
        qp.set_a(i, RETURN, r);
        returns[i] = r;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> v;
            if (j <= i) qp.set_d(i, j, 2*v);
        }
    }
    for (int i = 0; i < m; i++) {
        long C, V;
        cin >> C >> V;

        qp.set_b(COST, C);

        int lower = 0, upper = INT_MAX, middle;
        double best_return = 0;
        // Binary search
        while (lower + 1 != upper) {
            middle = lower + (upper - lower) / 2;
            qp.set_b(RETURN, middle);

            Solution s = CGAL::solve_nonnegative_quadratic_program(qp, ET());
            if (s.is_optimal() && CGAL::to_double(s.objective_value()) <= V) {
                lower = middle;
                // Compute return
                double total_return = 0;
                int indx = 0;
                for (auto it = s.variable_values_begin(); it != s.variable_values_end(); it++) {
                    total_return += returns[indx++] * CGAL::to_double(*it);
                }
                best_return = max(best_return, total_return);
            } else {
                upper = middle;
            }
        }
        cout << best_return << endl;
    }


}


int main() {
    ios_base::sync_with_stdio(false);
    int n, m;
```

```
78      while (true) {
79          cin >> n >> m;
80          if (n == 0 && m == 0) { break; }
81          portfolios(n, m);
82      }
83  }
```

The Phantom Menace

Keywords— Maximum Flow

```
/******
Given a directed graph find how many vertex need to be removed to disconect one set of vertex with another set.
This is solve creating a graph with an income and outcome node for each vertex. All the edges must have capacity one and then finding the mincut of the graph is possible to find the number of vertex
required to disconnect the two sets of vertex. The mincut is found computing the maximum flow.
******/

// Includes
// ========
// STL includes
#include <iostream>
#include <cstdlib>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace boost;
using namespace std;

// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt;

// Custom Edge Adder Class, that holds the references
// to the graph, capacity map, weight map and reverse edge map
// ===============================================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};


void phantom_menace() {
    int n, m, s, d;
    cin >> n >> m >> s >> d;

    // Create Graph and Maps
    Graph G(2*n+2);     // Input nodes: [0, n)  Output nodes: [n, 2*n)
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    Vertex src = 2 * n, sink = 2 * n + 1;

    // Reading graphs edges
```

```
    int i, j;
    for (int k = 0; k < m; k++) {
        cin >> i >> j;
        eaG.addEdge(n + i, j, 1);
    }
    for (int k = 0; k < n; k++) eaG.addEdge(k, n + k, 1);
    // Read starting vertex
    for (int k = 0; k < s; k++) {
        cin >> i;
        eaG.addEdge(src, i, 1);
    }
    // Read ending vertex
    for (int k = 0; k < d; k++) {
        cin >> j;
        eaG.addEdge(n + j, sink, 1);
    }
    // Compute the min cut as max flow
    int flow = push_relabel_max_flow(G, src, sink);
    cout << flow << endl;

}


int main() {
    int T; cin >> T;
    for (int i = 0; i < T; i++) {
        phantom_menace();
    }
}
```

Cantonal Courier

Keywords— Maximum Flow

```cpp
#include <vector>
#include <iostream>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;


// BGL Graph definitions
// =====================
// Graph Type with nested interior edge properties for Flow Algorithms
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
// Interior Property Maps
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;


// Custom Edge Adder Class, that holds the references
// to the graph, capacity map and reverse edge map
// ===============================================
class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    ReverseEdgeMap  &revedgemap;

public:
    // to initialize the Object
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, ReverseEdgeMap &revedgemap):
        G(G), capacitymap(capacitymap), revedgemap(revedgemap){}

    // to use the Function (add an edge)
    void addEdge(int from, int to, long capacity) {
        Edge e, reverseE;
        bool success;
        tie(e, success) = add_edge(from, to, G);
        tie(reverseE, success) = add_edge(to, from, G);
        capacitymap[e] = capacity;
        capacitymap[reverseE] = 0;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};

void courier() {
    int Z, J; cin >> Z >> J;

    Graph G(Z + J);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    // ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, revedgemap);

    Vertex src = add_vertex(G), sink = add_vertex(G);

    // Read zone costs
    int c;
    for (int i = 0; i < Z; i++) {
        cin >> c;
        eaG.addEdge(J+i, sink, c);
    }

    // Read jobs profits
    int total_revenue = 0;
    for (int i = 0; i < J; i++) {
        cin >> c;
        eaG.addEdge(src, i, c);
        total_revenue += c;
    }
```

```
78
79      // Read jobs and zones connections
80      for (int i = 0; i < J; i++) {
81          int N; cin >> N;
82          for (int j = 0; j < N; j++) {
83              int z; cin >> z;
84              eaG.addEdge(i, J+z, INT_MAX);
85          }
86      }
87
88      long flow = push_relabel_max_flow(G, src, sink);
89
90      cout << total_revenue - flow << endl;
91  }
92
93  int main() {
94      ios_base::sync_with_stdio(false);
95      int T; cin >> T;
96      for (int t = 0; t < T; t++) {
97          courier();
98      }
99  }
```

Index