AlgoLab AS 2014

January 18, 2015

Contents

1 ACM

1.1 Even Pairs

Keywords: Scanline

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

using namespace std;

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

    vector<int> elems(n);
    for(int i = 0; i < n; i++) {
        cin >> elems.at(i);
    }

    int even = 0;
    for(int start = 0; start < n; start++) {
        for(int end = start; end < n; end++) {
            int sum = 0;
            for(int i = start; i <= end; i++) {
                sum += elems.at(i);
            }

            if(sum % 2 == 0) {
                even++;
            }
        }
    }

    cout << even << endl;

    return 0;
}
```

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

using namespace std;

int main () {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int n;
  cin >> n;

  int nr_odd_input = 0;
  int even = 1;
  int odd = 0;
  int total_pairs = 0;
  short next_number;
  for (int i = 0; i < n; i++) {
    cin >> next_number;
    if (next_number == 1) {
      nr_odd_input++;
    }
    if ((nr_odd_input & 0x1) == 0) {
      //cout << "even: " << even << endl;
      total_pairs += even++;
    } else {
      //cout << "odd: " << odd << endl;
      total_pairs += odd++;
    }
  }

  cout << total_pairs << endl;
  return 0;
}
```

1.2 Build The Sum

Keywords:

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

using namespace std;

int main(void)
{
    int n;
    cin >> n; // ignore this

    while(cin >> n) {
        float sum = 0.0;
```

```
        for(int i=0; i < n; i++) {
            float v = 0.0;
            cin >> v;
            sum += v;
        }
        cout << sum << endl;
    }

    return 0;
}
```

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

using namespace std;

int main () {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int m;
    cin >> m;

    double sum = 0, c = 0, summand;
    for (int i = 0; i < m; i++) {
      cin >> summand;
      double y = summand - c;
      double t = sum + y;
      c = (t - sum) - y;
      sum = t;
    }

    cout << sum << endl;
  }

  return 0;
}
```

1.3 Shelves

Keywords:

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

using namespace std;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int N;
    cin >> N;

    for(int i = 0; i < N; i++) {
        int length, large_shelve_length, small_shelve_length;
        cin >> length;
        cin >> small_shelve_length;
        cin >> large_shelve_length;

        int opt_small_shelves = 0;
        int opt_large_shelves = 0;
        int opt_wall_left = length;

        int sqrt_len = (int)sqrt(length);
        if(large_shelve_length > sqrt_len) {
            for(int large_shelves = min(sqrt_len, (int)(length / large_shelve_length)); large_shelves >= 0; large_shelves
                --) {
                int wall_left_for_small = (length - (large_shelves * large_shelve_length));
                int wall_left = wall_left_for_small % small_shelve_length;
                if(wall_left < opt_wall_left) {
                    opt_wall_left = wall_left;
                    opt_small_shelves = (int)(wall_left_for_small / small_shelve_length);
                    opt_large_shelves = large_shelves;
                }

                if(opt_wall_left == 0) {
                    break;
                }
            }
        } else {
            int max_small = min(sqrt_len, (int)(length / small_shelve_length));
            for(int small_shelves = 0; small_shelves <= max_small; small_shelves++) {
                int wall_left_for_large = (length - (small_shelves * small_shelve_length));
                int wall_left = wall_left_for_large % large_shelve_length;
                if(wall_left < opt_wall_left) {
```

```
                opt_wall_left = wall_left;
                opt_small_shelves = small_shelves;
                opt_large_shelves = (int)(wall_left_for_large / large_shelve_length);
            }

            if(opt_wall_left == 0) {
                break;
            }
        }
    }

    cout << opt_small_shelves << " " << opt_large_shelves << " " << opt_wall_left << endl;
  }

  return 0;
}
```

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

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    unsigned int total_length, small_length, large_length;
    cin >> total_length >> small_length >> large_length;

    unsigned int best_nr_small_shelves = 0, best_nr_large_shelves = 0, best_uncovered_wall = total_length;
    int min_large_shelves = 0;
    if (((long long)small_length)*((long long)large_length) < (long long)numeric_limits<int>::max() && small_length*
        large_length <= total_length) {
      min_large_shelves = total_length / large_length - small_length;
    }

    for (int nr_large_shelves = total_length / large_length; nr_large_shelves >= min_large_shelves; nr_large_shelves--) {
      unsigned int uncovered_wall_large = total_length - (large_length*nr_large_shelves);
      unsigned int nr_small_shelves = uncovered_wall_large / small_length;
      unsigned int uncovered_wall = uncovered_wall_large - (small_length*nr_small_shelves);
      if (uncovered_wall < best_uncovered_wall) {
        best_uncovered_wall = uncovered_wall;
        best_nr_large_shelves = nr_large_shelves;
        best_nr_small_shelves = nr_small_shelves;
        if (best_uncovered_wall == 0) {
          break;
        }
      }
    }

    cout << best_nr_small_shelves << " " << best_nr_large_shelves << " " << best_uncovered_wall << endl;
  }

  return 0;
}
```

1.4 Checking Change

Keywords: Dynamic Programming

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

using namespace std;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int n;
    cin >> n;

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

        vector<int> coins(c);
        for(int j = 0; j < c; j++) {
            cin >> coins.at(j);
        }

        vector<int> ret(m);
        int max_ret = 0;
```

```
        for(int j = 0; j < m; j++) {
            int ret_val;
            cin >> ret_val;

            ret.at(j) = ret_val;
            if(ret_val > max_ret) {
                max_ret = ret_val;
            }
        }

        vector<int> table(max_ret + 1);
        table.at(0) = 0;

        for(int v = 1; v <= max_ret; v++) {
            table.at(v) = INT_MAX;
            for(auto coin : coins) {
                if(v - coin >= 0 && coin <= v) {
                    if(table.at(v-coin) != INT_MAX) {
                        table.at(v) = min(table.at(v), table.at(v-coin) + 1);
                    }
                }
            }
        }

        for(auto r : ret) {
            if(table.at(r) == INT_MAX) {
                cout << "not possible" << endl;
            } else {
                cout << table.at(r) << endl;
            }
        }
    }

    return 0;
}
```

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

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_coins, nr_test_values;
    cin >> nr_coins >> nr_test_values;

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

    int max_test_value = 0;
    vector<int> test_values(nr_test_values);
    for (int test_value_id = 0; test_value_id < nr_test_values; test_value_id++) {
      cin >> test_values[test_value_id];
      if (test_values[test_value_id] > max_test_value) {
        max_test_value = test_values[test_value_id];
      }
    }

    vector<int> nr_coins_for_values(max_test_value + 1, -1);
    nr_coins_for_values[0] = 0;
    for (int value = 0; value <= max_test_value; value++) {
      int nr_coins_for_value = nr_coins_for_values[value];
      if (nr_coins_for_value != -1) {
        for (int coin_id = 0; coin_id < nr_coins; coin_id++) {
          int new_value = value + coins[coin_id];
          if (new_value <= max_test_value) {
            if (nr_coins_for_values[new_value] == -1) {
              nr_coins_for_values[new_value] = nr_coins_for_value + 1;
            }
            else {
              nr_coins_for_values[new_value] = min(nr_coins_for_values[new_value], nr_coins_for_value + 1);
            }
          }
        }
      }
    }


    for (int test_value_id = 0; test_value_id < nr_test_values; test_value_id++) {
      int nr_coins_for_value = nr_coins_for_values[test_values[test_value_id]];
      if (nr_coins_for_value == -1) {
        cout << "not possible" << endl;
      }
      else {
        cout << nr_coins_for_value << endl;
```

```
      }
    }
  }

  return 0;
}
```

1.5 Even Matrices

Keywords:

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

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int n;
    cin >> n;
    vector<vector<bool> > matrix(n, vector<bool>(n, 0));
    for (int i = 0; i < n; i++) {
      vector<bool>& row = matrix[i];
      for (int j = 0; j < n; j++) {
        short input;
        cin >> input;
        row[j] = (input == 1);
      }
    }

    int even = 0;
    for (int row_i = 0; row_i < n; row_i++) {
      vector<bool> previous(n, 0);
      for (int row_j = row_i; row_j < n; row_j++) {
        for (int i = 0; i < n; i++) {
          previous[i] = previous[i] != matrix[row_j][i];
          //cout << matrix[row_j][i] << " ";
        }
        //cout << endl;
        int nr_odd_input = 0;
        int e = 1;
        int o = 0;
        for (int i = 0; i < n; i++) {
          if (previous[i] == 1) {
            nr_odd_input++;
          }
          if ((nr_odd_input & 0x1) == 0) {
            //cout << "even: " << even << endl;
            even += e++;
          }
          else {
            //cout << "odd: " << odd << endl;
            even += o++;
          }
        }
      }
    }
    cout << even << endl;

  }

  return 0;
}
```

1.6 Race Tracks

Keywords: BFS (Graph)

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>

using namespace std;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
```

```
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int width_x, height_y;
        cin >> width_x;
        cin >> height_y;

        int start_x, start_y;
        cin >> start_x;
        cin >> start_y;

        int end_x, end_y;
        cin >> end_x;
        cin >> end_y;

        int obstacle_count;
        cin >> obstacle_count;
        vector<vector<bool>> obstacles(width_x, vector<bool>(height_y, false));
        for(int j = 0; j < obstacle_count; j++) {
            int x1, x2, y1, y2;
            cin >> x1;
            cin >> y1;
            cin >> x2;
            cin >> y2;

            for(int k = x1; k <= x2; k++) {
                for(int l = y1; l <= y2; l++) {
                    obstacles.at(k).at(l) = true;
                }
            }
        }

        // check
        /*cout << "width: " << width_x << " & height: " << height_y << endl;
        cout << "start: x=" << start_x << ", y=" << start_y << endl;
        cout << "end: x=" << end_x << ", y=" << end_y << endl;
        cout << "obstacles:" << endl;
        for(int y = 0; y < height_y; y++) {
            for(int x = 0; x < width_x; x++) {
                if(obstacles.at(x).at(y)) {
                    cout << "X";
                } else if(start_x == x && start_y == y) {
                    cout << "S";
                } else if(end_x == x && end_y == y) {
                    cout << "E";
                } else {
                    cout << "O";
                }
            }
            cout << endl;
        }
        cout << "-----------------------" << endl << endl;*/

        // algo start
        //           x , y  , v_x, v_y, depth
        queue<tuple<int, int, int, int, int>> next_tiles;
        next_tiles.push(make_tuple(start_x, start_y, 0, 0, 0));
        bool found = false;
        vector<vector<vector<vector<bool>>>> visited(width_x, vector<vector<vector<bool>>>(height_y, vector<vector<bool
            >>(7, vector<bool>(7, false))));
        while(!next_tiles.empty()) {
            auto tile = next_tiles.front();
            next_tiles.pop();
            int x = get<0>(tile);
            int y = get<1>(tile);
            int v_x = get<2>(tile);
            int v_y = get<3>(tile);
            int depth = get<4>(tile);

            if(x < 0 || x >= width_x ||
                y < 0 || y >= height_y ||
                v_x < -3 || v_x > 3 ||
                v_y < -3 || v_y > 3 ||
                obstacles.at(x).at(y) ||
                visited.at(x).at(y).at(v_x + 3).at(v_y + 3)) {
                //cout << "ignore: x=" << x << ", y=" << y << ", v_x=" << v_x << ", v_y=" << v_y << ", depth=" << depth
                    << endl;
                continue;
            }

            /*for(int i_y = 0; i_y < height_y; i_y++) {
                for(int i_x = 0; i_x < width_x; i_x++) {
                    if(x == i_x && y == i_y) {
                        cout << "P";
                    } else if(obstacles.at(i_x).at(i_y)) {
                        cout << "X";
                    } else if(start_x == i_x && start_y == i_y) {
                        cout << "S";
                    } else if(end_x == i_x && end_y == i_y) {
                        cout << "E";
                    } else {
                        cout << "O";
                    }
                }
                cout << endl;
            }
```

```
            cout << endl << endl;*/

            //cout << "jump: x=" << x << ", y=" << y << ", v_x=" << v_x << ", v_y=" << v_y << ", depth=" << depth << endl
                ;

            if(x == end_x && y == end_y) {
                cout << "Optimal solution takes " << depth << " hops." << endl;
                found = true;
                break;
            }

            visited.at(x).at(y).at(v_x + 3).at(v_y + 3) = true;

            next_tiles.push(make_tuple(x + v_x - 1, y + v_y - 1,    v_x - 1, v_y - 1,   depth + 1));
            next_tiles.push(make_tuple(x + v_x - 1, y + v_y,        v_x - 1, v_y,       depth + 1));
            next_tiles.push(make_tuple(x + v_x - 1, y + v_y + 1 ,   v_x - 1, v_y + 1,   depth + 1));

            next_tiles.push(make_tuple(x + v_x, y + v_y - 1,    v_x,    v_y - 1,    depth + 1));
            next_tiles.push(make_tuple(x + v_x, y + v_y,        v_x,    v_y,        depth + 1));
            next_tiles.push(make_tuple(x + v_x, y + v_y + 1,    v_x,    v_y + 1,    depth + 1));

            next_tiles.push(make_tuple(x + v_x + 1, y + v_y - 1,    v_x + 1, v_y - 1,   depth + 1));
            next_tiles.push(make_tuple(x + v_x + 1, y + v_y,        v_x + 1, v_y,       depth + 1));
            next_tiles.push(make_tuple(x + v_x + 1, y + v_y + 1,    v_x + 1, v_y + 1,   depth + 1));

        }

        if(!found) {
            cout << "No solution." << endl;
        }
    }

    return 0;
}
```

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

using namespace std;

class State {
public:
  short x, y, speed_x, speed_y;
  int hops;

  State(short x, short y, short speed_x, short speed_y, int hops)
    :x(x), y(y), speed_x(speed_x), speed_y(speed_y), hops(hops) {}
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int width, height, start_x, start_y, end_x, end_y, nr_obstacles;
    cin >> width >> height >> start_x >> start_y >> end_x >> end_y >> nr_obstacles;

    vector<vector<vector<vector<bool> > > > race_track(width, vector<vector<vector<bool> > >(height, vector<vector<bool>
        >(7, vector<bool>(7, false))));

    for (int obstacle_id = 0; obstacle_id < nr_obstacles; obstacle_id++) {
      int obstacle_x_start, obstacle_y_start, obstacle_x_end, obstacle_y_end;
      cin >> obstacle_x_start >> obstacle_y_start >> obstacle_x_end >> obstacle_y_end;

      vector<vector<bool> > obstacle_indicator(7, vector<bool>(7, true));
      for (int x = obstacle_x_start; x <= obstacle_x_end; x++) {
        for (int y = obstacle_y_start; y <= obstacle_y_end; y++) {
          race_track[x][y] = obstacle_indicator;
        }
      }
    }

    int min_hops = -1;
    if (start_x == end_x && start_y == end_y) {
      min_hops = 0;
    }
    else {
      queue<State> states;
      states.push(State(start_x, start_y, 3, 3, 0));
      race_track[start_x][start_y][3][3] = true;
      //cout << "start: " << start_x << " " << start_y << " " << end_x << " " << end_y << endl;
      //cout << "width: " << width << " height: " << height << endl;
      while (!states.empty()) {
        State current_state = states.front();
        states.pop();
        //cout << current_state.x << " " << current_state.y << " " << current_state.speed_x << " " << current_state.
            speed_y << " " << current_state.hops << endl;
        for (int new_speed_x = max(current_state.speed_x - 1, 0); new_speed_x <= min(current_state.speed_x + 1, 6);
            new_speed_x++) {
```

```
          for (int new_speed_y = max(current_state.speed_y - 1, 0); new_speed_y <= min(current_state.speed_y + 1, 6);
              new_speed_y++) {
            int new_x = current_state.x + new_speed_x - 3;
            int new_y = current_state.y + new_speed_y - 3;
            //cout << "Checking " << new_x << " " << new_y << " " << new_speed_x << " " << new_speed_y << " " <<
                current_state.hops+1 << endl;
            //cout << " " << (new_x >= 0) << " " << (new_x < width) << " " << (new_y >= 0) << " " << (new_y < height) <<
                endl;
            if (new_x >= 0 && new_x < width && new_y >= 0 && new_y < height) {
              if (!race_track[new_x][new_y][new_speed_x][new_speed_y]) {
                if (new_x == end_x && new_y == end_y) {
                  min_hops = current_state.hops + 1;
                  goto solution_found;
                }
                else {
                  race_track[new_x][new_y][new_speed_x][new_speed_y] = true;
                  //cout << "Adding " << new_x << " " << new_y << " " << new_speed_x << " " << new_speed_y << " " <<
                      current_state.hops+1 << endl;
                  states.push(State(new_x, new_y, new_speed_x, new_speed_y, current_state.hops + 1));
                }
              }
            }
          }
        }
      }
    }
  solution_found:
    if (min_hops == -1) {
      cout << "No solution." << endl;
    }
    else {
      cout << "Optimal solution takes " << min_hops << " hops." << endl;
    }
  }

  return 0;
}
```

1.7 Boats

Keywords: Custom compare, Class with compare, Uses class, Greedy

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>

using namespace std;

class Data {
public:
    int length, ring_pos, min_pos;

    Data(int length, int ring_pos) : length(length), ring_pos(ring_pos), min_pos(ring_pos) {}

    bool operator< (const Data& other) const {
        if(min_pos == other.min_pos) {
            return ring_pos > other.ring_pos;
        }
        return min_pos > other.min_pos;
    }
};

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int N;
        cin >> N;

        priority_queue<Data> boats;
        for(int j = 0; j < N; j++) {
            int l, p;
            cin >> l;
            cin >> p;
            boats.push(Data(l, p));
        }

        int last_end = boats.top().ring_pos;
        boats.pop();
        int counter = 1;
        while(!boats.empty()) {
            auto boat = boats.top();
            boats.pop();
```

```
            if(boat.min_pos - boat.length >= last_end) {
                counter++;
                last_end = boat.min_pos;
            } else if(last_end <= boat.ring_pos) {
                // boat overlaps with a previous boat, we have to move it
                boat.min_pos = last_end + boat.length;
                if(boat.min_pos >= boat.ring_pos) {
                    boats.push(boat);
                }
            }
        }

        cout << counter << endl;
    }

    return 0;
}
```

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

using namespace std;

class Boat {
public:
  int length, ring_position, min_end_position;
  Boat(int length, int ring_position)
    :length(length), ring_position(ring_position) {
    this->min_end_position = ring_position;
  }
};

class CompareBoat {
public:
  bool operator()(const Boat& lhs, const Boat& rhs) {
    if (lhs.min_end_position == rhs.min_end_position) {
      return lhs.ring_position > rhs.ring_position;
    }
    return lhs.min_end_position > rhs.min_end_position;
  }
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_wizard_boats;
    cin >> nr_wizard_boats;

    priority_queue<Boat, vector<Boat>, CompareBoat> boats;
    for (int wizard_boat_id = 0; wizard_boat_id < nr_wizard_boats; wizard_boat_id++) {
      int boat_length, ring_position;
      cin >> boat_length >> ring_position;
      boats.push(Boat(boat_length, ring_position));
    }

    int current_end = numeric_limits<int>::min();
    int max_boats = 0;
    while (!boats.empty()) {
      Boat b = boats.top();
      boats.pop();
      //cout << "Check boat " << b.min_end_position << " " << b.ring_position << " " << b.length << endl;
      if (b.min_end_position - b.length >= current_end) {
        current_end = b.min_end_position;
        max_boats++;
      }
      else {
        if (current_end <= b.ring_position) {
          b.min_end_position = current_end + b.length;
          boats.push(b);
        }
      }
    }
    cout << max_boats << endl;
  }

  return 0;
}
```

1.8 Aliens

Keywords: Custom compare, Compare function, Compare struct, Uses class, Scanline

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

```cpp
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>

using namespace std;

class Alien {
public:
    int from;
    int to;
    bool dominated;
    Alien(int from, int to) : from(from), to(to), dominated(false) {}
};

bool cmp(Alien left, Alien right) {
    if(left.from == right.from) {
        return left.to > right.to;
    } else {
        return left.from < right.from;
    }
}

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int alien_count, human_count;
        cin >> alien_count >> human_count;

        vector<Alien> aliens;

        for(int j = 0; j < alien_count; j++) {
            int p, q;
            cin >> p >> q;

            aliens.push_back(Alien(p, q));
        }

        sort(aliens.begin(), aliens.end(), cmp);

        /*for(auto a : aliens) {
            cout << "alien: [" << a.from << ", " << a.to << "]" << endl;
        }
        cout << endl << endl;
        */

        int last_human_attacked = 0;
        int last_alien_end = 0;
        int count = 0;
        bool not_all_humans = false;
        bool id = false;
        Alien last = Alien(0, 0);

        for(int cur = 0; cur < alien_count; cur++) {
            Alien &cur_alien = aliens.at(cur);

            if(cur_alien.from > last_human_attacked + 1) {
                not_all_humans = true;
                break;
            } else {
                last_human_attacked = max(cur_alien.to, last_human_attacked);
            }

            if(!id && last.from == cur_alien.from && last.to == cur_alien.to && cur_alien.to != 0) {
                count --;
                id = true;
            } else  if(cur_alien.to > last_alien_end) {
                count++;
                last = cur_alien;
                last_alien_end = cur_alien.to;
                id = false;
            }
        }

        if(not_all_humans || last_human_attacked != human_count) {
            //cout << "!0" << " last = " << last_human_attacked << endl;
            cout << "0" << endl;
        } else {
            cout << count << endl;
        }
    }

    return 0;
}
```

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

```
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;

struct RangeComparator {
  bool operator() (const pair<int, int>& lhs, const pair<int, int>& rhs) {
    return (lhs.first == rhs.first) ? lhs.second > rhs.second : lhs.first < rhs.first;
  }
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_aliens, nr_humans;
    cin >> nr_aliens >> nr_humans;

    multiset<pair<int, int>, RangeComparator> ranges;
    for (int alien_id = 0; alien_id < nr_aliens; alien_id++) {
      int first_wounded, last_wounded;
      cin >> first_wounded >> last_wounded;
      if (first_wounded != 0) {
        ranges.insert(make_pair(first_wounded, last_wounded));
      }
    }

    pair<int, int> last_pair = make_pair(0, 0);
    int nr_superior = 0;
    bool gap_found = false;
    bool found_ident = false;

    //cout << endl;
    for (multiset<pair<int, int> >::iterator it = ranges.begin(); it != ranges.end(); ++it) {
      //cout << (*it).first << " " << (*it).second << endl;
      if ((*it).second > last_pair.second) {
        if ((*it).first > last_pair.second + 1) {
          //gap in ranges found no superior alien
          gap_found = true;
          break;
        }
        last_pair = (*it);
        found_ident = false;
        nr_superior++;
      }
      else if ((*it).second == last_pair.second && (*it).first == last_pair.first && !found_ident) {
        nr_superior--;
        found_ident = true;
      }
    }
    if (!gap_found && last_pair.second == nr_humans) {
      cout << nr_superior << endl;
    }
    else {
      cout << 0 << endl;
    }
  }

  return 0;
}
```

1.9 Next Path

Keywords: Custom compare, Compare struct, BFS (Graph)

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

using namespace std;

int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        // clean up
        best_distance.clear();

        // get graph properties
        int vertex_count, edge_count;
        cin >> vertex_count >> edge_count;
```

```
        // get start and target vertices
        int start_vertex, target_vertex;
        cin >> start_vertex >> target_vertex;
        start_vertex -= 1; // input starts at 1
        target_vertex -= 1; // input starts at 1

        // read in edges
        vector<vector<int > > edges(vertex_count, vector<int>());
        for(int edge_index = 0; edge_index < edge_count; edge_index++) {
            int from, to;
            cin >> from >> to;
            from -= 1; // input starts at 1
            to -= 1; // input starts at 1

            // create edge
            edges.at(from).push_back(to);
        }

        // keeps track of the second best solution
        int second_best = -1;

        // queue for vertices to visit
        // pair.first: lenth so far
        // pair.second: next vertex
        priority_queue<pair<int, int>, vector<pair<int, int> >, PairCompare> next_moves;
        next_moves.push(make_pair(0, start_vertex));

        // keep track how many times a vertex was reached
        vector<int> visited_counters(vertex_count, 0);
        visited_counters.at(start_vertex) = 1;

        // "The Algorithm" TM
        while(!next_moves.empty()) {
            pair<int, int> cur_move = next_moves.top();
            next_moves.pop();

            // extract information from current vertex we sit on
            int vertex = cur_move.second;
            int length = cur_move.first;

            // iterate over all neighbors
            for(int neighbor_index = 0; neighbor_index < edges.at(vertex).size(); neighbor_index++) {
                int neighbor_vertex = edges.at(vertex).at(neighbor_index);

                if(visited_counters.at(neighbor_vertex) < 2) {
                    // ^- only visit neighbor if it wasn't visited already twice
                    // v- update visited counter
                    visited_counters.at(neighbor_vertex)++;

                    if(neighbor_vertex == target_vertex && visited_counters.at(neighbor_vertex) == 2) {
                        // ok, so we found our target vertex and it was already visited twice (i.e. we visit it right now
                             for the second time)
                        // we can abort early :-D
                        second_best = length + 1;
                        goto _solution; // haha
                    } else {
                        // visit neighbor
                        next_moves.push(make_pair(length + 1, neighbor_vertex));
                    }
                }
            }
        }

    _solution:
        if(second_best == -1) {
            cout << "no" << endl;
        } else {
            cout << second_best << endl;
        }

    }

  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;

struct PairComparator {
  bool operator() (const pair<int, int>& lhs, const pair<int, int>& rhs) {
    return lhs.first > rhs.first;
  }
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
```

```
  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices, nr_edges;
    cin >> nr_vertices >> nr_edges;

    int start_v, end_v;
    cin >> start_v >> end_v;
    start_v--;
    end_v--;

    vector<vector<int> > graph(nr_vertices, vector<int>());
    for (int edge_id = 0; edge_id < nr_edges; edge_id++) {
      int edge_source, edge_target;
      cin >> edge_source >> edge_target;
      graph[edge_source - 1].push_back(edge_target - 1);
    }


    //cout << endl;
    int solution = -1;
    priority_queue<pair<int, int>, vector<pair<int, int> >, PairComparator> moves;
    moves.push(make_pair(0, start_v));
    vector<int> reachable(nr_vertices, 0);
    reachable[start_v] = 1;
    while (!moves.empty()) {
      pair<int, int> move = moves.top();
      //cout << move.first << " " << move.second << endl;
      moves.pop();
      int v = move.second;
      int length = move.first;
      for (unsigned int edge_id = 0; edge_id < graph[v].size(); edge_id++) {
        int reachable_v = graph[v][edge_id];
        if (reachable[reachable_v] < 2) {
          reachable[reachable_v]++;
          if (reachable_v == end_v && reachable[reachable_v] == 2) {
            solution = length + 1;
            goto solution_found;
          }
          else {
            moves.push(make_pair(length + 1, reachable_v));
          }
        }
      }
    }
  solution_found:
    if (solution == -1) {
      cout << "no" << endl;
    }
    else {
      cout << solution << endl;
    }
  }

  return 0;
}
```

2 Dynamic Programming

2.1 Longest Path

Keywords: BFS (Graph)

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

using namespace std;

int main(void) {
    int testcases;
    cin >> testcases;

    for(int t = 0; t < testcases; t++) {
        int vertices;
        cin >> vertices;

        vector<vector<int>> graph(vertices, vector<int>());
        for(int i = 0; i < vertices - 1; i++) {
            int from, to;
            cin >> from >> to;

            graph.at(from).push_back(to);
            graph.at(to).push_back(from);
        }

        int longest = 0;
        int longest_v = -1;

        int i = 0; // random start
        stack<int> tovisit;
        vector<bool> visited(vertices, false);
        int longest_from_i = 0;

        tovisit.push(i);
        int prev = -1;
        while(!tovisit.empty()) {
            int next = tovisit.top();
            tovisit.pop();

            if(next == -1) {
                if(longest < longest_from_i) {
                    longest = longest_from_i;
                    longest_v = prev;
                    //cout << "longest_v: " << longest_v << endl;
                }

                longest_from_i--;
                //cout << " ... " << endl;
                continue;
            }

            if(!visited.at(next)) {
                prev = next;

                longest_from_i++;
                //cout << "visiting: " << next << ", depth: " << longest_from_i << endl;
                visited.at(next) = true;
                tovisit.push(-1);
                for(auto add_v : graph.at(next)) {
                    tovisit.push(add_v);
                }
            }
        }

        // --------------

        stack<int> tovisit2;
        vector<bool> visited2(vertices, false);
        int longest_from_i2 = 0;

        tovisit2.push(longest_v);

        while(!tovisit2.empty()) {
            int next = tovisit2.top();
            tovisit2.pop();

            if(next == -1) {
                longest = max(longest, longest_from_i2);
                longest_from_i2--;
                //cout << " ... " << endl;
                continue;
            }

            if(!visited2.at(next)) {
                longest_from_i2++;
                //cout << "visiting: " << next << ", depth: " << longest_from_i2 << endl;
                visited2.at(next) = true;
                tovisit2.push(-1);
                for(auto add_v : graph.at(next)) {
                    tovisit2.push(add_v);
                }
```

```
            }
        }

        cout << longest << endl;

        /*int longest = 0;
        for(int i = 0; i < vertices; i++) {
            if(graph.at(i).size() != 1) {
                continue;
            }

            stack<int> tovisit;
            vector<bool> visited(vertices, false);
            int longest_from_i = 0;

            tovisit.push(i);

            while(!tovisit.empty()) {
                int next = tovisit.top();
                tovisit.pop();

                if(next == -1) {
                    longest = max(longest, longest_from_i);
                    longest_from_i--;
                    //cout << " ... " << endl;
                    continue;
                }

                if(!visited.at(next)) {
                    longest_from_i++;
                    //cout << "visiting: " << next << ", depth: " << longest_from_i << endl;
                    visited.at(next) = true;
                    tovisit.push(-1);
                    for(auto add_v : graph.at(next)) {
                        tovisit.push(add_v);
                    }
                }
            }
        }

        cout << longest << endl;*/
    }
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices;
    cin >> nr_vertices;
    vector<set<int> > vertices(nr_vertices, set<int>());
    for (int i = 0; i < nr_vertices - 1; i++) {
      int v1, v2;
      cin >> v1 >> v2;
      vertices[v1].insert(v2);
      vertices[v2].insert(v1);
    }
    int longest_path = -1;
    queue<int> leafs;
    for (int i = 0; i < nr_vertices; i++) {
      if (vertices[i].size() == 1) {
        leafs.push(i);
      }
    }

    vector<int> path_length(nr_vertices, 0);
    while (!leafs.empty()) {
      int leaf = leafs.front();
      leafs.pop();
      if (vertices[leaf].size() == 1) {
        int parent = *(vertices[leaf].begin());
        //cout << leaf << " " << parent << " " << path_length[leaf] << endl;
        vertices[parent].erase(leaf);
        if (vertices[parent].size() == 1) {
          leafs.push(parent);
        }
        longest_path = max(longest_path, path_length[parent] + path_length[leaf] + 1);
        path_length[parent] = max(path_length[parent], path_length[leaf] + 1);
      }
    }
```

```
    cout << (longest_path + 1) << endl;
  }

  return 0;
}
```

2.2 Light Pattern

Keywords: Scanline

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>

using namespace std;

void debug_state(vector<bool> state) {
    for(int i = 0; i < state.size(); i++) {
        if(state.at(i)) {
            cout << "[X]";
        } else {
            cout << "[ ]";
        }
    }

    cout << endl << endl;
}

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int bulb_count, bulbs_in_lightpattern, pattern;
        cin >> bulb_count >> bulbs_in_lightpattern >> pattern;

        // read initial state
        vector<bool> state(bulb_count);
        for(int j = 0; j < bulb_count; j++) {
            int v;
            cin >> v;
            state.at(j) = v == 1;
        }

        //debug_state(state);
        //cout << "TO" << endl;

        // calc target state
        vector<bool> target_state(bulbs_in_lightpattern);
        for(int j = bulbs_in_lightpattern - 1; j >= 0; j--) {
            target_state.at(j) = pattern & 0x1;
            pattern = pattern >> 1;
        }

        //debug_state(target_state);

        int group_count = (int)bulb_count / bulbs_in_lightpattern;
        int swapping_all = 1;
        int single_switching = 0;
        for(int group = group_count; group > 0; group--) {
            int from = group * bulbs_in_lightpattern;
            //cout << "working on group: " << group << " from: " << from << " to: " << from + bulbs_in_lightpattern - 1
                << endl;
            int local_swapping_all = 0;
            int local_single_switching = 0;
            for(int pos = 0; pos < bulbs_in_lightpattern; pos++) {
                if(state.at(from + pos) != target_state.at(pos)) {
                    local_single_switching++;
                } else {
                    local_swapping_all++;
                }
            }
            //cout << "\t current all: " << local_swapping_all << ", current single: " << local_single_switching << endl;

            int tmp_swapped = min(
                // already swapped till at least previously visited block
                min(
                    swapping_all + local_swapping_all, // how much do we need to changes in case we are already swapped
                    swapping_all + 2 + local_single_switching // we swap again, but only till current block and do normal
                         changes
                ),
                // not swapped yet
                min(
                    single_switching + 1 + local_swapping_all, // we swap till current block and do swapping changes
                    single_switching + 1 + local_single_switching // we swap till the next block and do normal changes in
                         current block
```

```
                )
            );

        int tmp_single_change = min(
            // not swapped yet
            min(
                single_switching + local_single_switching, // we just change each bulb by itself
                single_switching + 2 + local_swapping_all // swap, change swapped, swap back
                ),
            // swapped already
            min(
                swapping_all + 1 + local_single_switching, // swap back, do normal changes
                swapping_all + 1 + local_swapping_all
                )
        );

        swapping_all = tmp_swapped;
        single_switching = tmp_single_change;

        //cout << "\tall: " << swapping_all << ", single: " << single_switching << endl;
      }

      cout << min(swapping_all, single_switching) << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;



int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_lights, nr_lights_for_pattern, pattern;
    cin >> nr_lights >> nr_lights_for_pattern >> pattern;

    int nr_patterns = nr_lights / nr_lights_for_pattern;
    vector<int> pattern_changes(nr_patterns);
    for (int i = 0; i < nr_patterns; i++) {
      int tmp_pattern = 0;
      for (int j = 0; j < nr_lights_for_pattern; j++) {
        int light;
        cin >> light;
        tmp_pattern <<= 1;
        tmp_pattern |= light;
      }
      int pattern_xor = tmp_pattern ^ pattern;
      int changes = 0;
      for (int j = 0; j < nr_lights_for_pattern; j++) {
        if ((pattern_xor & (0x1 << j)) != 0) {
          changes++;
        }
      }
      pattern_changes[i] = changes;
    }

    int seconds_swapped = 1;
    int seconds_not_swapped = 0;
    for (int i = nr_patterns - 1; i >= 0; i--) {
      int changes = pattern_changes[i];
      int swapped_changes = nr_lights_for_pattern - changes;
      int tmp_seconds_not_swapped = min(
        min(seconds_not_swapped + changes, seconds_not_swapped + 2 + swapped_changes),
        min(seconds_swapped + 1 + changes, seconds_swapped + 1 + swapped_changes));

      seconds_swapped = min(
        min(seconds_swapped + swapped_changes, seconds_swapped + 2 + changes),
        min(seconds_not_swapped + 1 + changes, seconds_not_swapped + 1 + swapped_changes));
      seconds_not_swapped = tmp_seconds_not_swapped;
      //cout << seconds_not_swapped << " " << seconds_swapped << endl;
    }

    cout << min(seconds_not_swapped, seconds_swapped) << endl;
  }

  return 0;
}
```

2.3 Burning Coins

Keywords: Dynamic Programming

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>

using namespace std;

int at_least(vector<vector<int> >& table, vector<int>& values, int left, int right)
{
    // check first if we know the value already
    if(left == right) {
        // only one coin on the table, we can choose so it's ours
        return values[left];
    }

    if(table[left][right] > 0) {
        // already calculated value, reuse
        return table[left][right];
    }

    // calculate easy cases, fast paths
    int delta = right - left;
    if(delta == 1) {
        // case: left_coin, right_coin
        int best_of_both = max(values[left], values[right]);
        table[left][right] = best_of_both;
        return best_of_both;
    } else if(delta == 2) {
        // case: left_coin, middle_coin, right_coin
        // we have to take middle coin into account and resolve all three coins
        int left_coin = values[left];
        int middle_coin = values[left+1];
        int right_coin = values[right];

        int resulting_value = -2;
        if(left_coin > right_coin) {
            // we take the left coin, now we have to take into account that opposite
            // side will take the best of middle/right coin
            resulting_value = left_coin;
            if(middle_coin > right_coin) {
                resulting_value += right_coin;
            } else {
                resulting_value += middle_coin;
            }
        } else {
            // we take right coin, again take middle coin into account
            resulting_value = right_coin;
            if(middle_coin > left_coin) {
                resulting_value += left_coin;
            } else {
                resulting_value += middle_coin;
            }
        }

        table[left][right] = resulting_value;
        return resulting_value;
    }

    /*
     * now decide what happens in general
     */

    // assume we took the left coin
    int we_left_other_left = at_least(table, values, left+2, right); // opposite side takes also the next left one
    int we_left_other_right = at_least(table, values, left+1, right-1); // opposite side takes the right one

    // assume we took the right coin
    int we_right_other_left = we_left_other_right; //at_least(table, values, left+1, right-1);
    int we_right_other_right = at_least(table, values, left, right-2); // opposite side takes the right one too

    // find the minimum value we're guaranteed to get
    int we_get_at_least = max(
        values[left] + min(we_left_other_left, we_left_other_right),
        values[right] + min(we_right_other_left, we_right_other_right)
        );
    table[left][right] = we_get_at_least;
    return we_get_at_least;
}

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int coin_amount;
```

```
        cin >> coin_amount;
        vector<int> coins;
        for(int j = 0; j < coin_amount; j++) {
            int value;
            cin >> value;
            coins.push_back(value);
        }

        vector<vector<int> > table(coin_amount + 1, vector<int>(coin_amount + 1, -1));
        cout << at_least(table, coins, 0, coin_amount - 1) << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_coins;
    cin >> nr_coins;
    vector<int> coins(nr_coins);
    for (int i = 0; i < nr_coins; i++) {
      cin >> coins[i];
    }

    vector<vector<int> > ta(nr_coins + 1, vector<int>(nr_coins + 1, 0));
    vector<vector<int> > tb(nr_coins + 1, vector<int>(nr_coins + 1, 0));

    for (int l = 1; l <= nr_coins; l++) {
      for (int i = 0; i < nr_coins; i++) {
        ta[i][l] = max(coins[i] + tb[i + 1][l - 1], coins[i + l - 1] + tb[i][l - 1]);
        tb[i][l] = min(ta[i + 1][l - 1], ta[i][l - 1]);
      }
    }

    cout << ta[0][nr_coins] << endl;
  }

  return 0;
}
```

2.4 Poker Chips

Keywords: Dynamic Programming

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>

#define EMPTY_CELL -1

using namespace std;

int round(int stack_count, vector<int>& stack_sizes, vector<vector<int> >& chips, vector<vector<vector<vector<vector<int
    > > > > >& table, vector<int>& conf) {
  //cout << "at: " << conf.at(0) << " " << conf.at(1) << " " << conf.at(2) << " " << conf.at(3) << " " << conf.at(4) <<
      endl;
  // only calculate unknown cell values
  if(table[conf.at(0)][conf.at(1)][conf.at(2)][conf.at(3)][conf.at(4)] == EMPTY_CELL) {
    int max_value = EMPTY_CELL;

    // go over possible sets of poker chip stacks
    for(int set_index = 1; set_index < 32; set_index++) {
      int color = EMPTY_CELL;
      bool valid = true;
      int removed = 0;
      vector<int> new_conf(5);

      // go over all stacks and decide if it is part of the set, if so do calculation
      for(int stack_index = 0; stack_index < 5; stack_index++) {
        if((set_index >> stack_index) & 1) { // stack is in the current set
          if(conf.at(stack_index) == 0) {
            // can't remove a chip from current stack
```

```
            valid = false;
            break;
          }

          if(color == EMPTY_CELL) {
            // set current color
            color = chips.at(stack_index).at(conf.at(stack_index) - 1);
          } else if(color != chips.at(stack_index).at(conf.at(stack_index) - 1)) {
            // color top of current stack not the same as we selected
            valid = false;
            break;
          }

          removed++;
          new_conf.at(stack_index) = conf.at(stack_index) - 1;
        } else {
          new_conf.at(stack_index) = conf.at(stack_index);
        }
      }

      if(valid) {
        int points = 0;
        if(removed > 1) {
          points = pow(2, removed - 2);
        }

        int val = points + round(stack_count, stack_sizes, chips, table, new_conf);
        max_value = max(max_value, val);
      }
    }

    table[conf.at(0)][conf.at(1)][conf.at(2)][conf.at(3)][conf.at(4)] = max_value;
  }

  return table[conf.at(0)][conf.at(1)][conf.at(2)][conf.at(3)][conf.at(4)];
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int test_count;
  cin >> test_count;

  for(int test_index = 0; test_index < test_count; test_index++) {
    int stack_count;
    cin >> stack_count;

    // read in size of each stack
    vector<int> stack_sizes(5, 0);
    vector<int> conf(5, 0);
    for(int stack_index = 0; stack_index < stack_count; stack_index++) {
      int chip_amount;
      cin >> chip_amount;

      stack_sizes.at(stack_index) = chip_amount;
      conf.at(stack_index) = chip_amount;
    }

    // contains the chips of each stack at the start
    vector<vector<int > > chips;
    chips.reserve(stack_count);

    // read in chips in stack
    for(int stack_index = 0; stack_index < stack_count; stack_index++) {
      chips.push_back(vector<int>(stack_sizes[stack_index], -1));
      for(int chip_index = 0; chip_index < stack_sizes[stack_index]; chip_index++) {
        cin >> chips.at(stack_index).at(chip_index);
      }
    }

    // create DP table, initial value is EMPTY_CELL
    vector<vector<vector<vector<vector<int > > > > > table(stack_sizes[0] + 1,
      vector<vector<vector<vector<int > > > >(stack_sizes[1] + 1,
        vector<vector<vector<int > > >(stack_sizes[2] + 1,
          vector<vector<int > >(stack_sizes[3] + 1,
            vector<int>(stack_sizes[4] + 1, EMPTY_CELL)))));

    // initialise table
    table[0][0][0][0][0] = 0;

    // play the game
    cout << round(stack_count, stack_sizes, chips, table, conf) << endl;
  }
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

using namespace std;
```

```
int max_stacks = 5;

vector<int> stack_size;
vector<vector<int> > stacks;

vector<vector<vector<vector<vector<int> > > > > maximum;

int getMaximum(int first, int second, int third, int fourth, int fifth) {
  //cout << first << " " << second << " " << third << " " << fourth << " " << fifth << endl;
  if (maximum[first][second][third][fourth][fifth] == -1) {
    for (int i = 0; i < 2; i++) {
      if (first - i < 0) continue;
      for (int j = 0; j < 2; j++) {
        if (second - j < 0) continue;
        for (int k = 0; k < 2; k++) {
          if (third - k < 0) continue;
          for (int l = 0; l < 2; l++) {
            if (fourth - l < 0) continue;
            for (int m = 0; m < 2; m++) {
              if (fifth - m < 0) continue;
              int chips_to_remove = i + j + k + l + m;
              if (chips_to_remove == 0) continue;

              int chip = -1;
              if (i == 1) {
                //cout << "Setting chip to " << stacks[0][first-1] << endl;
                chip = stacks[0][first - 1];
              }
              if (j == 1) {
                if (chip == -1) {
                  chip = stacks[1][second - 1];
                }
                else {
                  //cout << "Compare to " << stacks[1][second-1] << endl;
                  if (chip != stacks[1][second - 1]) continue;
                }
              }
              if (k == 1) {
                if (chip == -1) {
                  chip = stacks[2][third - 1];
                }
                else {
                  if (chip != stacks[2][third - 1]) continue;
                }
              }
              if (l == 1) {
                if (chip == -1) {
                  chip = stacks[3][fourth - 1];
                }
                else {
                  if (chip != stacks[3][fourth - 1]) continue;
                }
              }
              if (m == 1) {
                if (chip == -1) {
                  chip = stacks[4][fifth - 1];
                }
                else {
                  if (chip != stacks[4][fifth - 1]) continue;
                }
              }
              //cout << "Maximum for: " << first-i << " " << second-j << " " << third-k << " " << fourth-l << " " <<
                  fifth-m << endl;
              int newMax = getMaximum(first - i, second - j, third - k, fourth - l, fifth - m);
              if (chips_to_remove > 1) {
                //cout << "Updataing for: " << first << " " << second << " " << third << " " << fourth << " " << fifth <<
                     endl;
                //cout << "By: " << (1 << (chips_to_remove-2)) << endl;
                newMax += 1 << (chips_to_remove - 2);
              }
              maximum[first][second][third][fourth][fifth] = max(maximum[first][second][third][fourth][fifth], newMax);
            }
          }
        }
      }
    }
  }
  return maximum[first][second][third][fourth][fifth];
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_stacks;
    cin >> nr_stacks;
    stack_size = vector<int>(max_stacks);
    for (int i = 0; i < nr_stacks; i++) {
      cin >> stack_size[i];
    }
    for (int i = nr_stacks; i < max_stacks; i++) {
```

```
      stack_size[i] = 0;
    }

    stacks = vector<vector<int> >(max_stacks);
    for (int i = 0; i < nr_stacks; i++) {
      stacks[i] = vector<int>(stack_size[i]);
      for (int j = 0; j < stack_size[i]; j++) {
        cin >> stacks[i][j];
      }
    }
    for (int i = nr_stacks; i < max_stacks; i++) {
      stacks[i] = vector<int>();
    }

    maximum = vector<vector<vector<vector<vector<int> > > > >(
      stack_size[0] + 1,
      vector<vector<vector<vector<int> > > >(
      stack_size[1] + 1,
      vector<vector<vector<int> > >(
      stack_size[2] + 1,
      vector<vector<int> >(
      stack_size[3] + 1,
      vector<int>(stack_size[4] + 1, -1)))));
    maximum[0][0][0][0][0] = 0;
    cout << getMaximum(stack_size[0], stack_size[1], stack_size[2], stack_size[3], stack_size[4]) << endl;
  }

  return 0;
}
```

3 BGL Introduction

3.1 Building a Graph

Keywords: Shortest path, Spanning tree, Graph with edge weight map

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

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

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    //read graph
    int nr_vertices, nr_edges;
    cin >> nr_vertices >> nr_edges;

    Graph g(nr_vertices);
    property_map<Graph, edge_weight_t>::type weight_map = get(edge_weight, g);
    for (int i = 0; i < nr_edges; i++) {
      int v1, v2, weight;
      cin >> v1 >> v2 >> weight;
      bool success;
      Edge e;
      tie(e, success) = add_edge(v1, v2, g);
      weight_map[e] = weight;
    }

    vector<Edge> spanning_tree;
    kruskal_minimum_spanning_tree(g, back_inserter(spanning_tree));

    int total_spanning_tree_edge_weight = 0;
    for (vector<Edge>::iterator it = spanning_tree.begin(); it != spanning_tree.end(); ++it) {
      total_spanning_tree_edge_weight += get(weight_map, *it);
    }

    vector<int> distances(num_vertices(g));
    dijkstra_shortest_paths(g, 0, distance_map(&distances[0]));
    int max_distance = 0;
    for (int i = 0; i < distances.size(); i++) {
      max_distance = max(max_distance, distances[i]);
    }

    cout << total_spanning_tree_edge_weight << " " << max_distance << endl;
  }

  return 0;
}
```

3.2 Ant Challenge

Keywords: Spanning tree, Shortest path, Graph with edge index map

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

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

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_index_t, int> > Graph;
```

```cpp
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int vertices, edges, species, start, end;
        cin >> vertices;
        cin >> edges;
        cin >> species;
        cin >> start;
        cin >> end;

        Graph graph(vertices);
        property_map<Graph, edge_index_t>::type indices = get(edge_index, graph);

        vector<vector<int > > weights(species, vector<int>(edges, -1));

        for(int j = 0; j < edges; j++) {
            int from, to;
            cin >> from;
            cin >> to;

            // add edge
            bool success;
            Edge edge;
            tie(edge, success) = add_edge(from, to, graph);
            indices[edge] = j;

            for(int k = 0; k < species; k++) {
                int species_weight;
                cin >> species_weight;

                // set weight
                weights[k][j] = species_weight;
            }
        }

        // no reason to know where the hives are...
        int ignore;
        for(int k = 0; k < species; k++) {
            cin >> ignore;
        }

        // find minimum spanning for each species
        vector<int> spanning_tree_weights(edges, numeric_limits<int>::max());
        for(int k = 0; k < species; k++) {
            vector<Edge> spanning_tree;
            kruskal_minimum_spanning_tree(graph, back_inserter(spanning_tree), weight_map(make_iterator_property_map(
                weights[k].begin(), indices)));

            for(vector<Edge>::iterator spanning_tree_edge = spanning_tree.begin(); spanning_tree_edge != spanning_tree.
                end(); ++spanning_tree_edge) {
                if(weights[k][indices[*spanning_tree_edge]] < spanning_tree_weights[indices[*spanning_tree_edge]]) {
                    spanning_tree_weights[indices[*spanning_tree_edge]] = weights[k][indices[*spanning_tree_edge]];
                }
            }
        }

        // now we have a minimal spanning tree we can use to find the shortest path from start to end
        vector<Vertex> predecessors(num_vertices(graph));
        vector<int> distances(num_vertices(graph));
        dijkstra_shortest_paths(graph, start, predecessor_map(&predecessors[0]).distance_map(&distances[0]).weight_map(
            make_iterator_property_map(&spanning_tree_weights[0], indices)));

        // ok, now we can read out the path length
        cout << distances[end] << endl;
    }

    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>

using namespace std;
using namespace boost;
```

```
typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_index_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

/*
typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_weight_t, vector<int> > > Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;


struct get_weight {
  int species_id;
  get_weight(int species_id) : species_id(species_id) {}

  typedef int result_type;
  int operator()(vector<int> x) const {return x[0];}
};

class record_edges : public dijkstra_visitor<>
{
  public:
  record_edges(map<int, Edge>& edges)
  : edges(edges) { }

  template <class Edge, class Graph>
  void edge_relaxed(Edge e, Graph& g) {
    cout << "bla" << endl;
    edges[target(e, g)] = e;
  }
  protected:
  map<int, Edge>& edges;
};

struct VertexInformation
{
  typedef boost::vertex_property_type type;
};

record_edges use_edges_visitor(use_edges);
dijkstra_shortest_paths(g, hives[i], weight_map(make_transform_value_property_map(get_weight(i), get(edge_weight, g))).
     visitor(use_edges_visitor));
*/

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices, nr_edges, nr_species, start_edge, end_edge;
    cin >> nr_vertices >> nr_edges >> nr_species >> start_edge >> end_edge;

    Graph g(nr_vertices);
    property_map<Graph, edge_index_t>::type ind = get(edge_index, g);
    vector<vector<int> > graph_species_weights(nr_species, vector<int>(nr_edges));
    for (int i = 0; i < nr_edges; i++) {
      int v1, v2;
      vector<int> weights(nr_species);
      cin >> v1 >> v2;
      for (int j = 0; j < nr_species; j++) {
        cin >> graph_species_weights[j][i];
      }
      bool success;
      Edge e;
      tie(e, success) = add_edge(v1, v2, g);
      ind[e] = i;
    }

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

    vector<int> edge_weights(nr_edges, numeric_limits<int>::max());
    for (int i = 0; i < nr_species; i++) {
      vector<Edge> spanning_tree;
      kruskal_minimum_spanning_tree(g, back_inserter(spanning_tree), weight_map(make_iterator_property_map(
          graph_species_weights[i].begin(), ind)));
      for (vector<Edge>::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei) {
        edge_weights[ind[*ei]] = min(edge_weights[ind[*ei]], graph_species_weights[i][ind[*ei]]);
      }
    }

    vector<int> distances(num_vertices(g), numeric_limits<int>::max());
    dijkstra_shortest_paths(g, start_edge, distance_map(&distances[0]).weight_map(make_iterator_property_map(&
        edge_weights[0], ind)));

    cout << distances[end_edge] << endl;
  }

  return 0;
}
```

3.3 Important Bridges

Keywords: Articulation points, Biconnected components

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

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

#define ISLAND 1
#define BRIDGE 2

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        // read test case
        int islands, bridges;
        cin >> islands;
        cin >> bridges;

        // create graph for test case
        Graph graph(islands + bridges);

        // keep track which vertex is what
        vector<int> vertex_type(islands + bridges, -1);

        // keep track of bridges trough island pairs
        vector<pair<int, int> > bridge_to_island(bridges, make_pair(-1, -1));

        // set vertex properties
        for(int bridge_vertex = 0; bridge_vertex < bridges; bridge_vertex++) {
            assert(vertex_type.at(bridge_vertex) == -1);
            vertex_type.at(bridge_vertex) = BRIDGE;
        }

        for(int island_vertex = bridges + 1; island_vertex < bridges+islands; island_vertex++) {
            assert(vertex_type.at(island_vertex) == -1);
            vertex_type.at(island_vertex) = ISLAND;
        }

        // read in bridges
        for(int k = 0; k < bridges; k++) {
            int from_island, to_island;
            cin >> from_island;
            cin >> to_island;

            // each bridge is also a vertices and connects both islands (also vertices) through edges
            // calculate vertices indexes first
            int from_island_index = bridges + from_island - 1; // input starts at 1, we start at 0
            int to_island_index = bridges + to_island - 1; // input starts at 1, we start at 0
            int bridge_index = k;

            // add edges:
            //          bridge
            //          /    \
            // from island  to island
            bool success;
            Edge edge;
            tie(edge, success) = add_edge(from_island_index, bridge_index, graph);
            tie(edge, success) = add_edge(to_island_index, bridge_index, graph);

            // keep track of islands the bridge connects
            bridge_to_island.at(bridge_index) = make_pair(from_island, to_island);
        }

        // find the important bridges
        vector<Vertex> vertices;
        articulation_points(graph, back_inserter(vertices));

        set<pair<int, int>> out_bridges;
        for(vector<Vertex>::iterator v2 = vertices.begin(); v2 != vertices.end(); ++v2) {
            //cout << "type: " << vertex_type[*v2] << " edge nr: " << *v2 << endl;
            // only if articulation point is a bridge, we're interested because it's an important bridge!
            if(vertex_type[*v2] == BRIDGE) {
                pair<int, int> islands_connected_by_important_bridge = bridge_to_island.at(*v2);
```

```
                out_bridges.insert(make_pair(min(islands_connected_by_important_bridge.first,
                    islands_connected_by_important_bridge.second), max(islands_connected_by_important_bridge.first,
                    islands_connected_by_important_bridge.second)));
            }
        }

        /*// print out the edges, sorted order
        set<pair<int, int>> out_bridges;
        for(vector<Vertex>::iterator v2 = vertices.begin(); v2 != vertices.end(); ++v2) {
            //cout << "city: " << *v2 << endl;
            //sort(island_to_bridge[*v2].begin(), island_to_bridge[*v2].end());
            for(vector<pair<int, int> >::iterator bridge_islands = island_to_bridge[*v2].begin(); bridge_islands !=
                island_to_bridge[*v2].end(); ++bridge_islands) {
                //cout << (*bridge_islands).first << " " << (*bridge_islands).second << endl;
                out_bridges.insert(make_pair(min((*bridge_islands).first, (*bridge_islands).second), max((*bridge_islands
                    ).first, (*bridge_islands).second)));
            }
        }*/

        cout << out_bridges.size() << endl;
        for(pair<int, int> bout : out_bridges) {
            cout << bout.first << " " << bout.second << endl;
        }
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/biconnected_components.hpp>

using namespace std;
using namespace boost;

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

typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_component_t, size_t> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices, nr_edges;
    cin >> nr_vertices >> nr_edges;

    Graph g(nr_vertices + 1);
    for (int i = 0; i < nr_edges; i++) {
      int v1, v2;
      cin >> v1 >> v2;
      bool success;
      Edge e;
      tie(e, success) = add_edge(v1, v2, g);
    }

    property_map<Graph, edge_component_t>::type component = get(edge_component, g);
    size_t nr_components = biconnected_components(g, component);

    vector<int> component_edges(nr_components, -1);

    graph_traits<Graph>::edge_iterator ei, ei_end;
    for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
      size_t edge_component_id = component[*ei];
      if (component_edges[edge_component_id] == -1) {
        component_edges[edge_component_id] = 1;
      }
      else {
        component_edges[edge_component_id] = -2;
      }
```

```
    }

    vector<pair<int, int> > bridges = vector<pair<int, int> >();
    IndexMap index = get(vertex_index, g);
    for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
      size_t edge_component_id = component[*ei];
      if (component_edges[edge_component_id] == 1) {
        int source_ind = index[source(*ei, g)];
        int target_ind = index[target(*ei, g)];
        bridges.push_back(make_pair(min(source_ind, target_ind), max(source_ind, target_ind)));
      }
    }
    sort(bridges.begin(), bridges.end());
    cout << bridges.size() << endl;
    for (vector<pair<int, int> >::iterator b_it = bridges.begin(); b_it != bridges.end(); ++b_it) {
      cout << b_it->first << " " << b_it->second << endl;
    }
  }


  return 0;
}
```

3.4 Shy Programmers

Keywords: Planarity

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        // read test case
        int employee_count, friendship_count;
        cin >> employee_count;
        cin >> friendship_count;

        // create graph for test case
        Graph graph(employee_count + 1);

        // add personal door edge
        for(int k = 0; k < employee_count; k++) {
            bool success;
            Edge edge;
            tie(edge, success) = add_edge(k, employee_count + 1, graph);
        }

        // add friendship connections
        for(int k = 0; k < friendship_count; k++) {
            int friend_A, friend_B;
            cin >> friend_A;
            cin >> friend_B;

            bool success;
            Edge edge;
            tie(edge, success) = add_edge(friend_A, friend_B, graph);
        }

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

    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices, nr_edges;
    cin >> nr_vertices >> nr_edges;

    Graph g(nr_vertices + 1);
    for (int i = 0; i < nr_edges; i++) {
      int v1, v2;
      cin >> v1 >> v2;
      bool success;
      Edge e;
      tie(e, success) = add_edge(v1, v2, g);
    }

    for (int i = 0; i < nr_vertices; i++) {
      bool success;
      Edge e;
      tie(e, success) = add_edge(i, nr_vertices, g);
    }

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

  return 0;
}
```

3.5 Fluid Borders

Keywords: Planarity

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
```

```
        int meta_blob_count;
        cin >> meta_blob_count;

        // create graph
        Graph graph(meta_blob_count);

        // read in election results
        vector<vector<int> > election_results(meta_blob_count, vector<int>(meta_blob_count - 1, -1));
        for(int voting_meta_blob = 0; voting_meta_blob < meta_blob_count; voting_meta_blob++) {
            for(int vote_place_index = 0; vote_place_index < meta_blob_count - 1; vote_place_index++) {
                // read in the meta-blob number that is at `vote_place_index` place in the election of meta-blob `
                    voting_meta_blob`
                cin >> election_results.at(voting_meta_blob).at(vote_place_index);
            }
        }

        int t = 0; // needed as result later
        for(; t < meta_blob_count - 1; t++) {
            for(int voting_meta_blob = 0; voting_meta_blob < meta_blob_count; voting_meta_blob++) {
                // add edge for current place under check `t`
                Edge edge;
                tie(edge, tuples::ignore) = add_edge(voting_meta_blob, election_results.at(voting_meta_blob).at(t), graph
                    );
            }

            // check if it's possible
            if(!boyer_myrvold_planarity_test(graph)) {
                break;
            }
        }

        cout << t << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices;
    cin >> nr_vertices;

    vector<vector<int> > election_outcomes(nr_vertices, vector<int>(nr_vertices - 1));
    for (int i = 0; i < nr_vertices; i++) {
      for (int j = 0; j < nr_vertices - 1; j++) {
        cin >> election_outcomes[i][j];
      }
    }
    int t = 0;
    Graph g(nr_vertices);
    for (int i = 0; i < nr_vertices - 1; i++) {
      for (int j = 0; j < nr_vertices; j++) {
        bool success;
        Edge e;
        tie(e, success) = add_edge(j, election_outcomes[j][i], g);
      }
      if (boyer_myrvold_planarity_test(g)) {
        t++;
      }
      else {
        break;
      }
    }

    cout << t << endl;
  }
```

```
    return 0;
}
```

4 Flows and Matchings

4.1 Buddies Selection

Keywords: Matching, Match size

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>
#include <string>

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

using namespace std;
using namespace boost;

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

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        // read test case
        int student_count, characteristic_count, min_solution;
        cin >> student_count >> characteristic_count >> min_solution;

        // create graph for test case
        Graph graph(student_count);

        // create a set of characteristics for each student from input
        map<int, set<string> > student_to_characs;
        for(int student_i = 0; student_i < student_count; student_i++) {
            set<string> charac_set;

            for(int j = 0; j < characteristic_count; j++) {
                string charac_value;
                cin >> charac_value;

                //cout << "\tbla: " << charac_value << endl;

                charac_set.insert(charac_value);
            }

            student_to_characs[student_i] = charac_set;
        }

        for(int k = 0; k < student_count; k++) {
            for(int p = k + 1; p < student_count; p++) {
                vector<string> intersec;

                set_intersection(student_to_characs.at(k).begin(), student_to_characs.at(k).end(),
                                 student_to_characs.at(p).begin(), student_to_characs.at(p).end(),
                                 back_inserter(intersec));

                //cout << "size A: " << student_to_characs.at(k).size() << endl;
                //cout << "size B: " << student_to_characs.at(p).size() << endl;
                //cout << "intersec size: " << intersec.size() << endl;
                if(intersec.size() > min_solution) {
                    //cout << "added" << endl;
                    // more in common than minimum given, add edge between students
                    bool success;
                    Edge edge;
                    tie(edge, success) = add_edge(k, p, graph);
                }
            }
        }

        // find matching
        vector<Vertex> mate(student_count);
        checked_edmonds_maximum_cardinality_matching(graph, &mate[0]);

        //cout << "size: " << matching_size(graph, &mate[0]) << endl;
        //cout << "half: " << (int)(student_count / 2) << endl;

        if(matching_size(graph, &mate[0]) == (int)(student_count / 2)) {
            cout << "not optimal" << endl;
        } else {
            cout << "optimal" << endl;
        }
    }
```

```
    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_students, nr_char, min_common_char;
    cin >> nr_students >> nr_char >> min_common_char;

    map<string, int> char_map = map<string, int>();
    vector<vector<int> > student_chars(nr_students, vector<int>(nr_char));
    int next_char_id = 0;
    for (int i = 0; i < nr_students; i++) {
      for (int j = 0; j < nr_char; j++) {
        string charact;
        cin >> charact;
        map<string, int>::iterator it = char_map.find(charact);
        int char_id;
        if (it == char_map.end()) {
          char_id = next_char_id;
          char_map[charact] = next_char_id++;
        }
        else {
          char_id = it->second;
        }
        student_chars[i][j] = char_id;
      }
    }
    for (int i = 0; i < nr_students; i++) {
      sort(student_chars[i].begin(), student_chars[i].end());
    }

    Graph g(nr_students);
    for (int i = 0; i < nr_students; i++) {
      vector<int>& char_a = student_chars[i];
      for (int j = i + 1; j < nr_students; j++) {
        vector<int>& char_b = student_chars[j];
        int common = 0;
        vector<int>::iterator a_it = char_a.begin();
        vector<int>::iterator b_it = char_b.begin();
        while (a_it != char_a.end() && b_it != char_b.end()) {
          if (*a_it == *b_it) {
            common++;
            ++a_it;
            ++b_it;
          }
          else if (*a_it < *b_it) {
            ++a_it;
          }
          else {
            ++b_it;
          }
        }
        if (common > min_common_char) {
          bool success;
          Edge e;
          tie(e, success) = add_edge(i, j, g);
        }
      }
    }

    vector<Vertex> mate(nr_students);
    edmonds_maximum_cardinality_matching(g, &mate[0]);

    bool success = true;
```

```
    for (vector<Vertex>::iterator v_it = mate.begin(); v_it != mate.end(); ++v_it) {
      if (*v_it == graph_traits<Graph>::null_vertex()) {
        success = false;
        break;
      }
    }
    if (success) {
      cout << "not optimal" << endl;
    }
    else {
      cout << "optimal" << endl;
    }
  }

  return 0;
}
```

4.2 Satellites

Keywords: Matching, DFS (Graph), Matching with DFS

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>
#include <string>

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

using namespace std;
using namespace boost;

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


// code given
typedef graph_traits<Graph>::out_edge_iterator  OutEdgeIt;

void DFS(int u, Graph &G, vector<bool> &visited, vector<Vertex> &mate) {
    OutEdgeIt ebeg, eend;
    visited[u] = true;
    for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
        const int v = target(*ebeg, G);
        //  v not vis.  && left to right with Non-Matching edges
        //             right to left with Matching edges
        if (!visited[v] && (((v == mate[u]) != (u < v)))) {
            DFS(v, G, visited, mate);
        }
    }
}
// END code given

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        // read test case
        int ground_count, satellite_count, link_count;
        cin >> ground_count >> satellite_count >> link_count;

        // create bipartit graph
        Graph graph(ground_count + satellite_count);

        // read links and create edges for them
        for(int li = 0; li < link_count; li++) {
            int ground_index, satellite_index;
            cin >> ground_index >> satellite_index;

            bool success;
            Edge edge;
            tie(edge, success) = add_edge(ground_index, satellite_index + ground_count, graph);
        }

        // find maximum matching
        vector<Vertex> mate(ground_count + satellite_count);
        checked_edmonds_maximum_cardinality_matching(graph, &mate[0]);

        // keep track of visited vertices
        vector<bool> visited(ground_count + satellite_count);
```

```
// code given
        // starting points for VC
        vector<int> startpoints;
        //cout << "Matching:" << endl;
        for (int i = 0; i < ground_count + satellite_count; ++i) {
            // output the matching
            if (mate[i] == graph_traits<Graph>::null_vertex() && i < ground_count) {
                startpoints.push_back(i);
            }
        }

        // run depth first visit
        for (int i = 0; i < startpoints.size(); ++i) {
            DFS(startpoints[i], graph, visited, mate);
        }
// END code given

        // collect unmarked
        vector<int> out_ground_ids;
        vector<int> out_satellite_ids;
        for(int index = 0; index < ground_count + satellite_count; index++) {
            if(index < ground_count && !visited[index]) {
                //cout << "ground ID: " << index << endl;
                out_ground_ids.push_back(index);
            } else if(index >= ground_count && visited[index]) {
                //cout << "satellite ID: " << index - ground_count << endl;
                out_satellite_ids.push_back(index - ground_count);
            }
        }

        // NOTE: abstand am ende kein Problem! So kompliziertes spaces einfügen nicht noetig :)
        cout << out_ground_ids.size() << " " << out_satellite_ids.size() << endl;
        bool first = true;
        for(int og : out_ground_ids) {
            if(!first) {
                 cout << " ";
            } else {
                first = false;
            }

            cout << og;
        }

        if(!first) {
            cout << " ";
        }

        first = true;
        for(int os : out_satellite_ids) {
            if(!first) {
                 cout << " ";
            } else {
                first = false;
            }

            cout << os;
        }

        cout << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> Graph;
typedef graph_traits<Graph>::vertex_descriptor  Vertex;
typedef graph_traits<Graph>::out_edge_iterator  OutEdgeIt;
const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();

void DFS(int u, Graph &G, vector<bool> &visited, vector<Vertex> &mate) {
  OutEdgeIt ebeg, eend;
  visited[u] = true;
  for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
```

```
    const int v = target(*ebeg, G);
    //  v not vis.  && left to right with Non-Matching edges
    //          right to left with Matching edges
    if (!visited[v] && (((v == mate[u]) != (u < v)))) {
      DFS(v, G, visited, mate);
    }
  }
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_ground, nr_sat, nr_links;
    cin >> nr_ground >> nr_sat >> nr_links;

    int nr_vertices = nr_ground + nr_sat;
    Graph g(nr_vertices);
    for (int i = 0; i < nr_links; i++) {
      int ground_id, sat_id;
      cin >> ground_id >> sat_id;
      add_edge(ground_id, nr_ground + sat_id, g);
    }

    // maximum matching
    vector<Vertex> mate(nr_vertices);
    edmonds_maximum_cardinality_matching(g, &mate[0]);

    // starting points for VC
    vector<int> startpoints;
    //cout << "Matching:" << endl;
    for (int i = 0; i < nr_vertices; ++i) {
      // output the matching
      if (mate[i] != NULL_VERTEX && i < nr_ground){
        //cout << i << " - " << mate[i] << endl;
        // make unmatched L vertices the root vertices of DFS,
        // i.e. "mark as visited"
      }
      else if (mate[i] == NULL_VERTEX && i < nr_ground) {
        startpoints.push_back(i);
      }
    }

    // run depth first visit
    vector<bool> visited(nr_vertices + 1);
    for (int i = 0; i < startpoints.size(); ++i) {
      DFS(startpoints[i], g, visited, mate);
    }
    // output Minimum vertex cover
    //cout << "Minimum Vertex Cover:" << endl;
    vector<int> monitored_ground;
    vector<int> monitored_station;
    for (int i = 0; i < nr_vertices; ++i) {
      if (visited[i] == 0 && i < nr_ground) {
        monitored_ground.push_back(i);
      }
      else if (visited[i] > 0 && i >= nr_ground) {
        monitored_station.push_back(i - nr_ground);
      }
    }

    cout << monitored_ground.size() << " " << monitored_station.size() << endl;
    for (vector<int>::iterator it = monitored_ground.begin(); it != monitored_ground.end(); ++it) {
      cout << *it << " ";
    }
    for (vector<int>::iterator it = monitored_station.begin(); it != monitored_station.end(); ++it) {
      cout << *it << " ";
    }
    cout << endl;
  }

  return 0;
}
```

4.3 Coin Tossing

Keywords: Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Custom add edge function, Max-flow

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

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

using namespace std;
```

```
using namespace boost;

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;
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 add_edge, also creates reverse edges with corresponding capacities.
void addEdge(int u, int v, long c, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, Graph &G) {
  Edge e, reverseE;
  tie(e, tuples::ignore) = add_edge(u, v, G);
  tie(reverseE, tuples::ignore) = add_edge(v, u, G);
  capacity[e] = c;
  capacity[reverseE] = 0;
  rev_edge[e] = reverseE;
  rev_edge[reverseE] = e;
}


int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
      int players, rounds;
      cin >> players >> rounds;

      // create graph and additional stuff
      Graph graph(players + rounds + 2);
      EdgeCapacityMap capacity = get(edge_capacity, graph);
    ReverseEdgeMap  rev_edge = get(edge_reverse, graph);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);

    // source ans target vertex
    int source = players + rounds;
    int target = players + rounds + 1;

      for(int roundIndex = 0; roundIndex < rounds; roundIndex++) {
        int playerA, playerB, outcome;
        cin >> playerA >> playerB >> outcome;

        int roundVertex = players + roundIndex;
        // source -> round vertex
        addEdge(source, roundVertex, 1, capacity, rev_edge, graph);

        // round vertex -> player vertex representing possible wins
        if(outcome == 0) {
          // both might have won
          addEdge(roundVertex, playerA, 1, capacity, rev_edge, graph);
          addEdge(roundVertex, playerB, 1, capacity, rev_edge, graph);
        } else if(outcome == 1) {
          // player A won
          addEdge(roundVertex, playerA, 1, capacity, rev_edge, graph);
        } else if(outcome == 2) {
          // player B won
          addEdge(roundVertex, playerB, 1, capacity, rev_edge, graph);
        }
      }

      int totalPoints = 0;
      for(int playerIndex = 0; playerIndex < players; playerIndex++) {
        int points;
        cin >> points;

        totalPoints += points;

        // player vertex -> target
        addEdge(playerIndex, target, points, capacity, rev_edge, graph);
      }

      long flowValue = push_relabel_max_flow(graph, source, target);

      if(flowValue == totalPoints && totalPoints == rounds) {
        cout << "yes" << endl;
      } else {
        cout << "no" << endl;
      }
    }

  return 0;
}
```

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

```cpp
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_players, nr_rounds;
    cin >> nr_players >> nr_rounds;

    vector<int> winners;
    vector<pair<int, int> > undecided;
    for (int i = 0; i < nr_rounds; i++) {
      int player_a, player_b, c;
      cin >> player_a >> player_b >> c;
      if (c == 0) {
        undecided.push_back(make_pair(player_a, player_b));
      }
      else if (c == 1) {
        winners.push_back(player_a);
      }
      else if (c == 2) {
        winners.push_back(player_b);
      }
    }

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

    //Remove certain results
    for (vector<int>::iterator it = winners.begin(); it != winners.end(); ++it) {
      points[*it]--;
    }

    int total_points_needed = 0;
    bool possible = true;
    for (int i = 0; i < nr_players; i++) {
      if (points[i] < 0) {
        possible = false;
        break;
      }
      total_points_needed += points[i];
    }
    if (total_points_needed != undecided.size()) {
      possible = false;
    }

    if (possible) {
      int nr_vertices = nr_players + undecided.size() + 2;
      int source = nr_vertices - 2;
      int sink = nr_vertices - 1;
      Graph g(nr_vertices);
      EdgeCapacityMap capacity = get(edge_capacity, g);
      ReverseEdgeMap rev_edge = get(edge_reverse, g);

      for (int i = 0; i < undecided.size(); i++) {
        int edge_id = nr_players + i;
        Edge e, rev_e;
        bool success;

        //add middle edges
        tie(e, success) = add_edge(edge_id, undecided[i].first, g);
        tie(rev_e, success) = add_edge(undecided[i].first, edge_id, g);
```

```
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;

        tie(e, success) = add_edge(edge_id, undecided[i].second, g);
        tie(rev_e, success) = add_edge(undecided[i].second, edge_id, g);
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;

        //add source edge
        tie(e, success) = add_edge(source, edge_id, g);
        tie(rev_e, success) = add_edge(edge_id, source, g);
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
      }

      int total_need = 0;
      for (int i = 0; i < nr_players; i++) {
        if (points[i] <= 0) {
          continue;
        }
        Edge e, rev_e;
        bool success;

        tie(e, success) = add_edge(i, sink, g);
        tie(rev_e, success) = add_edge(sink, i, g);
        capacity[e] = points[i];
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
      }

      possible = (total_points_needed == push_relabel_max_flow(g, source, sink));
    }
    if (possible) {
      cout << "yes" << endl;
    }
    else {
      cout << "no" << endl;
    }
  }

  return 0;
}
```

4.4 Kingdom Defence

Keywords: Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow, Custom add edge function inlined by hand

```
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <algorithm>

#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor Edge;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int test_count;
  cin >> test_count;
  for(int test = 0; test < test_count; test++) {
    int location_count, path_count;
    cin >> location_count >> path_count;

    // create graph
    Graph graph(location_count + 2); // don't forget source and sink
    EdgeCapacityMap capacity = get(edge_capacity, graph);
    ReverseEdgeMap rev_edge = get(edge_reverse, graph);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);
```

```
    // define source and sink vertex
    int source = location_count;
    int sink = location_count + 1;

    // holds locations, first element of pair is the amount of soldier it has and
    // the seond the amount it needs
    vector<pair<int, int> > locations(location_count);
    for(int location_index = 0; location_index < location_count; location_index++) {
      int has_soldiers, needs_soliders;
      cin >> has_soldiers >> needs_soliders;

      locations.at(location_index) = make_pair(has_soldiers, needs_soliders);
    }

    // read in paths for soldiers
    for(int path_index = 0; path_index < path_count; path_index++) {
      int from, to, min_soldiers, max_soldiers;

      cin >> from >> to >> min_soldiers >> max_soldiers;

      // modify how many soldiers a location needs
      // it needs more soldiers as some of them must move on to the next city
      locations.at(from).second += min_soldiers;
      // needs less soldiers, as it will for sure get some from the current path!
      locations.at(to).second -= min_soldiers;

      // add edge
      Edge edge, r_edge;
      tie(edge, tuples::ignore) = add_edge(from, to, graph);
      tie(r_edge, tuples::ignore) = add_edge(to, from, graph);
      // we already basically moved the minimum amount of soldiers above by modifieng the locations
      // we therefore are interested in the rest that might move over the path and can compare the resulting
      // flow with the expected sum of soldiers
      capacity[edge] = max_soldiers - min_soldiers;
      capacity[r_edge] = 0;
      rev_edge[edge] = r_edge;
      rev_edge[r_edge] = edge;
    }

    // add edges from the source to the city, with its "have soldiers" weights
    for(int location_index = 0; location_index < location_count; location_index++) {
      Edge edge, r_edge;
      tie(edge, tuples::ignore) = add_edge(source, location_index, graph);
      tie(r_edge, tuples::ignore) = add_edge(location_index, source, graph);
      capacity[edge] = locations.at(location_index).first;
      capacity[r_edge] = 0;
      rev_edge[edge] = r_edge;
      rev_edge[r_edge] = edge;
    }

    // add edges from the city to the sink with the city's "needs soldiers" weights
    // also keep track how much we need in total
    int need_total = 0;
    for(int location_index = 0; location_index < location_count; location_index++) {
      int needs = locations.at(location_index).second;
      if(needs <= 0) {
        continue;
      }

      Edge edge, r_edge;
      tie(edge, tuples::ignore) = add_edge(location_index, sink, graph);
      tie(r_edge, tuples::ignore) = add_edge(sink, location_index, graph);
      capacity[edge] = needs;
      capacity[r_edge] = 0;
      rev_edge[edge] = r_edge;
      rev_edge[r_edge] = edge;

      need_total += needs;
    }

    // do max flow
    int max = push_relabel_max_flow(graph, source, sink);

    // check if it corresponds to the expected amount (at least)
    if(max >= need_total) {
      cout << "yes" << endl;
    } else {
      cout << "no" << endl;
    }

  }

  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>
```

```cpp
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_locations, nr_paths;
    cin >> nr_locations >> nr_paths;

    Graph g(nr_locations + 2);
    int source = nr_locations;
    int sink = nr_locations + 1;
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);

    //first -> nr_stationed, second -> nr_needed
    vector<pair<int, int> > locations(nr_locations);
    for (int i = 0; i < nr_locations; i++) {
      int nr_stationed, nr_needed;
      cin >> nr_stationed >> nr_needed;
      locations[i] = make_pair(nr_stationed, nr_needed);
    }

    for (int i = 0; i < nr_paths; i++) {
      int from, to, min, max; //from, to can be equal!
      cin >> from >> to >> min >> max;
      locations[from].second += min;
      locations[to].second -= min;

      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(from, to, g);
      tie(rev_e, success) = add_edge(to, from, g);
      capacity[e] = max - min;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    for (int i = 0; i < nr_locations; i++) {
      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(source, i, g);
      tie(rev_e, success) = add_edge(i, source, g);
      capacity[e] = locations[i].first;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    int total_need = 0;
    for (int i = 0; i < nr_locations; i++) {
      if (locations[i].second < 0) {
        locations[i].second = 0;
        continue;
      }
      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(i, sink, g);
      tie(rev_e, success) = add_edge(sink, i, g);
      capacity[e] = locations[i].second;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
      total_need += locations[i].second;
    }

    int max = push_relabel_max_flow(g, source, sink);
```

```
    if (max >= total_need) {
      cout << "yes" << endl;
    }
    else {
      cout << "no" << endl;
    }
  }

  return 0;
}
```

4.5 The Great Game

Keywords: Dynamic Programming

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

using namespace std;

int get_min_moves(int start, vector<vector<int> >&trans, vector<int>& min_fields, vector<int>& max_fields, int
     field_count);

int get_max_moves(int start, vector<vector<int> >&trans, vector<int>& min_fields, vector<int>& max_fields, int
     field_count) {
    // if we reached the end, nothing can be doen
    if(start == field_count - 1) {
        return 0;
    }

    // only calculate if value unknown
    if(max_fields.at(start) == -1) {
        // keeps track of the maximum amount of steps needed to reach the end from the given starting point
        int cur_max_found = -1;

        // iterate over all edges one can follow at the `start` position
        for(vector<int>::iterator iter = trans.at(start).begin();
            iter != trans.at(start).end();
            ++iter) {
            // now we search for the largest minimum amount of steps needed to reach the end
            int possible_max = get_min_moves(*iter, trans, min_fields, max_fields, field_count);
            if(possible_max > cur_max_found) {
                cur_max_found = possible_max;
            }
        }

        // update value
        max_fields.at(start) = cur_max_found + 1; // we still have to take the edge we followed
    }

    // return solution
    return max_fields.at(start);
}

int get_min_moves(int start, vector<vector<int> >&trans, vector<int>& min_fields, vector<int>& max_fields, int
     field_count) {
    // if we reached the end, no more moves needed
    if(start == field_count - 1) {
        return 0;
    }

    // only calculate if we don't know the solution yet
    if(min_fields.at(start) == -1) {
        // search for the minimum amount of moves to win from the current starting point
        int cur_min_found = numeric_limits<int>::max();

        // iterate over all possible next moves, i.e. the edges leaving the starting position
        for(vector<int>::iterator iter = trans.at(start).begin();
            iter != trans.at(start).end();
            ++iter) {
            // `iter` refers now to the edge we can follow, i.e. the next position we reach

            // now we have to assume that our opponent will be in our way and make our life complicated.
            // search for the maximum of moves from the next point we can reach to the finish line.
            int possible_min = get_max_moves(*iter, trans, min_fields, max_fields, field_count);
            if(possible_min < cur_min_found) {
                cur_min_found = possible_min;
            }
        }

        min_fields.at(start) = cur_min_found + 1; // we still have to use the edge, so add one
    }

    // return solution
    return min_fields.at(start);
}
```

```
int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        // read in basic information
        int position_count, transition_count;
        cin >> position_count >> transition_count;

        int start_red, start_black;
        cin >> start_red >> start_black;
        start_red -= 1; start_black -= 1; // let it start with 0, not 1

        // read in the possible transitions
        vector<vector<int> > trans(position_count, vector<int>());
        for(int transition_index = 0; transition_index < transition_count; transition_index++) {
            int from, to;
            cin >> from >> to;
            from -= 1; to -= 1; // let it start at 0

            // create transition, it's directed!
            trans.at(from).push_back(to);
        }

        vector<int> min_fields(position_count, -1);
        vector<int> max_fields(position_count, -1);

        int red_min = get_min_moves(start_red, trans, min_fields, max_fields, position_count);
        int black_min = get_min_moves(start_black, trans, min_fields, max_fields, position_count);

        // now we calculate the minimum amount of games each of the players does
        int min_sherlock = -1;
        int min_moriarty = -1;

        // check if the steps needed to win with the red meeple is even ...
        if(red_min % 2 == 0) {
            // ... ok it is even. Now we have to find out how many game steps were needed to
            // move the red meeple to the target position, as every second move sherlock moves the
            // black and not the red meeple
            min_sherlock = ((red_min - 2) / 2) * 4 + 4;
        } else {
            min_sherlock = ((red_min - 1) / 2) * 4 + 1;
        }

        if(black_min % 2 == 0) {
            min_moriarty = ((black_min - 2) / 2) * 4 + 3;
        } else {
            min_moriarty = ((black_min - 1) / 2) * 4 + 2;
        }

        if(min_sherlock < min_moriarty) {
            cout << 0 << endl;
        } else {
            cout << 1 << endl;
        }

    }

  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

using namespace std;

int last_position;
vector<vector<int> > field;
vector<int> min_field;
vector<int> max_field;

int getMax(int pos);

int getMin(int pos) {
  if (pos == last_position) {
    return 0;
  }
  if (min_field[pos] == -1) {
    int cur_min = numeric_limits<int>::max();
    for (vector<int>::iterator it = field[pos].begin(); it != field[pos].end(); ++it) {
      int new_min = getMax(*it);
      if (new_min < cur_min) {
        cur_min = new_min;
```

```
      }
    }
    min_field[pos] = cur_min + 1;
  }
  return min_field[pos];
}

int getMax(int pos) {
  if (pos == last_position) {
    return 0;
  }
  if (max_field[pos] == -1) {
    int cur_max = -1;
    for (vector<int>::iterator it = field[pos].begin(); it != field[pos].end(); ++it) {
      int new_max = getMin(*it);
      if (new_max > cur_max) {
        cur_max = new_max;
      }
    }
    max_field[pos] = cur_max + 1;
  }
  return max_field[pos];
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_positions, nr_transitions;
    cin >> nr_positions >> nr_transitions;
    last_position = nr_positions - 1;

    int r_start, b_start;
    cin >> r_start >> b_start;
    //adjust starting position
    r_start--;
    b_start--;

    field = vector<vector<int> >(nr_positions, vector<int>());
    for (int i = 0; i < nr_transitions; i++) {
      int from, to;
      cin >> from >> to;
      from--;
      to--;
      field[from].push_back(to);
    }

    min_field = vector<int>(nr_positions, -1);
    max_field = vector<int>(nr_positions, -1);

    int r_min = getMin(r_start);
    int b_min = getMin(b_start);


    if (r_min % 2 == 1) {
      r_min = ((r_min - 1) / 2) * 4 + 1;
    }
    else {
      r_min = ((r_min - 2) / 2) * 4 + 4;
    }

    if (b_min % 2 == 1) {
      b_min = ((b_min - 1) / 2) * 4 + 2;
    }
    else {
      b_min = ((b_min - 2) / 2) * 4 + 3;
    }
    if (r_min < b_min) {
      cout << 0 << endl;
    }
    else {
      cout << 1 << endl;
    }
  }

  return 0;
}
```

4.6 Surveillance Photographs

Keywords: Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Custom add edge function, Max-flow

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

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

```cpp
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

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;
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 add_edge, also creates reverse edges with corresponding capacities.
void addEdge(int u, int v, long c, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, Graph &G) {
  Edge e, reverseE;
  tie(e, tuples::ignore) = add_edge(u, v, G);
  tie(reverseE, tuples::ignore) = add_edge(v, u, G);
  capacity[e] = c;
  capacity[reverseE] = 0;
  rev_edge[e] = reverseE;
  rev_edge[reverseE] = e;
}


int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        int intersection_count, street_count, station_count, photograph_count;
        cin >> intersection_count >> street_count >> station_count >> photograph_count;

        // create graph and additional stuff
        // we need a graph that can hold a sink, a source ant twice the citie's map
        Graph graph(2 * intersection_count + 2);
        EdgeCapacityMap capacity = get(edge_capacity, graph);
        ReverseEdgeMap  rev_edge = get(edge_reverse, graph);
        // not used: ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);
        int source = 2 * intersection_count;
        int sink = 2 * intersection_count + 1;


        // read in where police stations are located and connect them to source/sink
        for(int station_index = 0; station_index < station_count; station_index++) {
            int station_loc;
            cin >> station_loc;

            // source to police station with weight 1 as each station has only one officer
            addEdge(source, station_loc, 1, capacity, rev_edge, graph);

            // police station to sink with weight 1 as each station can only hold one photograph,
            // ATTENTION here police station is in the second set!
            addEdge(intersection_count + station_loc, sink, 1, capacity, rev_edge, graph);
        }

        // read where photographs are stored and connect this location from our first set (where policemen reach the
             location)
        // to our second set (where policement can only use a street once to get beck to a station)
        for(int photo_index = 0; photo_index < photograph_count; photo_index++) {
            int photo_loc;
            cin >> photo_loc;

            addEdge(photo_loc, intersection_count + photo_loc, 1, capacity, rev_edge, graph);
        }

        // read where the streets are
        for(int street_index = 0; street_index < street_count; street_index++) {
            int from, to;
            cin >> from >> to;

            // first add street with unbound traffic to the first set, as all policemen are free to move multiple times
            // through the same street without a photograph
            addEdge(from, to, numeric_limits<int>::max(), capacity, rev_edge, graph);

            // now we add the same street, but it can be used only once as now it is used by policemen
            // with photographs
            addEdge(from + intersection_count, to + intersection_count, 1, capacity, rev_edge, graph);
        }

      long flowValue = push_relabel_max_flow(graph, source, sink);

      cout <<flowValue<< endl;
    }

  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_intersections, nr_streets, nr_police_stations, nr_photographs;
    cin >> nr_intersections >> nr_streets >> nr_police_stations >> nr_photographs;

    Graph g(nr_intersections * 2 + 2);
    int source = nr_intersections * 2;
    int sink = nr_intersections * 2 + 1;
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);

    for (int i = 0; i < nr_police_stations; i++) {
      int station_location;
      cin >> station_location;

      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(source, station_location, g);
      tie(rev_e, success) = add_edge(station_location, source, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;

      tie(e, success) = add_edge(station_location + nr_intersections, sink, g);
      tie(rev_e, success) = add_edge(sink, station_location + nr_intersections, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    for (int i = 0; i < nr_photographs; i++) {
      int photgraph_location;
      cin >> photgraph_location;

      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(photgraph_location, photgraph_location + nr_intersections, g);
      tie(rev_e, success) = add_edge(photgraph_location + nr_intersections, photgraph_location, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    vector<pair<int, int> > streets(nr_streets);
    for (int i = 0; i < nr_streets; i++) {
      int from, to;
      cin >> from >> to;

      Edge e, rev_e;
      bool success;

      tie(e, success) = add_edge(from, to, g);
```

```
      tie(rev_e, success) = add_edge(to, from, g);
      capacity[e] = nr_police_stations;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;

      tie(e, success) = add_edge(from + nr_intersections, to + nr_intersections, g);
      tie(rev_e, success) = add_edge(to + nr_intersections, from + nr_intersections, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    int max = push_relabel_max_flow(g, source, sink);
    cout << max << endl;
  }

  return 0;
}
```

5 CGAL Introduction

5.1 Hit?

Keywords: CGAL Segment, CGAL Ray, CGAL do_intersect

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <stdexcept>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;
typedef K::Ray_2 Ray;

using namespace std;

int main()
{
  while(true) {
    int obsticale_count;
    cin >> obsticale_count;

    // kill switch for application
    if(obsticale_count == 0) {
      break;
    }

    // get the laser
    double ray_start_x, ray_start_y, ray_other_x, ray_other_y;
    cin >> ray_start_x >> ray_start_y >> ray_other_x >> ray_other_y;
    Ray laser_ray = Ray(Point(ray_start_x, ray_start_y), Point(ray_other_x, ray_other_y));

    // read in obsticles and check if we hit them with the ray
    bool hit = false;
    for(int i = 0; i < obsticale_count; i++) {
      // read in and create obsticle
      double obsticle_start_x, obsticle_start_y, obsticle_end_x, obsticle_end_y;
      cin >> obsticle_start_x >> obsticle_start_y >> obsticle_end_x >> obsticle_end_y;

      // we know we hit something, read input but do nothing with it :)
      if(hit) {
        continue;
      }

      Segment obsticle = Segment(Point(obsticle_start_x, obsticle_start_y), Point(obsticle_end_x, obsticle_end_y));

      // check if we hit it
      if(CGAL::do_intersect(laser_ray, obsticle)) {
        hit = true;
      }
    }

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

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;
typedef K::Ray_2 Ray;

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int l;
  for (cin >> l; l != 0; cin >> l) {
    long long x, y, a, b;
    cin >> x >> y >> a >> b;
    Ray ray(Point(x, y), Point(a, b));

    bool found_intersect = false;
    for (int i = 0; i < l; i++) {
      long long r, s, t, u;
      cin >> r >> s >> t >> u;
      if (CGAL::do_intersect(ray, Segment(Point(r, s), Point(t, u)))) {
        for (i++; i < l; i++) {
```

```
        cin >> r >> s >> t >> u;
      }
      found_intersect = true;
    }
  }
  if (found_intersect) {
    cout << "yes" << endl;
  }
  else {
    cout << "no" << endl;
  }
}

return 0;
}
```

5.2 Antenna

Keywords: floor_to_double, ceil_to_double, Minimal Circle, CGAL sqrt

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

#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 <CGAL/number_utils.h>

// typedefs
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K; // IMPORTANT!!!!! OTHERWISE NO SQRT!!!!!
typedef CGAL::Min_circle_2_traits_2<K>  Traits;
typedef CGAL::Min_circle_2<Traits>      Min_circle;
typedef K::Point_2 Point;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
      int people_count;
      cin >> people_count;

      // kill switch for application
      if(people_count == 0) {
        break;
      }

      // collect all positions of people living at
      vector<Point> peoples(people_count);
      for(int i = 0; i < people_count; i++) {
        double person_x, person_y;
        cin >> person_x >> person_y;

        peoples.at(i) = Point(person_x, person_y);
      }

      // create the circle covering all people with minimal surface
      Min_circle min_circle(peoples.begin(), peoples.end(), true);

      // get radius
      cout << ceil_to_double(sqrt(min_circle.circle().squared_radius())) << endl;
  }
}
```

```
#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>
#include <limits>
#include <algorithm>
```

```
#include <queue>
#include <set>
#include <utility>
#include <cmath>

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;
typedef K::Point_2 Point;

using namespace std;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    vector<Point> points(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points[i] = Point(x, y);
    }
    Min_circle mc(points.begin(), points.end(), true);
    cout << ceil_to_double(sqrt(mc.circle().squared_radius())) << endl;
  }

  return 0;
}
```

5.3 First Hit

Keywords: CGAL randomise, CGAL intersection

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <stdexcept>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;
typedef K::Ray_2 Ray;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

inline void laser_obsticale_segment(Segment &lo_segment, CGAL::Object intersec_obj) {
  if (const Point* op = CGAL::object_cast<Point>(&intersec_obj)) {
    lo_segment = Segment(lo_segment.source(), *op);
  } else if(const Segment* os = CGAL::object_cast&lt;Segment&gt;(&intersec_obj)) {
    // ray hits a segment, three possibilities
    if(CGAL::collinear_are_ordered_along_line(lo_segment.source(), (*os).source(), (*os).target())) {
      // order is: laser source -> start of segment -> end of segment
      lo_segment = Segment(lo_segment.source(), (*os).source());
    } else {
      lo_segment = Segment(lo_segment.source(), (*os).target());
    }
  } else {
    throw runtime_error("Bad Wolf");
```

```
  }
}

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
  //cout << fixed << setprecision(0);

  while(true) {
    int obsticale_count;
    cin >> obsticale_count;

    // kill switch for application
    if(obsticale_count == 0) {
      break;
    }

    // get the laser
    double ray_start_x, ray_start_y, ray_other_x, ray_other_y;
    cin >> ray_start_x >> ray_start_y >> ray_other_x >> ray_other_y;
    Ray laser_ray = Ray(Point(ray_start_x, ray_start_y), Point(ray_other_x, ray_other_y));

    vector&lt;Segment&gt; obsticle_segments(obsticale_count);

    // read in and create obsticle list
    for(int i = 0; i < obsticale_count; i++) {
      double obsticle_start_x, obsticle_start_y, obsticle_end_x, obsticle_end_y;
      cin >> obsticle_start_x >> obsticle_start_y >> obsticle_end_x >> obsticle_end_y;

      obsticle_segments[i] = Segment(Point(obsticle_start_x, obsticle_start_y), Point(obsticle_end_x, obsticle_end_y));
    }

    random_shuffle(obsticle_segments.begin(), obsticle_segments.end());

    // segment that starts at the source of the laser and ends, after our algo is done, at the point where the laser hits
        an
    // obsticle.

    // search for one intersection between the laser ray and an obsticle, create a segment that starts at the laser
        source and ends
    // at the point where the laser hits the obsticle
    bool hit_found = false;
    Segment lo_segment(laser_ray.source(), laser_ray.source());
    int obst_index = 0; // used to jump over already checked segments in the second for-loop

    for(; obst_index < obsticale_count; ++obst_index) {
      if(do_intersect(obsticle_segments[obst_index], laser_ray)) {
        hit_found = true;
        laser_obsticale_segment(lo_segment, intersection(obsticle_segments[obst_index], laser_ray));
        break;
      }
    }

    // check if we hit something
    if(!hit_found) {
      cout << "no" << endl;
      continue;
    }

    for(; obst_index < obsticale_count; ++obst_index) {
      if(do_intersect(lo_segment, obsticle_segments[obst_index])) {
        laser_obsticale_segment(lo_segment, intersection(obsticle_segments[obst_index], laser_ray));
      }
    }

    cout << floor_to_double(lo_segment.target().x()) << " " << floor_to_double(lo_segment.target().y()) << endl;
  }
}
```

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;
typedef K::Ray_2 Ray;

using namespace std;

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;
}

double ceil_to_double(const K::FT& x) {
```

```
  double a = ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a - 1 >= x) a -= 1;
  return a;
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << setiosflags(ios::fixed) << setprecision(0);

  int l;
  for (cin >> l; l != 0; cin >> l) {
    double x, y, a, b;
    cin >> x >> y >> a >> b;
    Point origin(x, y);
    Ray ray(origin, Point(a, b));

    bool found_intersect = false;
    K::FT nearest_dist;
    Point nearest_point;
    for (int i = 0; i < l; i++) {
      double r, s, t, u;
      cin >> r >> s >> t >> u;
      Segment seg(Point(r, s), Point(t, u));
      if (CGAL::do_intersect(ray, seg)) {
        auto o = CGAL::intersection(ray, seg);
        K::FT new_dist;
        Point new_point;
        if (const Point* op = boost::get<Point>(&*o)) {
          new_dist = Segment(origin, *op).squared_length();
          new_point = *op;
        }
        else if (const Segment* os = boost::get&lt;Segment&gt;(&*o)) {
          if (CGAL::collinear_are_ordered_along_line(origin, os->source(), os->target())) {
            new_dist = Segment(origin, os->source()).squared_length();
            new_point = os->source();
          }
          else if (CGAL::collinear_are_ordered_along_line(origin, os->target(), os->source())) {
            new_dist = Segment(origin, os->target()).squared_length();
            new_point = os->target();
          }
          else {
            //segment going through ray origin point -> distance is 0
            new_dist = 0.0;
            new_point = origin;
          }
        }
        else {
          throw runtime_error("strange segment intersection");
        }
        if (!found_intersect) {
          nearest_dist = new_dist;
          nearest_point = new_point;
          found_intersect = true;
        }
        else {
          if (new_dist < nearest_dist) {
            nearest_dist = new_dist;
            nearest_point = new_point;
          }
        }
      }
    }

    if (found_intersect) {
      cout << floor_to_double(nearest_point.x()) << " " << floor_to_double(nearest_point.y()) << endl;
    }
    else {
      cout << "no" << endl;
    }
  }

  return 0;
}
```

5.4 Almost Antenna

Keywords: CGAL qrt, Minimal Circle, floor_to_double, ceil_to_double

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

#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 <CGAL/number_utils.h>

// typedefs
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K; // IMPORTANT!!!!! OTHERWISE NO SQRT!!!!!
typedef CGAL::Min_circle_2_traits_2<K>  Traits;
```

```cpp
typedef CGAL::Min_circle_2<Traits>      Min_circle;
typedef K::Point_2 Point;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
      int people_count;
      cin >> people_count;

      // kill switch for application
      if(people_count == 0) {
        break;
      }

      // collect all positions of people living at
      vector<Point> peoples(people_count);
      for(int i = 0; i < people_count; i++) {
        double person_x, person_y;
        cin >> person_x >> person_y;

        peoples.at(i) = Point(person_x, person_y);
      }

      // create the circle covering all people with minimal surface
      Min_circle min_circle(peoples.begin(), peoples.end(), true);

      // now we remove the support points, calculate the radius for the resulting min circles and choose the smallest one
           as the solution
      K::FT min_rad = sqrt(min_circle.circle().squared_radius());
      for(auto iter = min_circle.support_points_begin(); iter != min_circle.support_points_end(); iter++) {
        for(int i = 0; i < people_count; i++) {
          if(peoples.at(i) == *iter) {
            int add = i > 0 ? -1 : 1;
            peoples.at(i) = peoples.at(i + add);

            Min_circle almost_circ(peoples.begin(), peoples.end(), true);
            K::FT almost_rad = sqrt(almost_circ.circle().squared_radius());
            if(almost_rad < min_rad) {
              min_rad = almost_rad;
            }

            peoples.at(i) = *iter;

            break;
          }
        }
      }

      // get radius
      cout << ceil_to_double(min_rad) << endl;
  }
}
```

```cpp
#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>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

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;
typedef K::Point_2 Point;

using namespace std;
```

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    vector<Point> points(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points[i] = Point(x, y);
    }
    Min_circle mc(points.begin(), points.end(), true);
    double min_radius = ceil_to_double(sqrt(mc.circle().squared_radius()));
    for (auto sp_it = mc.support_points_begin(); sp_it != mc.support_points_end(); ++sp_it) {
      Point support_point = *sp_it;
      for (int i = 0; i < n; i++) {
        if (points[i] == support_point) {
          /*Min_circle almost_mc(points.begin(), points.begin()+i, true);
          if (i+1 < n) {
          almost_mc.insert(points.begin()+i+1, points.end());
          }
          double new_radius = ceil_to_double(sqrt(almost_mc.circle().squared_radius()));
          if (new_radius < min_radius) {
          min_radius = new_radius;
          }*/
          if (i == 0) {
            if (n > 1) {
              points[i] = points[i + 1];
            }
            else {
              points[i] = Point(0, 0);
            }
          }
          else {
            points[i] = points[i - 1];
          }
          Min_circle almost_mc(points.begin(), points.end(), true);
          points[i] = support_point;
          double new_radius = ceil_to_double(sqrt(almost_mc.circle().squared_radius()));
          if (new_radius < min_radius) {
            min_radius = new_radius;
          }
          break;
        }
      }
    }
    cout << min_radius << endl;
  }

  return 0;
}
```

6 Proximity Structures

6.1 Graypes

Keywords: setprecision, Delaunay Triangulation, Finite edge iteration

```
#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::Edge_iterator  Edge_iterator;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
      int graype_count;
      cin >> graype_count;

      // kill switch for application
      if(graype_count == 0) {
        break;
      }

      // collect graype locations
      vector<K::Point_2> graype_locs;
      graype_locs.reserve(graype_count);

      for(int i = 0; i < graype_count; i++) {
        double graype_x, graype_y;
        cin >> graype_x >> graype_y;

        graype_locs.push_back(K::Point_2(graype_x, graype_y));
      }

      // construct triangulation
      Triangulation triang;
      triang.insert(graype_locs.begin(), graype_locs.end());

      // go trough apes, and search for shortest edge
      K::FT min_time;
      bool first = true;
      for (Edge_iterator edge = triang.finite_edges_begin(); edge != triang.finite_edges_end(); ++edge) {
        K::FT edge_time = triang.segment(edge).squared_length();
        if(edge_time < min_time || first) {
            first = false;
            min_time = edge_time;
        }
      }

      // calculate time to run
      cout << ceil(sqrt(CGAL::to_double(min_time)) / 2.0 * 100.0) << endl;
    }
}
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator Edge_iterator;
typedef Triangulation::Point Point;
```

```
typedef Triangulation::Segment Segment;

using namespace std;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    vector<Point> points;
    points.reserve(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points.push_back(Point(x, y));
    }

    Triangulation t;
    t.insert(points.begin(), points.end());
    Segment shortest = t.segment(*(t.finite_edges_begin()));
    for (Edge_iterator e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e) {
      Segment s = t.segment(*e);
      if (s.squared_length() < shortest.squared_length()) {
        shortest = s;
      }
    }

    cout << ceil(sqrt(CGAL::to_double(shortest.squared_length())) / 2.0*100.0) << endl;

  }

  return 0;
}
```

6.2 Bistro

Keywords: Delaunay Triangulation

```
#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::Edge_iterator  Edge_iterator;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
      int existing_count;
      cin >> existing_count;

      // kill switch for application
```

```
      if(existing_count == 0) {
        break;
      }

      // collect existing restaurants
      vector<K::Point_2> existing_locs;
      existing_locs.reserve(existing_count);

      for(int i=0; i < existing_count; i++) {
        double loc_x, loc_y;
        cin >> loc_x >> loc_y;

        existing_locs.push_back(K::Point_2(loc_x, loc_y));
      }

      // cosntruct triangulation
      Triangulation triang;
      triang.insert(existing_locs.begin(), existing_locs.end());

      // go through possible location
      int possible_count;
      cin >> possible_count;

      for(int i = 0; i < possible_count; i++) {
        int possible_x, possible_y;
        cin >> possible_x >> possible_y;

        K::Point_2 possible_point = K::Point_2(possible_x, possible_y);

        // find nearest vertex and by that the nearest point
        K::Point_2 nearest = triang.nearest_vertex(possible_point)->point();
        cout << CGAL::to_double(CGAL::squared_distance(nearest, possible_point)) << endl;
      }
    }
}
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator Edge_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;

using namespace std;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    vector<Point> points;
    points.reserve(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points.push_back(Point(x, y));
    }

    Triangulation t;
    t.insert(points.begin(), points.end());
    int m;
    cin >> m;

    for (int i = 0; i < m; i++) {
      double x, y;
```

```
      cin >> x >> y;
      Point loc(x, y);
      K::FT sqared_dist = CGAL::squared_distance(t.nearest_vertex(loc)->point(), loc);
      cout << CGAL::to_double(sqared_dist) << endl;
    }
  }

  return 0;
}
```

6.3 H1N1

Keywords: Delaunay Triangulation, CGAL Triangulation with DFS, Finite face iteration

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
    double a = std::floor(CGAL::to_double(x));
    while (a > x) a -= 1;
    while (a+1 <= x) a += 1;
    return a;
}

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

bool DFS(Triangulation &triang, double min_dist_for_edge, Triangulation::Face_handle &start, map<Triangulation::
     Face_handle, int> &visitor_map, int cur_iter) {
    // each infected person's point is connected to three edges, otherwise we wouldn't get triangle
    for(int edge_i = 0; edge_i < 3; edge_i++) {
        // get the segment representing the edge
        Triangulation::Segment edge_segment = triang.segment(start, edge_i);

        // check if we are still far away from both endpoints of the segment while passing through
        double distance = CGAL::to_double(edge_segment.squared_length());
        if(distance >= min_dist_for_edge) {
            // get the neighboring face
            Triangulation::Face_handle neighbor_face_h = start->neighbor(edge_i);

            // check that we didn't visit it already
            if(visitor_map[neighbor_face_h] == cur_iter) {
                continue; // use other edge
            }

            // check if neighboring face is infinite, i.e. we found a way out
            if(triang.is_infinite(neighbor_face_h)) {
                return true;
            }

            // mark as visited by current iteration
            visitor_map[neighbor_face_h] = cur_iter;

            // recursion, DFS
            if(DFS(triang, min_dist_for_edge, neighbor_face_h, visitor_map, cur_iter)) {
                return true;
            }
        }
    }

    return false;
}

int main() {
    // some basic setup stuff
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    cout << fixed << setprecision(0);

    while(true) {
        int infected_count;
        cin >> infected_count;

        // kill switch for application
        if(infected_count == 0) {
            break;
        }

        // read in infected people's location
```

```
        vector<K::Point_2> infected_locs;
        infected_locs.reserve(infected_count);
        for(int i = 0; i < infected_count; i++) {
            double infected_x, infected_y;
            cin >> infected_x >> infected_y;

            infected_locs.push_back(K::Point_2(infected_x, infected_y));
        }

        // create triangulation with the infected people
        Triangulation triang;
        triang.insert(infected_locs.begin(), infected_locs.end());

        // prepare a map mapping a face handle to an integer, the integer represents the DFS step in which the face was
        // visited already (to not visit multiple time the same face)
        map<Triangulation::Face_handle, int> visitor_map;
        for (Triangulation::Face_iterator it = triang.finite_faces_begin(); it != triang.finite_faces_end(); it++) {
            visitor_map[it] = -1;
        }

        // go trough people trying to escape
        int escapee_count;
        cin >> escapee_count;
        for(int i = 0; i < escapee_count; i++) {
            // read location of escapee and the minimum distance expected
            double escapee_x, escapee_y, min_dist;
            cin >> escapee_x >> escapee_y >> min_dist;

            K::Point_2 escapee_loc(escapee_x, escapee_y);

            // get nearst vertex's point in the triangulation
            Triangulation::Point nearest_vertex_point = triang.nearest_vertex(escapee_loc)->point();

            // check that we not already violate the distance condition
            if(CGAL::to_double(CGAL::squared_distance(escapee_loc, nearest_vertex_point)) < min_dist) {
                cout << "n";
                continue; // jump over the rest, we already know the escapee is near an infected person
            }

            // find the face handle of our escapee
            Triangulation::Face_handle face_h = triang.locate(escapee_loc);

            // update for the face handle that we visited it in the current round
            visitor_map[face_h] = i;
            if(triang.is_infinite(face_h) || // face is already outside, escapee can escape, no DFS needed
                DFS(triang, min_dist * 4.0, face_h, visitor_map, i)) { // use DFS to find a way out, multiply by 4.0 (=
                    2.0^2 (not squared distances)) as we must be min_dist away from one ende of an edge and min_dist
                    from the other, multiply by 2^2
                cout << "y";
            } else {
                cout << "n";
            }
        }

        cout << endl;
    }
}
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<int, 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::Face_handle Face_handle;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_faces_iterator All_faces_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;

using namespace std;

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

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

bool find_exit(Triangulation& t, Face_handle& face, K::FT& min_segment_squared_length, int current_id) {
  for (int i = 0; i < 3; i++) {
    Segment s = t.segment(face, i);
    if (s.squared_length() >= min_segment_squared_length) {
      Face_handle neighbor_face = face->neighbor(i);
      if (neighbor_face->info() == current_id) {
        continue;
      }
      if (t.is_infinite(neighbor_face)) {
        return true;
      }
      neighbor_face->info() = current_id;
      if (find_exit(t, neighbor_face, min_segment_squared_length, current_id)) {
        return true;
      }
    }
  }
  return false;
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    vector<Point> points;
    points.reserve(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points.push_back(Point(x, y));
    }

    Triangulation t;
    t.insert(points.begin(), points.end());
    for (All_faces_iterator f = t.all_faces_begin(); f != t.all_faces_end(); ++f) {
      f->info() = 0;
    }

    int m;
    cin >> m;

    for (int i = 0; i < m; i++) {
      double x, y;
      K::FT distance, min_segment_squared_length;
      cin >> x >> y >> distance;
      Point loc(x, y);
      K::FT squared_dist = CGAL::squared_distance(t.nearest_vertex(loc)->point(), loc);
      Face_handle face = t.locate(loc);
      min_segment_squared_length = distance*4.0;

      face->info() = i + 1;
      if (squared_dist >= distance && (t.is_infinite(face) || find_exit(t, face, min_segment_squared_length, i + 1))) {
        cout << 'y';
      }
      else {
        cout << 'n';
      }
    }
    cout << endl;
  }

  return 0;
}
```

6.4 Germs

Keywords: Delaunay Triangulation, Finite vertices iteration, Finite edge iteration, Iteration over std::map

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
    double a = std::floor(CGAL::to_double(x));
```

```
    while (a > x) a -= 1;
    while (a+1 <= x) a += 1;
    return a;
}

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

double hours(double distance) {
    double x = sqrt(distance) - 0.5; // first sqrt to get the distance ...
    double result;
    if (x <= 0.0) {
        result = 0;
    } else {
        result = ceil(sqrt(x)); // second sqrt because t^2 + 0.5 = x => t = sqrt(x - 0.5);
    }
    return result;
}

int main() {

    while(true) {
        int bacteria_count;
        cin >> bacteria_count;

        // kill switch for application
        if(bacteria_count == 0) {
            break;
        }

        // read in boundaries of the dish
        double left_border, right_border, bottom_border, top_border;
        cin >> left_border >> bottom_border >> right_border >> top_border;

        // collect bacteria's center information
        vector<K::Point_2> bacteria_centers;
        bacteria_centers.reserve(bacteria_count);
        for(int i = 0; i < bacteria_count; i++) {
            double bacteria_x, bacteria_y;
            cin >> bacteria_x >> bacteria_y;

            bacteria_centers.push_back(K::Point_2(bacteria_x, bacteria_y));
        }

        // create triangulation
        Triangulation triang;
        triang.insert(bacteria_centers.begin(), bacteria_centers.end());

        // keep track of the distances for each bacteria
        map<Triangulation::Point, double> distances;
        //distances.reserve(bacteria_count);

        // calculate initial distance: distance between the bacteria and the nearest dish boundary
        for(Triangulation::Finite_vertices_iterator vertex_iter = triang.finite_vertices_begin(); vertex_iter != triang.
             finite_vertices_end(); ++vertex_iter) {
            Triangulation::Point vertex = vertex_iter->point();
            distances[vertex] = min(
                min(vertex.x() - left_border, right_border - vertex.x()), // left/right minimum
                min(vertex.y() - bottom_border, top_border - vertex.y()) // top/bottom minimum
            );

            distances[vertex] *= distances[vertex]; // square distance as we work with squared ones
        }

        // compute distance to other two neighbours and update distance if it is smaller
        for(Triangulation::Finite_edges_iterator edge_iter = triang.finite_edges_begin(); edge_iter != triang.
             finite_edges_end(); ++edge_iter) {
            Triangulation::Vertex_handle vertex1 = edge_iter->first->vertex(triang.cw(edge_iter->second));
            Triangulation::Vertex_handle vertex2 = edge_iter->first->vertex(triang.ccw(edge_iter->second));

            Triangulation::Point vertex1_point = vertex1->point();
            Triangulation::Point vertex2_point = vertex2->point();

            // calculate distance of the points of both vertex and half them (divide by 4 as distance is squared and 4 =
                2^2)
            double vertex_distance = CGAL::to_double(CGAL::squared_distance(vertex1_point, vertex2_point)) / 4;

            // update distances to minimum
            distances[vertex1_point] = min(distances[vertex1_point], vertex_distance);
            distances[vertex2_point] = min(distances[vertex2_point], vertex_distance);
        }

        // now we know the minimum distance for each bacteria to another one or the borders of the dish

        // extract distances into a vector and sort it
        vector<double> only_distances;
        only_distances.reserve(bacteria_count);
        for(map<Triangulation::Point, double>::iterator iter = distances.begin(); iter != distances.end(); ++iter) {
            only_distances.push_back(iter->second);
        }
```

```
        // sort distances
        sort(only_distances.begin(), only_distances.end());

        // print out information
        cout << hours(only_distances[0]) << " " << hours(only_distances[bacteria_count/2]) << " " << hours(only_distances
            [bacteria_count - 1]) << endl;
    }
}
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_with_info_2<K::FT, 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::Face_handle Face_handle;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_faces_iterator All_faces_iterator;
typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator Finite_edges_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;

using namespace std;

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

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

double distance_to_time(K::FT distance) {
  double tmp = sqrt(CGAL::to_double(distance)) - 0.5;
  return (tmp > 0.0) ? ceil(sqrt(tmp)) : 0.0;
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int n;
  for (cin >> n; n != 0; cin >> n) {
    double l, b, r, t;
    cin >> l >> b >> r >> t;

    vector<Point> points;
    points.reserve(n);
    for (int i = 0; i < n; i++) {
      double x, y;
      cin >> x >> y;
      points.push_back(Point(x, y));
    }
    Triangulation bac;
    bac.insert(points.begin(), points.end());

    for (Finite_vertices_iterator it = bac.finite_vertices_begin(); it != bac.finite_vertices_end(); ++it) {
      Point& p = it->point();
      it->info() = min(min(p.x() - l, r - p.x()), min(p.y() - b, t - p.y()));
      it->info() *= it->info();
    }

    for (Finite_edges_iterator it = bac.finite_edges_begin(); it != bac.finite_edges_end(); ++it) {
      Triangulation::Vertex_handle v1 = it->first->vertex((it->second + 1) % 3);
      Triangulation::Vertex_handle v2 = it->first->vertex((it->second + 2) % 3);
      K::FT d = CGAL::squared_distance(v1->point(), v2->point()) / 4;
      v1->info() = min(v1->info(), d);
      v2->info() = min(v2->info(), d);
    }
```

```
    vector<K::FT> bac_expand_distances;
    bac_expand_distances.reserve(n);
    for (Finite_vertices_iterator it = bac.finite_vertices_begin(); it != bac.finite_vertices_end(); ++it) {
      bac_expand_distances.push_back(it->info());
    }
    sort(bac_expand_distances.begin(), bac_expand_distances.end());
    cout << distance_to_time(bac_expand_distances[0]) << " " << distance_to_time(bac_expand_distances[n / 2]) << " " <<
        distance_to_time(bac_expand_distances[n - 1]) << endl;
  }

  return 0;
}
```

6.5 Hiking Maps

Keywords: Delaunay Triangulation, CGAL turn function, CGAL right_turn, Scanline

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<int, 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::Face_handle Face_handle;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_faces_iterator All_faces_iterator;
typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator Finite_edges_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;
typedef Triangulation::Line_face_circulator Line_face_circulator;
typedef K::Line_2 Line;

using namespace std;

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

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

inline bool containsSegment(const vector<Point>& triangle, const Point& s1, const Point& s2) {
  return !CGAL::right_turn(triangle[1], triangle[0], s1) &&
    !CGAL::right_turn(triangle[3], triangle[2], s1) &&
    !CGAL::right_turn(triangle[5], triangle[4], s1) &&
    !CGAL::right_turn(triangle[1], triangle[0], s2) &&
    !CGAL::right_turn(triangle[3], triangle[2], s2) &&
    !CGAL::right_turn(triangle[5], triangle[4], s2);
}

void print_array(vector<int>& a) {
  for (int i = 0; i < a.size(); i++) {
    cout << a[i] << " ";
  }
  cout << endl;
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int nr_testcases;
  cin >> nr_testcases;
  for (int testcase = 0; testcase < nr_testcases; ++testcase) {
    int m, n;
```

```
    cin >> m >> n;

    vector<Point> legs;
    legs.reserve(m);
    for (int i = 0; i < m; i++) {
      int x, y;
      cin >> x >> y;
      legs.push_back(Point(x, y));
    }

    vector<vector<Point> > triangles;
    triangles.reserve(n);
    for (int i = 0; i < n; i++) {
      vector<Point> triangle;
      triangle.reserve(6);

      int x0, y0, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5;
      cin >> x0 >> y0 >> x1 >> y1 >> x2 >> y2 >> x3 >> y3 >> x4 >> y4 >> x5 >> y5;
      Point p0(x0, y0), p1(x1, y1), p2(x2, y2), p3(x3, y3), p4(x4, y4), p5(x5, y5);

      if (CGAL::left_turn(p2, p1, p0)) {
        triangle.push_back(p0);
        triangle.push_back(p1);
      }
      else {
        triangle.push_back(p1);
        triangle.push_back(p0);
      }
      if (CGAL::left_turn(p4, p3, p2)) {
        triangle.push_back(p2);
        triangle.push_back(p3);
      }
      else {
        triangle.push_back(p3);
        triangle.push_back(p2);
      }
      if (CGAL::left_turn(p0, p5, p4)) {
        triangle.push_back(p4);
        triangle.push_back(p5);
      }
      else {
        triangle.push_back(p5);
        triangle.push_back(p4);
      }
      triangles.push_back(triangle);
    }

    vector<int> leg_to_triangle(m - 1, -1);
    vector<int> triangle_contributions(n, 0);
    int found_legs = 0;
    int bound_max = 0;
    int i = 0;
    while (found_legs < m - 1 && i < n) {
      for (int j = 0; j < m - 1; j++) {
        if (containsSegment(triangles[i], legs[j], legs[j + 1])) {
          if (leg_to_triangle[j] == -1) {
            found_legs++;
          }
          else {
            triangle_contributions[leg_to_triangle[j]]--;
            //cout << "remove leg " << j << " covered by " << leg_to_triangle[j] << endl;
          }
          leg_to_triangle[j] = i;
          //cout << "leg " << j << " covered by " << i << endl;
          triangle_contributions[i]++;
        }
      }
      bound_max = i;
      i++;
    }
    int bound_min = 0;
    while (triangle_contributions[bound_min] == 0) bound_min++;
    int min_range = bound_max - bound_min;

    //cout << "min: " << bound_min << " max: " << bound_max << endl;
    while (i < n) {
      for (int j = 0; j < m - 1; j++) {
        if (containsSegment(triangles[i], legs[j], legs[j + 1])) {
          triangle_contributions[leg_to_triangle[j]]--;
          leg_to_triangle[j] = i;
          triangle_contributions[i]++;
        }
      }
      bound_max = i;
      while (triangle_contributions[bound_min] == 0) bound_min++;
      min_range = min(min_range, bound_max - bound_min);
      //cout << "min: " << bound_min << " max: " << bound_max << endl;
      i++;
    }

    cout << min_range + 1 << endl;
  }

  return 0;
}
```

7 Linear/Quadratic Programming

7.1 What is the Maximum?

Keywords: Quadratic Program, ceil_to_double, floor_to_double

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

const int var_x = 0; const int X = var_x;
const int var_y = 1; const int Y = var_y;
const int var_z = 2; const int Z = var_z;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}


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

int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
    int p, a, b;
    cin >> p;

    // kill switch for application
    if(p == 0) {
      break;
    }

    cin >> a >> b;

    if(p == 1) {
      //                          v- set finite lower bound (x, y >= 0), defaults to 0
      //                                    v- no upper bound
      Program lp(CGAL::SMALLER, true, 0, false, 0);

      // x + y <= 4
      lp.set_a(var_x, 0, 1);
      lp.set_a(var_y, 0, 1);
      lp.set_b(0, 4);

      // 4x + 2y <= ab
      lp.set_a(var_x, 1, 4);
      lp.set_a(var_y, 1, 2);
      lp.set_b(1, a * b);

      // -x + y <= 1
      lp.set_a(var_x, 2, -1);
      lp.set_a(var_y, 2, 1);
      lp.set_b(2, 1);

      // maximize: by -ax^2 so we have to minimize -by + ax^2
      lp.set_c(var_y, -b);
      lp.set_d(var_x, var_x, 2*a);

      // solve it
      Solution s = CGAL::solve_nonnegative_quadratic_program(lp, ET());
      assert(s.solves_quadratic_program(lp));

      if(s.is_unbounded()) {
        cout << "unbounded" << endl;
      } else if(s.is_infeasible()) {
```

```
        cout << "no" << endl;
      } else {
        cout << floor_to_double(-s.objective_value()) << endl;
      }
    } else {
      // here we have an upper bound for the variables, but no lower bound
      Program lp(CGAL::LARGER, false, 0, false, 0);

      // bounds by benji
      // z >= 0
      lp.set_u(X, true, 0);
      lp.set_u(Y, true, 0);
      lp.set_l(Z, true, 0);

      // x + y >= -4
      lp.set_a(var_x, 0, 1);
      lp.set_a(var_y, 0, 1);
      lp.set_b(0, -4);

      // 4x + 2y + z^2 => -ab => we substitute z^2 by just z (don't forget to do the same for the minimized formula
          dingsi)
      // i.e. we get 4x + 2y + z >= -ab
      lp.set_a(var_x, 1, 4);
      lp.set_a(var_y, 1, 2);
      lp.set_a(var_z, 1, 1);
      lp.set_b(1, -(a*b));

      // -x + y >= -11
      lp.set_a(var_x, 2, -1);
      lp.set_a(var_y, 2, 1);
      lp.set_b(2, -1);

      // minimize ax^2 + by + z^4 => after our substitution: ax^2 + by + z^2
      lp.set_d(var_x, var_x, 2*a);
      lp.set_d(var_z, var_z, 2 * 1);
      lp.set_c(var_y, b);

      // solve it
      Solution s = CGAL::solve_quadratic_program(lp, ET());
      assert(s.solves_quadratic_program(lp));

      if(s.is_unbounded()) {
        cout << "unbounded" << endl;
      } else if(s.is_infeasible()) {
        cout << "no" << endl;
      } else {
        cout << ceil_to_double(s.objective_value()) << endl;
      }
    }
  }
}
```

7.2 Diets

Keywords: Quadratic Program, Non-negative quadratic program

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

const int var_x = 0; const int X = var_x;
const int var_y = 1; const int Y = var_y;
const int var_z = 2; const int Z = var_z;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}


int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a - 1 >= x) a -= 1;
```

```
  return a;
}

int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
    int nutrient_count, food_count;
    cin >> nutrient_count >> food_count;

    // kill switch for application
    if(nutrient_count == 0 && food_count == 0) {
      break;
    }

    // create linear programming application
    Program lp(CGAL::SMALLER, true, 0, false, 0);

    // read in nutrients
    vector<int> nutrient_min;
    vector<int> nutrient_max;
    nutrient_min.reserve(nutrient_count);
    nutrient_max.reserve(nutrient_count);
    for(int i = 0; i < nutrient_count; i++) {
      int min_amount, max_amount;
      cin >> nutrient_min[i] >> nutrient_max[i];
    }

    // read in foods and amount of nutrients
    vector<int> prices;
    vector<vector<int> > nutrient_amount(food_count, vector<int>(nutrient_count));
    prices.reserve(food_count);
    for(int i = 0; i < food_count; i++) {
      cin >> prices[i];

      for(int j = 0; j < nutrient_count; j++) {
        cin >> nutrient_amount[i][j];
      }
    }

    // vars: first food, then nutrients

    // set objective
    for(int i = 0; i < food_count; i++) {
      // we want to pay the minimum
      // results in the sum of food_var * food_price
      lp.set_c(i, prices[i]);
    }

    // set inequalities
    int eq_counter = 0;
    for(int nutrient_index = 0; nutrient_index < nutrient_count; nutrient_index++) {
      for(int food_index = 0; food_index < food_count; food_index++) {
        lp.set_a(food_index, eq_counter, nutrient_amount[food_index][nutrient_index]); // A := sum over: food * amount of
             nutrition
      }

      lp.set_b(eq_counter, nutrient_max[nutrient_index]); // A <= maximum needed
      eq_counter++;
    }

    for(int nutrient_index = 0; nutrient_index < nutrient_count; nutrient_index++) {
      for(int food_index = 0; food_index < food_count; food_index++) {
        lp.set_a(food_index, eq_counter, nutrient_amount[food_index][nutrient_index]); // B := sum over: food * amount of
             nutrition
      }

      // B >= minimum needed
      lp.set_b(eq_counter, nutrient_min[nutrient_index]);
      lp.set_r(eq_counter, CGAL::LARGER);
      eq_counter++;
    }

    // find solution and print out
    Solution s = CGAL::solve_nonnegative_quadratic_program(lp, ET());

    if(s.is_optimal()) {
      cout << floor_to_double(s.objective_value()) << endl;
    } else {
      cout << "No such diet." << endl;
    }
  }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
```

```
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

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

using namespace std;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

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

int main() {

  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int n, m;
  for (cin >> n >> m; n != 0; cin >> n >> m) {
    Program lp(CGAL::SMALLER, true, 0, false, 0);


    for (int i = 0; i < n; i++) {
      int min_nut, max_nut;
      cin >> min_nut >> max_nut;
      lp.set_r(i, CGAL::LARGER);
      lp.set_b(i, min_nut);
      lp.set_r(i + n, CGAL::SMALLER);
      lp.set_b(i + n, max_nut);
    }

    for (int i = 0; i < m; i++) {
      int price;
      cin >> price;
      lp.set_c(i, price); //add price to function to minimize
      for (int j = 0; j < n; j++) {
        int nut_amount;
        cin >> nut_amount;
        lp.set_a(i, j, nut_amount);
        lp.set_a(i, j + n, nut_amount);
      }
    }

    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 {
      cout << floor_to_double(s.objective_value()) << endl;
    }
  }

  return 0;
}
```

7.3 Portfolios

Keywords: Quadratic Program

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

// choose exact integral type
```

```cpp
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}


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

int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
    int asset_count, people_count;
    cin >> asset_count >> people_count;

    // kill switch for application
    if(asset_count == 0 && people_count == 0) {
      break;
    }

    // containers for asset properties
    vector<int> asset_cost(asset_count);
    vector<int> asset_expected_return(asset_count);

    // read in asset properties
    for(int asset_index = 0; asset_index < asset_count; asset_index++) {
      cin >> asset_cost.at(asset_index) >> asset_expected_return.at(asset_index);
    }

    // read and store covariance
    vector<vector<int> > covariance(asset_count, vector<int>(asset_count));
    for(int assetA = 0; assetA < asset_count; assetA++) {
      for(int assetB = 0; assetB < asset_count; assetB++) {
        cin >> covariance.at(assetA).at(assetB);
      }
    }

    // go trough each investor and calculate for him/her the result
    for(int person_index = 0; person_index < people_count; person_index++) {
      int max_cost, min_return, max_variance;
      cin >> max_cost >> min_return >> max_variance;

      // by default, we have a nonnegative QP with Ax >= b
      Program qp (CGAL::LARGER, true, 0, false, 0);

      // equation counter
      int eq_counter = 0;

      // add inequastion for expected return
      for(int asset_index = 0; asset_index < asset_count; asset_index++) {
        // sum of each asset amount times its expected return ...
        qp.set_a(asset_index, eq_counter, asset_expected_return[asset_index]);
      }
      // ... >= investor's expected minimal return
      qp.set_b(eq_counter, min_return);
      eq_counter++;

      // add inequastion for max cost for the investor
      for(int asset_index = 0; asset_index < asset_count; asset_index++) {
        // sum of each asset's cost times how many of them we buy ...
        qp.set_a(asset_index, eq_counter, asset_cost[asset_index]);
      }
      // ... <= investor's maximum cost
      qp.set_b(eq_counter, max_cost);
      qp.set_r(eq_counter, CGAL::SMALLER);
      eq_counter++;

      // objective function
      for(int assetA = 0; assetA < asset_count; assetA++) {
        for(int assetB = 0; assetB < asset_count; assetB++) {
          qp.set_d(assetA, assetB, 2 * covariance[assetA][assetB]);
```

```
      }
    }

    // calculate
    Solution s = CGAL::solve_nonnegative_quadratic_program(qp, ET());
    assert(s.solves_quadratic_program(qp));

    // Output
    if (s.is_optimal() && s.objective_value() <= max_variance) {
      cout << "Yes." << endl;
    } else {
      cout << "No." << endl;
    }
  }
 }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

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

using namespace std;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

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

int main() {

  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  //n = Total assets
  //m = number of people
  int n, m;
  for (cin >> n >> m; n != 0; cin >> n >> m) {
    Program qp(CGAL::LARGER, true, 0, false, 0);
    qp.set_r(0, CGAL::SMALLER); //cost
    qp.set_r(1, CGAL::LARGER); //expectet return

    //read cost and expected return of assets
    for (int i = 0; i < n; i++) {
      int cost, expected_return;
      cin >> cost >> expected_return;
      qp.set_a(i, 0, cost);
      qp.set_a(i, 1, expected_return);
    }
    //read covariance matrix
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        int covar;
        cin >> covar;
        if (j <= i) {
          qp.set_d(i, j, covar * 2);
        }
      }
    }

    for (int i = 0; i < m; i++) {
      int C, R, V;
      cin >> C >> R >> V;
```

```
      qp.set_b(0, C);
      qp.set_b(1, R);

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

      if (s.is_infeasible() || s.is_unbounded() || s.objective_value() > V) {
        cout << "No." << endl;
      }
      else {
        cout << "Yes." << endl;
      }
    }
  }

  return 0;
}
```

7.4 Inball

Keywords: Quadratic Program, Quadratic Program: Maximize

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}


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

int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
    int inequality_count;
    cin >> inequality_count;

    // kill switch for application
    if(inequality_count == 0) {
      break;
    }

    int dimension;
    cin >> dimension;

    // create linear programming application
    Program lp(CGAL::SMALLER, false, 0, false, 0);

    // set our target of maximixing the radius, i.e. we take the negative of the radius and minimize the whole thing
    int var_radius = dimension; // everything below ‘dimension‘ is a variable for the dimensions (?)
    lp.set_c(var_radius, -1); // maximize by minimizing the negative

    // add constraints from input
    for(int constraint_index = 0; constraint_index < inequality_count; constraint_index++) {
      int distance = 0;

      for(int dim_var_index = 0; dim_var_index < dimension; dim_var_index++) {
        int a_i;
        cin >> a_i;

        lp.set_a(dim_var_index, constraint_index, a_i);
```

```
        lp.set_a(dim_var_index, constraint_index + inequality_count, a_i);

        distance += a_i * a_i;
      }

      lp.set_a(var_radius, constraint_index, sqrt(distance));

      int b_i;
      cin >> b_i;
      lp.set_b(constraint_index, b_i);
      lp.set_b(constraint_index + inequality_count, b_i);
    }

    // find solution and print out
    Solution s = CGAL::solve_linear_program(lp, ET());
    assert(s.solves_linear_program(lp));
    if(s.is_optimal()) {
      cout << -1 * ceil_to_double(s.objective_value()) << endl;
    } else if (s.is_unbounded()) {
      cout << "inf" << endl;
    } else {
      cout << "none" << endl;
    }
  }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int n;
  for (cin >> n; n != 0; cin >> n) {
    int dimensions;
    cin >> dimensions;

    Program lp(CGAL::LARGER, false, 0, false, 0);
    for (int i = 0; i < n; ++i) {
      int sum = 0;
      for (int j = 0; j < dimensions; ++j) {
        int ai;
        cin >> ai;
        lp.set_a(j, i, ai);
        sum += pow(ai, 2);
      }
      lp.set_a(dimensions, i, -static_cast<long>(sqrt(sum)));

      int bi;
      cin >> bi;
      lp.set_b(i, -bi);
    }
    lp.set_l(dimensions, true, 0);
    lp.set_c(dimensions, -1);
```

```
    Solution s = CGAL::solve_linear_program(lp, ET());
    assert(s.solves_linear_program(lp));
    if (s.status() == CGAL::QP_OPTIMAL){
      cout << -1 * ceil_to_double(s.objective_value()) << endl;
    }
    else if (s.status() == CGAL::QP_UNBOUNDED) {
      cout << "inf" << endl;
    }
    else {
      cout << "none" << endl;
    }
  }

  return 0;
}
```

7.5 Collisions

Keywords: Point set, Delaunay Triangulation, Finite edge iteration, Triangulation with info()

```
#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/Point_set_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

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::Face_handle Face_handle;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_faces_iterator All_faces_iterator;
typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator Finite_edges_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;
typedef Triangulation::Vertex_handle Vertex_handle;

using namespace std;

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;
  for(int test = 0; test < test_count; test++) {
    int airplane_count;
    cin >> airplane_count;

    // read minimum distance
    K::FT input_dist;
    cin >> input_dist;

    K::FT min_dist = input_dist * input_dist;

    // read in airplane location
    vector<pair<Point, int> > airplanes;
    airplanes.reserve(airplane_count);
    for(int airplane_index = 0; airplane_index < airplane_count; airplane_index++) {
      int x, y;
      cin >> x >> y;

      airplanes.push_back(make_pair(Point(x, y), airplane_index));
    }

    // keeps track of planes on collision course
    vector<bool> collisions(airplane_count, false);

    // triangulate planes
    Triangulation airplane_triang;
    airplane_triang.insert(airplanes.begin(), airplanes.end());

    // iterate trough all edges
    for(Finite_edges_iterator edge_iter = airplane_triang.finite_edges_begin();
      edge_iter != airplane_triang.finite_edges_end();
      ++edge_iter) {
      // get the airplanes
      Vertex_handle plane1_vertex = edge_iter->first->vertex((edge_iter->second + 1) % 3);
      Vertex_handle plane2_vertex = edge_iter->first->vertex((edge_iter->second + 2) % 3);

      Point plane1 = plane1_vertex->point();
      Point plane2 = plane2_vertex->point();
      //cout << "plane1: " << plane1 << ", plane2: " << plane2 << endl;

      int plane1_index = plane1_vertex->info();
```

```
      int plane2_index = plane2_vertex->info();
      //cout << "\tindex: plane1: " << plane1_index << ", plane2: " << plane2_index << endl;

      // check if distance is not violated
      K::FT plane_dist = CGAL::squared_distance(plane1, plane2);
      //cout << "\tdist: " << plane_dist << endl;

      if(plane_dist < min_dist) {
        collisions.at(plane1_index) = true;
        collisions.at(plane2_index) = true;
      }
    }

    // calculate how many airplanes have a plane not far enough away
    int planes_in_danger = 0;
    for(int plane_index = 0; plane_index < airplane_count; plane_index++) {
      if(collisions.at(plane_index)) {
        planes_in_danger++;
      }
    }

    cout << planes_in_danger << endl;
  }

  return 0;
}
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_with_info_2<bool, 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::Face_handle Face_handle;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_faces_iterator All_faces_iterator;
typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator Finite_edges_iterator;
typedef Triangulation::Point Point;
typedef Triangulation::Segment Segment;

using namespace std;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int nr_testcases;
  cin >> nr_testcases;
  for (int testcase = 0; testcase < nr_testcases; testcase++) {
    int nr_planes;
    K::FT min_distance;
    cin >> nr_planes >> min_distance;
    K::FT min_distance_squared = min_distance*min_distance;

    vector<Point> points;
    points.reserve(nr_planes);
    for (int i = 0; i < nr_planes; i++) {
      int x, y;
      cin >> x >> y;
      points.push_back(Point(x, y));
```

```
    }

    Triangulation t;
    t.insert(points.begin(), points.end());

    for (Finite_vertices_iterator it = t.finite_vertices_begin(); it != t.finite_vertices_end(); ++it) {
      it->info() = false;
    }

    for (Finite_edges_iterator it = t.finite_edges_begin(); it != t.finite_edges_end(); ++it) {
      Triangulation::Vertex_handle v1 = it->first->vertex((it->second + 1) % 3);
      Triangulation::Vertex_handle v2 = it->first->vertex((it->second + 2) % 3);
      K::FT d = CGAL::squared_distance(v1->point(), v2->point());
      if (d < min_distance_squared) {
        v1->info() = true;
        v2->info() = true;
      }
    }

    int colliding_planes = 0;
    for (Finite_vertices_iterator it = t.finite_vertices_begin(); it != t.finite_vertices_end(); ++it) {
      if (it->info()) {
        colliding_planes++;
      }
    }

    cout << colliding_planes << endl;
  }

  return 0;
}
```

8 Exam Preparation

8.1 TheeV

Keywords: CGAL, Minimal Circle, Custom compare, Compare function, Binary search

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

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

// typedefs
typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Min_circle_2_traits_2<K>  Traits;
typedef CGAL::Min_circle_2<Traits>      Min_circle;
typedef K::Point_2 Point;

using namespace std;

// global vars, needed by multiple functions
Point capital;
vector<Point> cities;
vector<K::FT> distances;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

bool city_order(Point cityA, Point cityB) {
  return squared_distance(cityA, capital) > squared_distance(cityB, capital);
}

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;

  for(int t = 0; t < test_count; t++) {
    int city_count;
    cin >> city_count;

    // check for early termination
    if(city_count <= 2) {
      int ignore;

      // read in not needed information
      cin >> ignore >> ignore;
      if(city_count == 2) {
        cin >> ignore >> ignore;
      }

      // put in each of the cities one antenna with radius 0, done.
      cout << 0 << endl;
      continue;
    }

    // read in capital city
    double capital_x, capital_y;
    cin >> capital_x >> capital_y;
    capital = Point(capital_x, capital_y);

    // create vector containing all non-capital cities
    cities = vector<Point>(city_count - 1); // capital not part of it

    // read in city coordinates
    for(int city_index = 1; city_index < city_count; city_index++) { // capital already read, so start with 1, i.e.
      second city
      double city_x, city_y;
      cin >> city_x >> city_y;

      cities.at(city_index - 1) = Point(city_x, city_y);
    }

    // sort cities by descending distance from the capital
```

```
    sort(cities.begin(), cities.end(), city_order);

    // create vector with all distances for each city from the capital
    distances = vector<K::FT>(city_count - 1);
    for(int i = 0; i < city_count - 1; i++) {
      distances.at(i) = squared_distance(cities.at(i), capital);
    }

    // initiate radius temporary information
    K::FT radius1 = distances[1]; // currently the first antenna has maximal radius to contain in the furthest city
    K::FT old_radius1 = radius1;

    K::FT radius2 = 0.0; // second antenno has no radius yet
    K::FT old_radius2 = 0.0;

    // create min circle for all cities
    Min_circle min_circ_antenna2(cities.begin(), ++cities.begin(), true); // antenna 2 contains only the city farthest
        away from first anntena as a start
    Traits::Circle circ2;
    int cur_index = 1;

    // make first antenna's radius smaller and second antenna's radius larger to find an optimum
    while(radius2 < radius1) { // iterate as long as the second antenna is still smaller as the first one
      // move current radius to old radius
      old_radius1 = radius1;
      old_radius2 = radius2;

      // add next outmost city to antenna 2's reach
      min_circ_antenna2.insert(cities[cur_index]);

      // get the circle of the min circ to get information about the radius
      circ2 = min_circ_antenna2.circle();

      // update radius
      radius1 = distances[cur_index + 1]; // first antenna only has to reach one city less as the one after that is now
          covered by the second antenna
      radius2 = circ2.squared_radius();

      // update index, maybe we add another city to second antenna's reach
      cur_index++;
    }

    // we updates all four radius variables, now extract the optimum
    K::FT result = min( // we're interested in the minimum radius needed
      max(radius1, radius2), // we must take the maximum of both, as both antenna have to have the same radius!
      max(old_radius1, old_radius2) // same reason as before
      );
    cout << ceil_to_double(result) << endl;
  }
}
```

```
#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>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

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

using namespace std;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);
  int nr_testcases;
  cin >> nr_testcases;
  for (int testcase = 0; testcase < nr_testcases; testcase++) {
    int n;
```

```
    cin >> n;
    vector<Point> points(n);
    vector<K::FT> distances(n);
    for (int i = 0; i < n; i++) {
      int x, y;
      cin >> x >> y;
      points[i] = Point(x, y);
      distances[i] = CGAL::squared_distance(points[0], points[i]);
    }
    vector<K::FT> distances_sorted = distances;
    sort(distances_sorted.begin(), distances_sorted.end());

    int min_bound = 0;
    int max_bound = distances_sorted.size() - 1;
    while (min_bound != max_bound) {
      int half = (min_bound + max_bound) / 2;
      K::FT squared_dist = distances_sorted[half];
      //cout << "min = " << min_bound << " max = " << max_bound << " half = " << half << " squared_dist = " <<
          squared_dist << endl;

      vector<Point> uncovered_points;
      for (int i = 0; i < n; i++) {
        if (distances[i] > squared_dist) {
          //cout << "adding point" << points[i] << endl;
          uncovered_points.push_back(points[i]);
        }
      }
      //cout << uncovered_points.size() << endl;
      Min_circle mc(uncovered_points.begin(), uncovered_points.end(), true);
      //cout << "valid = " << mc.is_valid() << endl;
      K::FT second_rad_squared = mc.circle().squared_radius();
      //cout << "second_rad_squared = " << second_rad_squared << endl;
      if (second_rad_squared > squared_dist) {
        //cout << "increase lower bound" << endl;
        if (second_rad_squared < distances_sorted[half + 1]) {
          max_bound = min_bound = half;
        }
        else {
          min_bound = half + 1;
        }
      }
      else {
        //cout << "decrease upper bound" << endl;
        max_bound = half;
      }
    }

    K::FT squared_dist = distances_sorted[min_bound];
    vector<Point> uncovered_points;
    for (int i = 0; i < n; i++) {
      if (distances[i] > squared_dist) {
        uncovered_points.push_back(points[i]);
      }
    }
    Min_circle mc(uncovered_points.begin(), uncovered_points.end(), true);
    K::FT second_rad_squared = mc.circle().squared_radius();
    cout << ceil_to_double(max(squared_dist, second_rad_squared)) << endl;
  }

  return 0;
}
```

8.2 Algocoön Group

Keywords: BGL, Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

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

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, int,
    property<edge_residual_capacity_t, int,
    property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

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;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_count;
    cin >> test_count;

    for(int test_index = 0; test_index < test_count; test_index++) {
        int figure_count, limb_count;
        cin >> figure_count >> limb_count;

        // create graph
        Graph graph(figure_count);

        // get graph's properties
        EdgeCapacityMap capacity = get(edge_capacity, graph);
        ReverseEdgeMap reverse_edge_map = get(edge_reverse, graph);
        ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);

        // read in limbs
        for(int limb_index = 0; limb_index < limb_count; limb_index++) {
            int from, to, cost;
            cin >> from >> to >> cost;

            Edge edge;
            Edge rev_edge;
            tie(edge, tuples::ignore) = add_edge(from, to, graph);
            tie(rev_edge, tuples::ignore) = add_edge(to, from, graph);
            capacity[edge] = cost;
            capacity[rev_edge] = 0;
            reverse_edge_map[edge] = rev_edge;
            reverse_edge_map[rev_edge] = edge;
        }

        // find bist source and sink
        int best_source = -1;
        int best_sink = -1;
        int best_value = numeric_limits<int>::max();

        // attention: start at 1, otherwise assertion because sink == source
        for(int figure_index = 1; figure_index < figure_count; figure_index++) {
            // search for best sink
            int max_flow = push_relabel_max_flow(graph, 0, figure_index);
            if(max_flow < best_value) {
                best_value = max_flow;
                best_source = 0;
                best_sink = figure_index;
            }

            // search for best source
            max_flow = push_relabel_max_flow(graph, figure_index, 0);
            if(max_flow < best_value) {
                best_value = max_flow;
                best_source = figure_index;
                best_sink = 0;
            }
        }

        // rerun for found best sink and source
        push_relabel_max_flow(graph, best_source, best_sink);

        std::queue<int> Q;
        Q.push(best_source);

        vector<bool> visited(figure_count, false);
        visited.at(best_source) = true;

        while(!Q.empty()) {
            const int figure = Q.front();
            Q.pop();
            graph_traits<Graph>::out_edge_iterator out_iter, out_end;
            for(tie(out_iter, out_end) = out_edges(figure, graph); out_iter != out_end; ++out_iter) {
                const int edge_end_v = target(*out_iter, graph);

                if(res_capacity[*out_iter] == 0 || visited[edge_end_v]) {
                    continue;
                }

                visited[edge_end_v] = true;
                Q.push(edge_end_v);
            }
        }

        cout << best_value << endl << count(visited.begin(), visited.end(), true);
        for(int i = 0; i < figure_count; i++) {
            if(visited[i]) {
                cout << " " << i;
            }
        }
        cout << endl;
    }
```

```
    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

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

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualMap;

inline void add_flow_edge(int start, int end, int c, EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge, Graph& g) {
  Edge e, rev_e;
  bool success;
  tie(e, success) = add_edge(start, end, g);
  if (success) {
    tie(rev_e, success) = add_edge(end, start, g);
    capacity[e] = c;
    capacity[rev_e] = 0;
    rev_edge[e] = rev_e;
    rev_edge[rev_e] = e;
  }
  else {
    capacity[e] += c;
  }
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_figures, nr_limbs;
    cin >> nr_figures >> nr_limbs;

    Graph g(nr_figures);

    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);
    ResidualMap residual_map = get(edge_residual_capacity, g);

    for (int i = 0; i < nr_limbs; i++) {
      int from, to, cost;
      cin >> from >> to >> cost;
      add_flow_edge(from, to, cost, capacity, rev_edge, g);
    }

    int best_sink;
    int best_source;
    int min_flow = numeric_limits<int>::max();
    for (int i = 1; i < nr_figures; i++) {
      int flow = push_relabel_max_flow(g, 0, i);
      if (flow < min_flow) {
        min_flow = flow;
        best_source = 0;
        best_sink = i;
      }
      flow = push_relabel_max_flow(g, i, 0);
      if (flow < min_flow) {
        min_flow = flow;
        best_source = i;
        best_sink = 0;
      }
    }

    min_flow = push_relabel_max_flow(g, best_source, best_sink);

    vector<int> vis(nr_figures, false);
    vis[best_source] = true;
    std::queue<int> to_visit;
    to_visit.push(best_source);
```

```
    while (!to_visit.empty()) {
      const int u = to_visit.front();
      to_visit.pop();

      graph_traits<Graph>::out_edge_iterator ebeg, eend;
      for (tie(ebeg, eend) = out_edges(u, g); ebeg != eend; ++ebeg) {
        const int v = target(*ebeg, g);
        if (residual_map[*ebeg] == 0 || vis[v]) {
          continue;
        }
        vis[v] = true;
        to_visit.push(v);
      }
    }

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

  return EXIT_SUCCESS;
}
```

8.3 Monkey Island

Keywords: BGL, Strong component, Edge iteration, Uses class, Custom compare, Compare function, Greedy, DFS (Graph)

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/strong_components.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_iterator EdgeIterator;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int i = 0; i < test_cases; i++) {
        int location_count, road_count;
        cin >> location_count >> road_count;

        // create graph
        Graph graph(location_count);

        // read in directed roads and create edges for them
        for(int j = 0; j < road_count; j++) {
            // read vertices information
            int from, to;
            cin >> from >> to;

            // locations start at 1, inside our graph at 0
            from -= 1;
            to -= 1;

            // create edge
            add_edge(from, to, graph);
        }

        // read in costs for building police station at a vertex
        vector<int> costs(location_count);
        for(int j= 0; j < location_count; j++) {
            cin >> costs.at(j);
            //cout << "-> " << costs.at(j) << endl;;
        }

        // retrieve strong components (maximum set of vertices for which we have a path from any pair in this set).
        vector<int> scc(location_count); // each vertex is labeled with an int representing the set the vertex belongs to
             (to which strong component set)
        int scc_count = strong_components(graph, &scc[0]);
```

```
        // records which strong component set has an incoming edge from an other strong component set
        vector<bool> incoming(scc_count, false);

        // iterate over all graph edges
        EdgeIterator edge_iter, edge_end;
        for(tie(edge_iter, edge_end) = edges(graph); edge_iter != edge_end; ++edge_iter) {
            // get vertices connected by the edge
            int from_vertex = source(*edge_iter, graph);
            int to_vertex = target(*edge_iter, graph);

            // check that both vertices are not part of the same strong component
            if(scc[from_vertex] != scc[to_vertex]) {
                // ok, so both vertices don't have edges in both directions.
                // furthermore we have a *directed* edge from the strong component set of from_vertex to the one of
                    to_vertex, record that we have an incoming
                // edge.
                incoming[scc[to_vertex]] = true;
            }
        }

        // so, now we know which strong component set has incoming edges from other sets. Therefore we don't have to
            build a police station inside a strong component
        // set that has an incoming edge! Reason: we can build one in the strong component set that has an edge to the
            other set and with that we reach all vertices in
        // both strong component sets.

        // keeps track of minimum costs needed
        vector<int> min_costs(scc_count, numeric_limits<int>::max());

        // iterate over all locations
        for(int location_index = 0; location_index < location_count; location_index++) {
            if(!incoming[scc[location_index]]) {
                // the strong component set the location belongs to has *no* incoming edge.

                // assign the minimum cost for the strong component set the current location belongs to is defined
                // by the smaller number from the set {already known minimum, cost to build in the current location a
                    police station}
                //cout << "min_costs = " << min(min_costs[scc[location_index]], costs[location_index]) << " ( " <<
                    min_costs[scc[location_index]] << " or " << costs[location_index] << " )" << endl;
                min_costs[scc[location_index]] = min(min_costs[scc[location_index]], costs[location_index]);
            }
        }

        // sum up the costs for all police stations, jumping over strong component sets with incoming edges!
        int cost_sum = 0;
        for(int scc_index = 0; scc_index < scc_count; scc_index++) {
            if(!incoming[scc_index]) {
                //cout << "==> " << cost_sum << " + " << min_costs[scc_index] << " = ";
                cost_sum += min_costs[scc_index];
                //cout << cost_sum << endl;
            }
        }

        // out with result
        cout << cost_sum << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <queue>
#include <limits>
#include <stack>

using namespace std;

class Location {
public:
  int cost, id;
  bool visited;
  vector<Location*> roads;

  Location(int id)
    : cost(0), id(id), visited(false), roads(vector<Location*>()) {}
};

bool sortLocations(const Location* lhs, const Location* rhs) {
  return lhs->cost < rhs->cost;
}

int main(void) {
  int cases;
  cin >> cases;

  for (int c = 0; c < cases; c++) {
    int nr_locations;
    cin >> nr_locations;
    int nr_roads;
    cin >> nr_roads;
```

```
    vector<Location*> locations(nr_locations, NULL);
    for (int i = 0; i < nr_locations; i++) {
      locations[i] = new Location(i + 1);
    }
    for (int i = 0; i < nr_roads; i++) {
      int from, to;
      cin >> from >> to;
      locations[from - 1]->roads.push_back(locations[to - 1]);
    }
    for (int i = 0; i < nr_locations; i++) {
      int cost;
      cin >> cost;
      locations[i]->cost = cost;
    }

    sort(locations.begin(), locations.end(), sortLocations);

    int cost = 0;
    for (int i = 0; i < nr_locations; i++) {
      Location* currentLoc = locations[i];
      //cout << "Location " << currentLoc->id << " with cost " << currentLoc->cost << endl;
      if (!currentLoc->visited) {
        cost += currentLoc->cost;
        int tmpCost = currentLoc->cost;
        stack<Location*> locToVisit;
        locToVisit.push(currentLoc);
        vector<bool> vis(nr_locations, false);
        while (!locToVisit.empty()) {
          Location* nextLocation = locToVisit.top();
          locToVisit.pop();
          if (!vis[nextLocation->id]) {
            if (nextLocation->visited) {
              if (nextLocation->cost > 0) {
                //cout << "Reducing cost by " << nextLocation->cost << endl;
                cost -= nextLocation->cost;
                nextLocation->cost = 0;
              }
            }
            else {
              nextLocation->visited = true;
              nextLocation->cost = 0;
            }
            vis[nextLocation->id] = true;
            for (unsigned int j = 0; j < nextLocation->roads.size(); j++) {
              locToVisit.push(nextLocation->roads[j]);
            }
          }
        }
        currentLoc->cost = tmpCost;
      }
    }

    cout << cost << endl;
    for (int i = 0; i < nr_locations; i++) {
      delete locations[i];
    }
  }
}
```

8.4 Odd Route

Keywords: ACM, BFS (Graph), Custom BFS

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

using namespace std;

int main() {
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        // get graph properties
        int vertex_count, edge_count;
        cin >> vertex_count >> edge_count;

        // get start and target vertices
        int start_vertex, target_vertex;
        cin >> start_vertex >> target_vertex;

        // read in vertices
        vector<Vertex> vertices(vertex_count);
        for(int edge_index = 0; edge_index < edge_count; edge_index++) {
            int from, to, weight;
            cin >> from >> to >> weight;

            vertices.at(from).edges.push_back(make_pair(to, weight));
```

```cpp
        }

        priority_queue<NextMove, vector<NextMove>, MoveCompare> to_visit;
        to_visit.push(NextMove(start_vertex, 0, 0));

        int shortest_weight_sum = numeric_limits<int>::max();

        while(!to_visit.empty()) {
            NextMove cur_move = to_visit.top();
            to_visit.pop();

            if(cur_move.is_odd_weight()) {
                if(cur_move.is_odd_length()) {
                    // odd weight, odd length
                    if(vertices.at(cur_move.current_vertex).visited_odd_edges_odd_weight) {
                        // next edge was already visited with odd weight and odd length, do not revisit
                        continue;
                    } else {
                        if(cur_move.current_vertex == target_vertex) {
                            // found the end with searched configuration, update shortest weight
                            shortest_weight_sum = cur_move.weight_sum;
                            break;
                        }

                        // visited it now with the configuration
                        vertices[cur_move.current_vertex].visited_odd_edges_odd_weight = true;
                    }
                } else {
                    // odd weight, even length
                    if(vertices.at(cur_move.current_vertex).visited_even_edges_odd_weight) {
                        // already visited with that configuration
                        continue;
                    } else {
                        vertices.at(cur_move.current_vertex).visited_even_edges_odd_weight = true;
                    }
                }
            } else {
                if(cur_move.is_odd_length()) {
                    // even weight, odd length
                    if(vertices.at(cur_move.current_vertex).visited_odd_edges_even_weight) {
                        // already visited with that configuration
                        continue;
                    } else {
                        vertices.at(cur_move.current_vertex).visited_odd_edges_even_weight = true;
                    }
                } else {
                    // even weight, even length
                    if(vertices.at(cur_move.current_vertex).visited_even_edges_even_weight) {
                        // already visited with that configuration
                        continue;
                    } else {
                        vertices.at(cur_move.current_vertex).visited_even_edges_even_weight = true;
                    }
                }
            }

            // ok, if we reach this point, we found an edge that we didn't visit in the current configuration,
            // have to visit it with current configuration
            vector<pair<int, int> >& edges = vertices.at(cur_move.current_vertex).edges;
            for(int next_edge = 0; next_edge < edges.size(); next_edge++) {
                to_visit.push(NextMove(edges.at(next_edge).first, // use the next vertex that can be reached by current
                     vertex
                    cur_move.weight_sum + edges.at(next_edge).second, // add weight of the edge we would follow
                    cur_move.edges_length + 1)); // we use an edge more, wow!
            }
        }

        if(shortest_weight_sum == numeric_limits<int>::max()) {
            cout << "no" << endl;
        } else {
            cout << shortest_weight_sum - 1 << endl;
        }
    }

    return 0;
}
```

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

using namespace std;

class Vertex {
public:
  bool v_even_edges_even_weight;
  bool v_even_edges_odd_weight;
  bool v_odd_edges_even_weight;
  bool v_odd_edges_odd_weight;

  vector<pair<int, int> > edges;
```

```
  Vertex() {
    v_even_edges_even_weight = false;
    v_even_edges_odd_weight = false;
    v_odd_edges_even_weight = false;
    v_odd_edges_odd_weight = false;
  }
};

class Move {
public:
  int current_vertex;
  int weight_sum;
  int edges_sum;

  Move(int current_vertex, int weight_sum, int edges_sum)
    : current_vertex(current_vertex), weight_sum(weight_sum), edges_sum(edges_sum)
  { }

  bool is_odd_weight() { return weight_sum % 2; }
  bool is_odd_edges() { return edges_sum % 2; }
};

struct MoveComparator {
  bool operator() (const Move& lhs, const Move& rhs) {
    return lhs.weight_sum > rhs.weight_sum;
  }
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_vertices, nr_edges;
    cin >> nr_vertices >> nr_edges;
    vector<Vertex> vertices(nr_vertices);

    int start, end;
    cin >> start >> end;

    for (int i = 0; i < nr_edges; i++) {
      int from, to, weight;
      cin >> from >> to >> weight;
      vertices[from].edges.push_back(make_pair(to, weight));
    }

    priority_queue<Move, vector<Move>, MoveComparator> to_visit;
    to_visit.push(Move(start, 0, 0));
    int shortest_weight_sum = -1;
    while (!to_visit.empty()) {
      Move cur_move = to_visit.top();
      to_visit.pop();
      if (cur_move.is_odd_weight()) {
        if (cur_move.is_odd_edges()) {
          if (vertices[cur_move.current_vertex].v_odd_edges_odd_weight) {
            continue;
          }
          else {
            if (cur_move.current_vertex == end) {
              shortest_weight_sum = cur_move.weight_sum;
              break;
            }
            vertices[cur_move.current_vertex].v_odd_edges_odd_weight = true;
          }
        }
        else {
          if (vertices[cur_move.current_vertex].v_even_edges_odd_weight) {
            continue;
          }
          else {
            vertices[cur_move.current_vertex].v_even_edges_odd_weight = true;
          }
        }
      }
      else {
        if (cur_move.is_odd_edges()) {
          if (vertices[cur_move.current_vertex].v_odd_edges_even_weight) {
            continue;
          }
          else {
            vertices[cur_move.current_vertex].v_odd_edges_even_weight = true;
          }
        }
        else {
          if (vertices[cur_move.current_vertex].v_even_edges_even_weight) {
            continue;
          }
          else {
            vertices[cur_move.current_vertex].v_even_edges_even_weight = true;
          }
        }
      }
      vector<pair<int, int> >& edges = vertices[cur_move.current_vertex].edges;
```

```
      for (int i = 0; i < edges.size(); i++) {
        to_visit.push(Move(edges[i].first, cur_move.weight_sum + edges[i].second, cur_move.edges_sum + 1));
      }
    }

    if (shortest_weight_sum == -1) {
      cout << "no" << endl;
    }
    else {
      cout << shortest_weight_sum << endl;
    }
  }

  return 0;
}
```

8.5 Divisor Distance

Keywords: Prime Sieve, Next Prime, ACM

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

using namespace std;

#define MAX_PRIM 10000000
bitset<MAX_PRIM> is_prime;

void prime_sieve() {
  is_prime.set(); // sets all to true
  // 0, 1 not prime
  is_prime.reset(0);
  is_prime.reset(1);

  for(int prime = 2; prime * prime <= MAX_PRIM; prime++) {
    if(is_prime.test(prime)) {
      // is a prime number, so all multiples of it are not
      for(int multiple = prime + prime; multiple < MAX_PRIM; multiple += prime) {
        is_prime.reset(multiple); // set multiple to false, as it is not prime
      }
    }
  }
}

int get_next_prime(int cur) {
  for(int i = cur + 1; i * i < MAX_PRIM; i ++) {
    if(is_prime.test(i)) {
      return i;
    }
  }

  return 0;
}

int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    prime_sieve();

    for(int test = 0; test < test_cases; test++) {
      int max_nr, vertices_pair_count;
      cin >> max_nr >> vertices_pair_count;

      for(int vertices_pair_index = 0; vertices_pair_index < vertices_pair_count; vertices_pair_index++) {
        int from, to;
        cin >> from >> to;

        int length = 0;
        int a_factor = 2;
        int b_factor = 2;

        // divise `from` and `to` with prime numbers as long as they do not match
        while(from != to) {
          // always lower the current "maximum" of `from` and `to`
          if(from > to) {
            if(is_prime.test(from)) {
              from = 1;
            } else {
              // find a divisor for current `from`
              while(from % a_factor != 0) {
                a_factor = get_next_prime(a_factor);
              }

              // found a divisor for `from`, " " remove " " it
              from = from / a_factor;
            }
```

```
        } else {
          // following code works the same as for 'from' and 'a_factor'
          if(is_prime.test(to)) {
            to = 1;
          } else {
            while(to % b_factor != 0) {
              b_factor = get_next_prime(b_factor);
            }

            to = to / b_factor;
          }
        }

        length++;
      }

      cout << length << endl;
    }
  }

  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <queue>
#include <limits>
#include <cmath>

using namespace std;


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

  const int max_n = 10000000;
  const int sqrt_max_n = sqrt(max_n) + 1;
  vector<bool> is_prime(max_n, true);
  vector<int> primes;

  //cout << "Calculating primes" << endl;
  is_prime[0] = false;
  is_prime[1] = false;
  for (int i = 2; i <= sqrt_max_n; i++) {
    if (is_prime[i]) {
      for (int j = 2 * i; j <= max_n; j += i) {
        is_prime[j] = false;
      }
    }
  }

  //cout << "Adding primes to vector" << endl;
  for (int i = 0; i <= max_n; i++) {
    if (is_prime[i]) {
      primes.push_back(i);
    }
  }

  int tests;
  cin >> tests;

  //cout << "Starting tests" << endl;
  for (int test = 0; test < tests; test++) {
    int n, cases;
    cin >> n >> cases;
    for (int c = 0; c < cases; c++) {
      int v1, v2, hops = 0;
      cin >> v1 >> v2;
      while (v1 != v2) {
        if (v1 > v2) {
          //cout << "Changing " << v1 << " to " << largest_div[v1] << endl;
          if (is_prime[v1]) {
            v1 = 1;
          }
          else {
            for (unsigned int j = 0; j < primes.size(); j++) {
              if (v1 % primes[j] == 0) {
                v1 = v1 / primes[j];
                break;
              }
            }
          }
          //v1 = largest_div[v1];
          hops++;
        }
        else {
          //cout << "Changing " << v2 << " to " << largest_div[v2] << endl;
          if (is_prime[v2]) {
            v2 = 1;
          }
          else {
            for (unsigned int j = 0; j < primes.size(); j++) {
              if (v2 % primes[j] == 0) {
```

```
                v2 = v2 / primes[j];
                break;
              }
            }
          }
          //v2 = largest_div[v2];
          hops++;
        }
      }
      cout << hops << endl;
    }
  }
}
```

8.6 Portfolios Revisited

Keywords: CGAL, Quadratic Program, Non-negative quadratic program, Exponential bound search, Binary search

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}


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

bool feasible(int asset_count, int max_cost, int max_covar, int expected_return, vector<int>& asset_costs, vector<int>&
    asset_returns, vector<vector<int> >& asset_covar) {
  // use quadratic programming to check if given parameters result in a feasable result, i.e. given parameters result in
      a possible solution

  // create QP program: lower bound is 0 for amount of assets bough and generally we're interested in <= inequastions
  Program qp(CGAL::SMALLER, true, 0, false, 0);

  // restrict total cost
  for(int asset_index = 0; asset_index < asset_count; asset_index++) {
    qp.set_a(asset_index, 0, asset_costs.at(asset_index)); // sum of all asset's cost that were bought must be ...
  }

  qp.set_b(0, max_cost); // ... <= maximum cost investor is ready to pay

  // restrict how much expected return we expect, be careful as our general operation is '<=', we have to switch it here,
      as we wan't at least some amount of
  // expected return
  for(int asset_index = 0; asset_index < asset_count; asset_index++) {
    qp.set_a(asset_index, 1, asset_returns[asset_index]); // sum of all asset's expected return that were bought must be
        ...
  }
  qp.set_r(1, CGAL::LARGER); // ... >= ...
  qp.set_b(1, expected_return); // the expected return

  // set the objective function:
  // total portfolio variance must be not too large
  for(int assetA_index = 0; assetA_index < asset_count; assetA_index++) {
    for(int assetB_index = 0; assetB_index <= assetA_index; assetB_index++) { // careful: '<= assetA_index'!
      qp.set_d(assetA_index, assetB_index, 2 * asset_covar.at(assetA_index).at(assetB_index)); // careful: don't forget
          the 2 * because of how the matrix works...
    }
  }

  // find solution
  Solution sol = CGAL::solve_nonnegative_quadratic_program(qp, ET());
  return sol.is_optimal() && sol.objective_value() <= max_covar;
}
```

```
int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  while(true) {
    int asset_count, friend_count;
    cin >> asset_count >> friend_count;

    // kill switch
    if(asset_count == 0 && friend_count == 0) {
      break;
    }

    // collect asset costs and risk factors
    vector<int> asset_costs(asset_count);
    vector<int> asset_returns(asset_count);
    for(int asset_index = 0; asset_index < asset_count; asset_index++) {
      cin >> asset_costs.at(asset_index) >> asset_returns.at(asset_index);
    }

    // collect covariance between two assets
    vector<vector<int > > asset_covar(asset_count, vector<int>(asset_count));
    for(int assetA_index = 0; assetA_index < asset_count; assetA_index++) {
      for(int assetB_index = 0; assetB_index < asset_count; assetB_index++) {
        cin >> asset_covar.at(assetA_index).at(assetB_index);
      }
    }

    // go trough investors and calculate maximum expected outcome for given conditions
    for(int investor_index = 0; investor_index < friend_count; investor_index++) {
      int max_cost, max_covar;
      cin >> max_cost >> max_covar;

      // search for upper bound for return, so we don't have to search through every possible combination
      int r = 1;
      while(feasible(asset_count, max_cost, max_covar, r, asset_costs, asset_returns, asset_covar)) {
        r *= 2;
      }

      // now we know in which range ([r/2, r]) to search for the maximum
      // use binary search to find it
      int ok = r / 2;
      int low = r / 2;
      int high = r;
      while(low <= high) {
        r = (high - low) / 2 + low;
        if(feasible(asset_count, max_cost, max_covar, r, asset_costs, asset_returns, asset_covar)) {
          ok = r;
          low = r + 1;
        } else {
          high = r - 1;
        }
      }

      cout << ok << endl;
    }
  }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

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

using namespace std;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
```

```
}

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

bool is_possible(Program& qp, int expected_return, int max_variance) {
  qp.set_b(1, expected_return);

  Solution s = CGAL::solve_nonnegative_quadratic_program(qp, ET());
  assert(s.solves_quadratic_program(qp));
  return !(s.is_infeasible() || s.is_unbounded() || s.objective_value() > max_variance);
}

int main() {

  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int nr_assets, nr_friends;
  for (cin >> nr_assets >> nr_friends; nr_assets != 0 && nr_friends != 0; cin >> nr_assets >> nr_friends) {
    Program qp(CGAL::LARGER, true, 0, false, 0);
    qp.set_r(0, CGAL::SMALLER); //cost
    qp.set_r(1, CGAL::LARGER); //expectet return

    for (int i = 0; i < nr_assets; ++i) {
      int asset_cost, asset_return;
      cin >> asset_cost >> asset_return;
      qp.set_a(i, 0, asset_cost);
      qp.set_a(i, 1, asset_return);
    }

    for (int i = 0; i < nr_assets; ++i) {
      for (int j = 0; j < nr_assets; ++j) {
        int asset_covariance;
        cin >> asset_covariance;
        if (j <= i) {
          qp.set_d(i, j, asset_covariance * 2);
        }
      }
    }

    for (int friend_id = 0; friend_id < nr_friends; ++friend_id) {
      int max_invest, max_variance;
      cin >> max_invest >> max_variance;
      qp.set_b(0, max_invest);

      int expected_return = 1;
      while (is_possible(qp, expected_return, max_variance)) {
        expected_return *= 2;
      }
      int ok = expected_return / 2;
      int min_bound = expected_return / 2;
      int max_bound = expected_return;
      while (min_bound <= max_bound) {
        int middle = (max_bound - min_bound) / 2 + min_bound;
        if (is_possible(qp, middle, max_variance)) {
          ok = middle;
          min_bound = middle + 1;
        }
        else {
          max_bound = middle - 1;
        }
      }
      cout << ok << endl;
    }
  }
  return 0;
}
```

8.7 Tetris

Keywords: BGL, Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>

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

using namespace std;
```

```
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, unsigned int,
    property<edge_residual_capacity_t, unsigned int,
    property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

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;

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test_index = 0; test_index < test_cases; test_index++) {
        int game_width, brick_count;
        cin >> game_width >> brick_count;

        // define index of source and sink vertex
        int source_index = game_width + 1;
        int sink_index = game_width;

        // create graph, don't forget we need a source and sink too
        Graph graph(game_width * 2 + 1);

        // get graph's properties
        EdgeCapacityMap capacity = get(edge_capacity, graph);
        ReverseEdgeMap rev_edge = get(edge_reverse, graph);
        // not used: ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);


        // create edges for splits, as only one can exist at any location. So add edges with weight 1
        for(int game_location = 0; game_location < game_width; game_location++) {
            bool new_edge;
            Edge edge, reverse_edge;
            tie(edge, new_edge) = add_edge(game_location, game_width + game_location + 1, graph);
            tie(reverse_edge, new_edge) = add_edge(game_width + game_location + 1, game_location, graph);

            capacity[edge] = 1;
            capacity[reverse_edge] = 0;
            rev_edge[edge] = reverse_edge;
            rev_edge[reverse_edge] = edge;
        }

        // read in widths of blocks
        for(int brick_index = 0; brick_index < brick_count; brick_index++) {
            int start, end;
            cin >> start >> end;

            // make sure we get expected order
            if(start > end) {
                int tmp = end;
                end = start;
                start = tmp;
            }

            // check if we stay inside the game field
            if(end > game_width) {
                continue;
            }

            // add edge for brick
            bool new_edge;
            Edge edge, reverse_edge;
            tie(edge, new_edge) = add_edge(start + game_width + 1, end, graph);
            tie(reverse_edge, new_edge) = add_edge(end, start + game_width + 1, graph);

            capacity[edge] = 1;
            capacity[reverse_edge] = 0;
            rev_edge[edge] = reverse_edge;
            rev_edge[reverse_edge] = edge;
        }

        long max_flow = push_relabel_max_flow(graph, source_index, sink_index);
        cout << max_flow << endl;
    }

    return 0;
}
```

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

```
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int width, nr_bricks;
    cin >> width >> nr_bricks;

    Graph g(width * 2 + 1);
    int source = width + 1;
    int sink = width;
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);

    Edge e, rev_e;
    bool success;
    for (int i = 1; i < width; i++) {
      tie(e, success) = add_edge(i, width + i + 1, g);
      tie(rev_e, success) = add_edge(width + i + 1, i, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    for (int i = 0; i < nr_bricks; i++) {
      int first_end, second_end;
      cin >> first_end >> second_end;
      if (first_end > second_end) {
        int tmp = first_end;
        first_end = second_end;
        second_end = tmp;
      }
      tie(e, success) = add_edge(width + 1 + first_end, second_end, g);
      tie(rev_e, success) = add_edge(second_end, width + 1 + first_end, g);
      capacity[e] = 1;
      capacity[rev_e] = 0;
      rev_edge[e] = rev_e;
      rev_edge[rev_e] = e;
    }

    int flow = push_relabel_max_flow(g, source, sink);
    cout << flow << endl;
  }

  return 0;
}
```

8.8 Stamp Exhibition

Keywords: CGAL, Quadratic Program

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
```

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

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;

typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

using namespace CGAL;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;

  for (int test = 0; test < test_count; test++) {
    int lamp_count, stamp_count, wall_count;
    cin >> lamp_count >> stamp_count >> wall_count;

    // read in lamps
    vector<Point> lamps;
    lamps.reserve(lamp_count);

    for(int lamp_index = 0; lamp_index < lamp_count; lamp_index++) {
      int lamp_x, lamp_y;
      cin >> lamp_x >> lamp_y;

      lamps.push_back(Point(lamp_x, lamp_y));
    }

    // collect lamps and the maximum light they are allowed to get
    vector<Point> stamps;
    stamps.reserve(stamp_count);

    vector<int> stamp_max_light;
    stamp_max_light.reserve(stamp_count);

    for(int stamp_index = 0; stamp_index < stamp_count; stamp_index++) {
      int stamp_x, stamp_y, max_light;
      cin >> stamp_x >> stamp_y >> max_light;

      stamps.push_back(Point(stamp_x, stamp_y));
      stamp_max_light.push_back(max_light);
    }

    // read in walls
    vector&lt;Segment&gt; walls;
    walls.reserve(wall_count);
    for(int wall_index = 0; wall_index < wall_count; wall_index++) {
      int wall_start_x, wall_start_y, wall_end_x, wall_end_y;
      cin >> wall_start_x >> wall_start_y >> wall_end_x >> wall_end_y;

      walls.push_back(Segment(Point(wall_start_x, wall_start_y), Point(wall_end_x, wall_end_y)));
    }

    // Create quadratic programming instance
    Program qp(CGAL::SMALLER, true, 1, true, 4096);
    for(int stamp_index = 0; stamp_index < stamp_count; stamp_index++) {
      for(int lamp_index = 0; lamp_index < lamp_count; lamp_index++) {
        Segment light_to_stamp(lamps.at(lamp_index), stamps.at(stamp_index));

        // check if a wall blocks the light beam from lamp to stamp
        bool light_blocked = false;
        for(int wall_index = 0; wall_index < wall_count; wall_index++) {
```

```
          if(do_intersect(light_to_stamp, walls.at(wall_index))) {
            light_blocked = true;
            break;
          }
        }

        if(!light_blocked) {
          double d = CGAL::to_double(1 / light_to_stamp.squared_length());
          qp.set_a(lamp_index, stamp_index, d);
          qp.set_a(lamp_index, stamp_count + stamp_index, d);
        }
      }

      qp.set_b(stamp_index, stamp_max_light.at(stamp_index));
      qp.set_b(stamp_count + stamp_index, 1);
      qp.set_r(stamp_count + stamp_index, CGAL::LARGER);
    }

    // get solution
    Solution sol = solve_linear_program(qp, ET());
    if(sol.is_infeasible()) {
      cout << "no" << endl;
    } else {
      cout << "yes" << endl;
    }
  }

  return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Segment_2 Segment;

typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;

using namespace CGAL;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_lights, nr_stamps, nr_walls;
    cin >> nr_lights >> nr_stamps >> nr_walls;

    vector<Point> lights;
    lights.reserve(nr_lights);
    for (int i = 0; i < nr_lights; i++) {
      int light_x, light_y;
      cin >> light_x >> light_y;
```

```
      lights.push_back(Point(light_x, light_y));
    }
    vector<Point> stamps;
    vector<int> stamps_max_int;
    stamps.reserve(nr_stamps);
    stamps_max_int.reserve(nr_stamps);
    for (int i = 0; i < nr_stamps; i++) {
      int stamp_x, stamp_y, stamp_max_light_intensity;
      cin >> stamp_x >> stamp_y >> stamp_max_light_intensity;
      stamps.push_back(Point(stamp_x, stamp_y));
      stamps_max_int.push_back(stamp_max_light_intensity);
    }
    vector&lt;Segment&gt; walls;
    walls.reserve(nr_walls);
    for (int i = 0; i < nr_walls; i++) {
      int wall_start_x, wall_start_y, wall_end_x, wall_end_y;
      cin >> wall_start_x >> wall_start_y >> wall_end_x >> wall_end_y;
      walls.push_back(Segment(Point(wall_start_x, wall_start_y), Point(wall_end_x, wall_end_y)));
    }

    Program qp(CGAL::SMALLER, true, 1, true, 4096);
    for (int i = 0; i < nr_stamps; i++) {
      for (int j = 0; j < nr_lights; j++) {
        Segment light_to_stamp(lights[j], stamps[i]);
        bool blocked = false;
        for (int k = 0; k < nr_walls; k++) {
          if (do_intersect(light_to_stamp, walls[k])) {
            blocked = true;
            break;
          }
        }
        if (!blocked) {
          double d = CGAL::to_double(1 / light_to_stamp.squared_length());
          qp.set_a(j, i, d);
          qp.set_a(j, nr_stamps + i, d);
        }
      }
      qp.set_b(i, stamps_max_int[i]);
      qp.set_b(nr_stamps + i, 1);
      qp.set_r(nr_stamps + i, CGAL::LARGER);
    }
    Solution sol = solve_linear_program(qp, ET());
    assert(sol.solves_linear_program(qp));
    if (sol.is_infeasible()) {
      cout << "no" << endl;
    }
    else {
      cout << "yes" << endl;
    }
  }

  return 0;
}
```

8.9 Placing Knights

Keywords: BGL, Matching

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>
#include <string>

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

using namespace std;
using namespace boost;

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

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        int width;
        cin >> width;

        // 2D vector indicating if the field is not a hole
        vector<bool> can_occupy(width * width);
```

```
        int hole_count = 0;
        for(int row = 0; row < width; row++) {
            for(int column = 0; column < width; column++) {
                int indicator;
                cin >> indicator;

                if(indicator == 1) {
                    can_occupy.at(row + column * width) = true;
                } else {
                    hole_count++;
                    can_occupy.at(row + column * width) = false;
                }
            }
        }

        // create graph for game field
        Graph graph(width * width);

        // create edges
        for(int row = 0; row < width; row++) {
            for(int column = 0; column < width; column++) {
                int field_index = row + column * width;

                if(!can_occupy.at(field_index)) {
                    continue;
                }

                // i = row, j = column
                // [i - 1, j - 2]
                if(row - 1 >= 0 && column - 2 >= 0 && can_occupy.at((row - 1) + (column - 2) * width)) {
                    add_edge(field_index, (row - 1) + (column - 2) * width, graph);
                }

                // [i - 1, j + 2]
                if(row - 1 >= 0 && column + 2 < width && can_occupy.at((row - 1) + (column + 2) * width)) {
                    add_edge(field_index, (row - 1) + (column + 2) * width, graph);
                }

                // [i + 1, j - 2]
                if(row + 1 < width && column - 2 >= 0 && can_occupy.at((row + 1) + (column - 2) * width)) {
                    add_edge(field_index, (row + 1) + (column - 2) * width, graph);
                }

                // [i + 1, j + 2]
                if(row + 1 < width && column + 2 < width && can_occupy.at((row + 1) + (column + 2) * width)) {
                    add_edge(field_index, (row + 1) + (column + 2) * width, graph);
                }

                // [i - 2, j - 1]
                if(row - 2 >= 0 && column - 1 >= 0 && can_occupy.at((row - 2) + (column - 1) * width)) {
                    add_edge(field_index, (row - 2) + (column - 1) * width, graph);
                }

                // [i - 2, j + 1]
                if(row - 2 >= 0 && column + 1 < width && can_occupy.at((row - 2) + (column + 1) * width)) {
                    add_edge(field_index, (row - 2) + (column + 1) * width, graph);
                }

                // [i + 2, j - 1]
                if(row + 2 < width && column - 1 >= 0 && can_occupy.at((row + 2) + (column - 1) * width)) {
                    add_edge(field_index, (row + 2) + (column - 1) * width, graph);
                }

                // [i + 2, j + 1]
                if(row + 2 < width && column + 1 < width && can_occupy.at((row + 2) + (column + 1) * width)) {
                    add_edge(field_index, (row + 2) + (column + 1) * width, graph);
                }
            }
        }

        // max cardinality
        vector<Vertex> mate(width * width);
        checked_edmonds_maximum_cardinality_matching(graph, &mate[0]);
        int matches = matching_size(graph, &mate[0]);

        //cout << "-> " << matches << endl;

        cout << ((width * width) - hole_count) - matches << endl;

    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <map>
#include <string>

#include <boost/config.hpp>
```

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> Graph;
typedef graph_traits<Graph>::vertex_descriptor  Vertex;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  vector<pair<int, int> > moves;
  moves.reserve(8);
  moves.push_back(make_pair(-1, -2));
  moves.push_back(make_pair(-1, 2));
  moves.push_back(make_pair(1, -2));
  moves.push_back(make_pair(1, 2));
  moves.push_back(make_pair(-2, -1));
  moves.push_back(make_pair(-2, 1));
  moves.push_back(make_pair(2, -1));
  moves.push_back(make_pair(2, 1));

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int board_side_length;
    cin >> board_side_length;

    vector<vector<int> > board(board_side_length, vector<int>(board_side_length, -1));
    int vertex_id = 0;
    for (int i = 0; i < board_side_length; ++i) {
      for (int j = 0; j < board_side_length; ++j) {
        int field_status;
        cin >> field_status;
        if (field_status == 1) {
          //add_vertex(g);
          board[i][j] = vertex_id++;
        }
      }
    }
    int nr_vertices = vertex_id;
    Graph g(nr_vertices);

    for (int i = 0; i < board_side_length; ++i) {
      for (int j = 0; j < board_side_length; ++j) {
        if (board[i][j] != -1) {
          for (int k = 0; k < moves.size(); ++k) {
            int x_cord = i + moves[k].first;
            int y_cord = j + moves[k].second;
            if (x_cord >= 0 && x_cord < board_side_length && y_cord >= 0 && y_cord < board_side_length && board[x_cord][
                y_cord] != -1) {
              add_edge(board[i][j], board[x_cord][y_cord], g);
            }
          }
        }
      }
    }

    vector<Vertex> mate(nr_vertices);
    edmonds_maximum_cardinality_matching(g, &mate[0]);
    int m_size = matching_size(g, &mate[0]);
    cout << m_size << endl;
    cout << nr_vertices - m_size << endl;
  }

  return 0;
}
```

8.10 Beach Bar

Keywords: ACM, Scanline

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
#include <utility>
#include <set>
#include <limits>

using namespace std;

#define MAX_BAR_RANGE 200

int main() {
```

```
cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

int test_count;
cin >> test_count;

for(int test = 0; test < test_count; test++) {
  int parasols_count;
  cin >> parasols_count;

  // read in parasol locations
  vector<int> parasols_locs(parasols_count);
  for(int parasol_index = 0; parasol_index < parasols_count; parasol_index++) {
    cin >> parasols_locs.at(parasol_index);
  }

  // first sort parasols, so we can start from the one furthest away in negative direction
  sort(parasols_locs.begin(), parasols_locs.end());

  // keep track of the maximum amount of parasols
  int max_parasols = 0;
  int cur_parasol_count = 0;

  // keep track of `parasols_locs` indexes resulting in a ranges in which we can build a bar
  vector<pair<int, int> > bar_ranges;

  // keeps track of current start/stop index
  int start_index = 0;
  int stop_index = 0;

  bar_ranges.push_back(make_pair(start_index, stop_index));

  // iterate through parasols and collect bar ranges
  for(int parasol_index = 0; parasol_index < parasols_count; parasol_index++) {
    // if the current parasol is too far away from the previous one (defined by `start_index`), remove
    // parasols as long as there are more between the original `start_index` and current one until we're
    // back in the range of a bar
    for(; parasols_locs.at(parasol_index) - parasols_locs.at(start_index) > MAX_BAR_RANGE; start_index++) {
      cur_parasol_count--;
    }

    stop_index = parasol_index; // new "last" parasol in range of the bar
    cur_parasol_count++; // as we added the current parasol, increase the amount

    if(max_parasols < cur_parasol_count) {
      // we found a range with more parasols as before!
      max_parasols = cur_parasol_count;

      // as we found a better range, remove the previously found ones
      bar_ranges.clear();
      // ... and add the better one
      bar_ranges.push_back(make_pair(start_index, stop_index));
    } else if(max_parasols == cur_parasol_count) {
      // found another location for the bar with equal amount of reached parasols
      // we add it to our list of possible locations
      bar_ranges.push_back(make_pair(start_index, stop_index));
    }
  }

  // so, now we have a list of parisol ranges where we can build a bar in the middle and reach maximum
  // amount of customers. Next we have to find the one bar with the minimum distance.

  set<int> bar_locs;
  int min_distance_found = numeric_limits<int>::max();

  for(int range_index = 0; range_index < bar_ranges.size(); range_index++) {
    // stupid name, but basically `added_range / 2` is the location of the bar :-)
    int added_range = parasols_locs[bar_ranges.at(range_index).first] +
      parasols_locs[bar_ranges.at(range_index).second];

    int diff = (parasols_locs[bar_ranges.at(range_index).second] -
      parasols_locs[bar_ranges.at(range_index).first] + 1) / 2; // +1 because of rounding

    if(diff > min_distance_found) {
      // we found something with a better minimal distance, so ignore this bar!
      continue;
    } else if(diff < min_distance_found) {
      // oh, we found a better minimum, replace everything
      min_distance_found = diff;
      bar_locs.clear();
    }

    // now add the current bar to our bar location collection, as we know it is the best one
    // or as good as the best we not till now
    if(added_range % 2 == 0) {
      // even distance
      bar_locs.insert(added_range / 2);
    } else {
      bar_locs.insert((added_range - 1) / 2);
      bar_locs.insert((added_range - 1) / 2 + 1);
    }
  }

  // output basic information
  cout << max_parasols << " " << min_distance_found << endl;
```

```
    // output bar location in order: it is, we use a set!

    set<int>::iterator loc_iter = bar_locs.begin();
    cout << *loc_iter++;
    for(; loc_iter != bar_locs.end(); ++loc_iter) {
      cout << " " << *loc_iter;
    }

    cout << endl;
  }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <stack>

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_parasols;
    cin >> nr_parasols;
    vector<int> parasols;
    parasols.reserve(nr_parasols);
    for (int i = 0; i < nr_parasols; i++) {
      int location;
      cin >> location;
      parasols.push_back(location);
    }
    sort(parasols.begin(), parasols.end());

    queue<int> covered_parasols;
    int best_nr_covered = 0;
    int best_min_distance = 0;
    set<int> best_locations;

    for (int i = 0; i < nr_parasols; i++) {
      int cur = parasols[i];
      covered_parasols.push(cur);
      while (covered_parasols.front() < cur - 200) {
        covered_parasols.pop();
      }
      int min_distance = ceil((cur - covered_parasols.front()) / 2.0);
      if (covered_parasols.size() > best_nr_covered) {
        best_nr_covered = covered_parasols.size();
        best_min_distance = min_distance;
        best_locations.clear();
        //add location
        int added = cur + covered_parasols.front();
        if (added % 2 == 0)
          best_locations.insert(added / 2);
        else {
          best_locations.insert((added - 1) / 2);
          best_locations.insert((added - 1) / 2 + 1);
        }
      }
      else if (covered_parasols.size() == best_nr_covered) {
        if (min_distance < best_min_distance) {
          best_min_distance = min_distance;
          best_locations.clear();
          //add location
          int added = cur + covered_parasols.front();
          if (added % 2 == 0)
            best_locations.insert(added / 2);
          else {
            best_locations.insert((added - 1) / 2);
            best_locations.insert((added - 1) / 2 + 1);
          }
        }
        else if (min_distance == best_min_distance) {
          //add location
          int added = cur + covered_parasols.front();
          if (added % 2 == 0)
            best_locations.insert(added / 2);
          else {
            best_locations.insert((added - 1) / 2);
            best_locations.insert((added - 1) / 2 + 1);
          }
        }
      }
    }
```

```
    cout << best_nr_covered << " " << best_min_distance << endl;
    set<int>::iterator it = best_locations.begin();
    cout << *it;
    for (++it; it != best_locations.end(); ++it) {
      cout << " " << *it;
    }
    cout << endl
  }

  return 0;
}
```

8.11 Light the Stage

Keywords: CGAL, Point set, Triangulation with info()

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Point_set_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef K::Point_2 Point;
typedef K::Circle_2 Circle;

typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Point_set_2<K, Tds> PSet;
typedef CGAL::Point_set_2<K, Tds>::Vertex_handle Vertex_handle;

using namespace std;

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;

  for(int test = 0; test < test_count; test++) {
    int participant_count, lamp_count;
    cin >> participant_count >> lamp_count;

    // read in participants
    vector<K::Point_2> participant_locations(participant_count);
    vector<int> participant_radi(participant_count, -1);
    for(int participant_index = 0; participant_index < participant_count; participant_index++) {
      double p_x, p_y, p_r;
      cin >> p_x >> p_y >> p_r;

      participant_locations.at(participant_index) = K::Point_2(p_x, p_y);
      participant_radi.at(participant_index) = p_r;
    }

    // read in lamp information
    int lamp_height;
    cin >> lamp_height;

    // make it a vector of a pair. The moment we feed it in to Point_set_2 we can access the second
    // member of the pair by accessing ->info() :-)
    vector<pair<K::Point_2, int> > lamp_locations(lamp_count);
    for(int lamp_index = 0; lamp_index < lamp_count; lamp_index++) {
      double l_x, l_y;
      cin >> l_x >> l_y;

      lamp_locations.at(lamp_index) = make_pair(K::Point_2(l_x, l_y), lamp_index);
    }

    // create a triangulation of the lamps using Point_set_2
    PSet lamps_triang(lamp_locations.begin(), lamp_locations.end());

    // contains the ID of the light that hit a participant first, or MAX if he survived till the end
    vector<int> min_light_hit(participant_count, numeric_limits<int>::max());

    // keeps track of the max light that hit someone
    int max_light_hit = -1;

    // now we iterate over each person and find out which lamp hits each person first, i.e. how
    // long they survive
    for(int participant_index = 0; participant_index < participant_count; participant_index++) {
      K::Point_2 participant_point = participant_locations.at(participant_index); // we need it more than once :-)
      // get nearest lamp
      Vertex_handle lamp_vertex = lamps_triang.nearest_neighbor(participant_point);

      /* Geometry lesson:
       * We know that the light is a cone with given height and it's a 90 degree one:
       *
       *                x <---- lamp point
       *               /|\
```

```
       *               / | \
       *              g  h  g
       *             /   |   \
       *            ----------
       *                 ^- 'h' and the bottom have a 90 degree, therefore the triangle on the left/right (h, g and half
           of the bottom)
       * have a 90 degree and two 45 degrees, i.e. we can find out what the length of the bottom is (twice the bottom
           part of the triangle)
       * which is in this case 2 * 'h' (bottom is of 2 * 'h' length).
       */

      // calculate the distance from our participant that still could be hit by light to "kill" the
      // participant.
      // do it ^2 (squared) because we compare it to a squared distance
      double max_dist_to_kill = pow(participant_radi.at(participant_index) + lamp_height, 2);

      // if the nearest lamp is already too far away, we don't have to search for other lamps,
      // as they will even farther away
      if(max_dist_to_kill <= squared_distance(lamp_vertex->point(), participant_point)) {
        continue;
      }

      // iterate over all lamps in the search for one that hits the participant first
      for(int check_lamp_index = 0; check_lamp_index < lamp_count; check_lamp_index++) {
        if(max_dist_to_kill > squared_distance(lamp_locations.at(check_lamp_index).first, participant_point)) {
          // found such a lamp, stop searching as we only interested in the minimum!
          min_light_hit.at(participant_index) = lamp_locations.at(check_lamp_index).second;
          max_light_hit = max(max_light_hit, lamp_locations.at(check_lamp_index).second);
          break;
        }
      }

      /* nice idea, but too slow...

      // create a circle in which every light source would kill the current participant
      K::Circle_2 circle_of_death = K::Circle_2(participant_point, max_dist_to_kill);

      // create a vector of vertex handles that represent our lights that are inside the circle_of_death
      vector<Vertex_handle> lights_hitting_participant;

      // so, micro optimizing stuff: don't do the rest of the algorithm if the lamp ID is higher than the
      // already found one
      if(min_light_hit.at(participant_index) < lamp_vertex->info()) {
        continue;
      }

      // collect the lights "of death" :-)
      lamps_triang.range_search(circle_of_death, back_inserter(lights_hitting_participant));

      // iterate over the bad lights and write down the smallest light hitting
      for(vector<Vertex_handle>::iterator iter = lights_hitting_participant.begin();
        iter != lights_hitting_participant.end();
        ++iter) {
        int lamp_id = (*iter)->info();

        if(circle_of_death.has_on_boundary((*iter)->point())) {
          // point is on boundary, this is not a hit according to the exercise
          continue;
        }

        min_light_hit.at(participant_index) = min(min_light_hit.at(participant_index), lamp_id);
        max_light_hit = max(max_light_hit, lamp_id);
      }
      */
    }

    // now we have to extract for each light who survived the longest
    vector<int> rank_list;

    // first we search for participants which were not hit by a light
    for(int participant_index = 0; participant_index < participant_count; participant_index++) {
      if(min_light_hit.at(participant_index) == numeric_limits<int>::max()) {
        rank_list.push_back(participant_index);
      }
    }

    // if all participants were hit, we have to search for the ones that were hit last
    if(rank_list.size() <= 0) {
      for(int participant_index = 0; participant_index < participant_count; participant_index++) {
        if(min_light_hit.at(participant_index) == max_light_hit) {
          rank_list.push_back(participant_index);
        }
      }
    }

    // print out rank in sorted order
    sort(rank_list.begin(), rank_list.end());
    for(int i = 0; i < rank_list.size(); i++) {
      cout << rank_list.at(i) << " ";
    }
    cout << endl;
  }
}
```

```
#include <iostream>
```

```cpp
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>


#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Point_set_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef K::Point_2 Point;
typedef K::Circle_2 Circle;

typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Point_set_2<K, Tds> PSet;
typedef CGAL::Point_set_2<K, Tds>::Vertex_handle Vertex_handle;

using namespace std;

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

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

class CompareVertex {
public:
  bool operator()(const Vertex_handle& lhs, const Vertex_handle& rhs) {
    return lhs->info() < rhs->info();
  }
};

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_players, nr_lights;
    cin >> nr_players >> nr_lights;

    vector<Point> players;
    vector<K::FT> players_r;
    players.reserve(nr_players);
    players_r.reserve(nr_players);
    for (int i = 0; i < nr_players; i++) {
      int player_x, player_y, player_r;
      cin >> player_x >> player_y >> player_r;
      players.push_back(Point(player_x, player_y));
      players_r.push_back(player_r);
    }

    K::FT light_r;
    cin >> light_r;
    vector<pair<Point, int> > lights;
    lights.reserve(nr_lights);
    for (int i = 0; i < nr_lights; i++) {
      int light_x, light_y;
      cin >> light_x >> light_y;
      lights.push_back(make_pair(Point(light_x, light_y), i));
    }

    PSet pset(lights.begin(), lights.end());
    vector<int> min_light_hit(nr_players, numeric_limits<int>::max());
    for (int i = 0; i < nr_players; i++) {
      Point player = players[i];
      Vertex_handle light = pset.nearest_neighbor(player);
      double max_dist = pow(players_r[i] + light_r, 2);

      if (max_dist <= squared_distance(light->point(), player)) {
        continue;
      }

      for (int j = 0; j < nr_lights; j++) {
```

```
        if (max_dist > squared_distance(lights[j].first, player)) {
          min_light_hit[i] = lights[j].second;
          break;
        }
      }
    }


    int max_light_hit = *max_element(min_light_hit.begin(), min_light_hit.end());
    for (int i = 0; i < nr_players; i++) {
      if (min_light_hit[i] == max_light_hit) {
        cout << i << " ";
      }
    }
    cout << endl;

    //Binary search would propably work...

    //Too slow
    /*PSet pset(lights.begin(), lights.end());
    vector<int> player_death_time(nr_players, -1);
    for (int i = 0; i < nr_players; i++) {
    K::FT circle_squared_dist = pow(players_r[i] + light_r, 2);
    Circle c(players[i], circle_squared_dist);
    vector<Vertex_handle> result;
    pset.range_search(c, back_inserter(result));
    //cout << "player: " << i << " found: " << result.size() << endl;
    sort(result.begin(), result.end(), CompareVertex());
    for (int j = 0; j < result.size(); ++j) {
    int death_time = result[j]->info();
    if (player_death_time[i] == -1 || death_time < player_death_time[i]) {
    if (squared_distance(result[j]->point(), players[i]) < circle_squared_dist) {
    player_death_time[i] = death_time;
    break;
    }
    }
    }
    }

    int death_max = 0;
    for (int i = 0; i < nr_players; i++) {
    if (player_death_time[i] > death_max) {
    death_max = player_death_time[i];
    }
    if (player_death_time[i] == -1) {
    death_max = -1;
    break;
    }
    }
    for (int i = 0; i < nr_players; i++) {
    if (player_death_time[i] == death_max) {
    cout << i << " ";
    }
    }
    cout << endl;*/
  }

  return 0;
}
```

8.12 Search Snippets

Keywords: ACM, Scanline, Custom compare, Compare in class

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <bitset>
#include <set>

using namespace std;

class PairCompare
{
public:
    bool operator() (pair<int, int>& lhs, pair<int, int>& rhs) {
        return lhs.first > rhs.first;
    }
};

int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
      int word_count;
        cin >> word_count;

        // read in how often each word occurs
```

```
        vector<int> occurs(word_count, 0);
        for(int word_index = 0; word_index < word_count; word_index++) {
            cin >> occurs.at(word_index);
        }

        // read in where it occurs
        priority_queue<pair<int, int>, vector<pair<int, int> >, PairCompare> locs;
        for(int word_index = 0; word_index < word_count; word_index++) {
            for(int i = 0; i < occurs.at(word_index); i++) {
                int l;
                cin >> l;
                locs.push(make_pair(l, word_index));
            }
        }

        // find first range with all words
        vector<int> first_set_loc(word_count, -1);
        set<pair<int, int> > words;
        int words_found = 0;
        int last_loc = numeric_limits<int>::min();
        while(!locs.empty() && words_found != word_count) {
            pair<int, int> cur = locs.top();
            locs.pop();

            if(first_set_loc.at(cur.second) == -1) {
                words_found++;
            } else {
                words.erase(make_pair(first_set_loc.at(cur.second), cur.second));
            }

            first_set_loc.at(cur.second) = cur.first;
            words.insert(cur);
            last_loc = cur.first;
        }

        //cout << "words_found: " << words_found << endl;
        //for(int x = 0; x < word_count; x++) {
            //cout << "\tword " << x << ", loc: " << first_set_loc.at(x) << endl;
            //locs.push(make_pair(first_set_loc.at(x), x));
        //}
        //cout << "min: " << words.begin()->first << ", max: " << last_loc << endl;

        int min_dist = last_loc - words.begin()->first;

        while(!locs.empty()) {
            pair<int, int> cur = locs.top();
            locs.pop();

            // modify location
            words.erase(make_pair(first_set_loc.at(cur.second), cur.second));
            words.insert(cur);
            first_set_loc.at(cur.second) = cur.first;

            // check range
            int d = cur.first - words.begin()->first;
            min_dist = min(min_dist, d);
        }

        cout << min_dist + 1 << endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#define MOD_NUM 100003

using namespace std;

struct PairComparator {
  bool operator() (const pair<int, int>& lhs, const pair<int, int>& rhs) {
    return lhs.first > rhs.first;
  }
};


int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
```

```
    int nr_words;
    cin >> nr_words;

    vector<int> nr_word_occurrences;
    nr_word_occurrences.reserve(nr_words);
    for (int i = 0; i < nr_words; ++i) {
      int nr_word_occurrence;
      cin >> nr_word_occurrence;
      nr_word_occurrences.push_back(nr_word_occurrence);
    }

    priority_queue<pair<int, int>, vector<pair<int, int> >, PairComparator> words;
    for (int i = 0; i < nr_words; ++i) {
      for (int j = 0; j < nr_word_occurrences[i]; ++j) {
        int word_occurrence;
        cin >> word_occurrence;
        words.push(make_pair(word_occurrence, i));
      }
    }

    //find first valid range
    set<pair<int, int> > max_word_occurrences_sorted;
    vector<int> max_word_occurences(nr_words, -1);
    int found_words = 0;
    int bound_max = 0;
    while (found_words < nr_words) {
      pair<int, int> word = words.top();
      words.pop();

      if (max_word_occurences[word.second] == -1) {
        //new word
        found_words++;
      }
      else {
        //already inserted word
        max_word_occurrences_sorted.erase(make_pair(max_word_occurences[word.second], word.second));
      }
      max_word_occurences[word.second] = word.first;
      max_word_occurrences_sorted.insert(word);
      bound_max = word.first;
    }

    int min_range = bound_max - max_word_occurrences_sorted.begin()->first;
    while (!words.empty()) {
      pair<int, int> word = words.top();
      words.pop();

      max_word_occurrences_sorted.erase(make_pair(max_word_occurences[word.second], word.second));
      max_word_occurences[word.second] = word.first;
      max_word_occurrences_sorted.insert(word);
      bound_max = word.first;

      min_range = min(min_range, bound_max - max_word_occurrences_sorted.begin()->first);
    }

    cout << min_range+1 << endl;
  }

  return 0;
}
```

8.13 Radiation Therapy

Keywords: CGAL, Quadratic Program, Exponential bound search, Binary search

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

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

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}
```

```
int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a - 1 >= x) a -= 1;
  return a;
}

bool can_solve(int dimension, vector<tuple<int, int, int> >& healthy, vector<tuple<int, int, int> >& cancer) {
  // use EQUAL, as we set it manually if we wish => or <=
  Program qp(CGAL::EQUAL, false, 0, false, 0);

  // add healthy cell inequations
  for(int cell = 0; cell < healthy.size(); cell++) {
    int var_index = 0;
    for(int x_dim = 0; x_dim <= dimension; x_dim++) {
      for(int y_dim = 0; y_dim <= dimension; y_dim++) {
        for(int z_dim = 0; z_dim <= dimension; z_dim++) {
          if(x_dim + y_dim + z_dim <= dimension) {
            double value = pow(get<0>(healthy.at(cell)), x_dim) *
              pow(get<1>(healthy.at(cell)), y_dim) *
              pow(get<2>(healthy.at(cell)), z_dim);

            qp.set_a(var_index, cell, value);
            //cout << "value H: " << value << endl;

            var_index++;
          } else {
            break;
          }
        }
      }
    }

    qp.set_r(cell, CGAL::LARGER);
    qp.set_b(cell, 1);
  }

  // add cancer cell inequations
  for(int cell = 0; cell < cancer.size(); cell++) {
    int var_index = 0;
    for(int x_dim = 0; x_dim <= dimension; x_dim++) {
      for(int y_dim = 0; y_dim <= dimension; y_dim++) {
        for(int z_dim = 0; z_dim <= dimension; z_dim++) {
          if(x_dim + y_dim + z_dim <= dimension) {
            double value = pow(get<0>(cancer.at(cell)), x_dim) *
              pow(get<1>(cancer.at(cell)), y_dim) *
              pow(get<2>(cancer.at(cell)), z_dim);

            qp.set_a(var_index, cell + healthy.size(), value);

            //cout << "value C: " << value << endl;

            var_index++;
          } else {
            break;
          }
        }
      }
    }

    qp.set_r(cell + healthy.size(), CGAL::SMALLER);
    qp.set_b(cell + healthy.size(), -1);
  }

  // solve
  CGAL::Quadratic_program_options options;
  options.set_pricing_strategy(CGAL::QP_BLAND);     // Bland's rule to avoid cycling...
  Solution s = CGAL::solve_linear_program(qp, ET(), options);

  //cout << s << endl;

  return !s.is_infeasible();
}

int main() {

  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;
  for(int test = 0; test < test_count; test++) {
    //cout << endl << endl;
    int healthy_count, cancer_count;
    cin >> healthy_count >> cancer_count;

    // read in healthy cell location
    vector<tuple<int, int, int> > healthy(healthy_count);
    for(int healthy_index = 0; healthy_index < healthy_count; healthy_index++) {
      int x, y, z;
      cin >> x >> y >> z;
```

```
      healthy.at(healthy_index) = make_tuple(x, y, z);
    }

    // read in cancer cell location
    vector<tuple<int, int, int> > cancer(cancer_count);
    for(int cancer_index = 0; cancer_index < cancer_count; cancer_index++) {
      int x, y, z;
      cin >> x >> y >> z;

      cancer.at(cancer_index) = make_tuple(x, y, z);
    }

    // search upper lower
    int exp_value = 1;
    bool exp_found = false;
    do {
      exp_value = exp_value * 2;
      exp_found = true;
      //cout << "inc: for: " << exp_value << " is: " << can_solve(exp_value, healthy, cancer) << endl;
    } while(exp_value <= 30 && !can_solve(exp_value, healthy, cancer));

    // do binary search for the right dimension
    int min_d = 0;
    int max_d = 30;
    if(exp_found) {
      min_d = exp_value / 2 - 1;
      max_d = exp_value > 30 ? 30 : exp_value;
    }

    //cout << "min set to " << min_d << " and max to " << max_d << endl;

    bool last_worked = false;
    while(min_d < max_d) {
      int mid_d = (min_d + max_d) / 2;
      //cout << "binary search from: " << min_d << " to " << max_d << ", checking: " << mid_d << endl;
      if(can_solve(mid_d, healthy, cancer)) {
        // [min_d, mid_d] contains the solution
        max_d = mid_d;
        last_worked = true;
        //cout << "\tOK" << endl;
      } else {
        // [mid_d + 1, max_d] contains the solution
        min_d = mid_d + 1;
        last_worked = false;
        //cout << "\tBAD!" << endl;
      }
    }

    if(min_d == 30 && !last_worked) {
      cout << "Impossible!" << endl;
    } else {
      cout << min_d << endl;
    }
  }
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

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

using namespace std;

int floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a + 1 <= x) a += 1;
  return a;
}

int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a - 1 >= x) a -= 1;
```

```
  return a;
}

bool check_degree(vector<tuple<int, int, int> > cells, int nr_healthy, int degree) {
  //cout << "check with: " << degree << " ";
  Program qp(CGAL::LARGER, false, 0, false, 0);
  for (int i = 0; i < cells.size(); i++) {
    int id = 0;
    tuple<int, int, int>& cell = cells[i];
    for (int x = 0; x <= degree; x++) {
      for (int y = 0; y <= degree; y++) {
        for (int z = 0; z <= degree; z++) {
          if (x + y + z <= degree) {
            qp.set_a(id, i, pow(get<0>(cell), x)*pow(get<1>(cell), y)*pow(get<2>(cell), z));
            id++;
          }
          else {
            break;
          }
        }
      }
    }
    if (i < nr_healthy) {
      qp.set_b(i, 1);
      qp.set_r(i, CGAL::LARGER);
    }
    else {
      qp.set_b(i, -1);
      qp.set_r(i, CGAL::SMALLER);
    }
  }
  CGAL::Quadratic_program_options options;
  options.set_pricing_strategy(CGAL::QP_BLAND); // Bland's rule to avoid cycling...
  Solution s = CGAL::solve_linear_program(qp, ET(), options);
  //cout << (s.is_optimal() == 1 ? "true" : "false") << endl;
  return !s.is_infeasible();
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_healthy, nr_tumor;
    cin >> nr_healthy >> nr_tumor;
    int nr_cells = nr_healthy + nr_tumor;

    vector<tuple<int, int, int> > cells;
    cells.reserve(nr_cells);
    for (int i = 0; i < nr_cells; i++) {
      int cell_x, cell_y, cell_z;
      cin >> cell_x >> cell_y >> cell_z;
      cells.push_back(make_tuple(cell_x, cell_y, cell_z));
    }

    int exp_value = 0;
    while (exp_value < 30 && !check_degree(cells, nr_healthy, exp_value)) {
      exp_value = exp_value == 0 ? 1 : exp_value * 2;
    }

    int min_bound = exp_value == 0 ? 0 : exp_value / 2 + 1;
    int max_bound = exp_value > 30 ? 30 : exp_value;

    //int min_bound = 0;
    //int max_bound = 30;

    bool last_check = false;
    while (min_bound < max_bound) {
      int test_degree = (min_bound + max_bound) / 2;
      //int test_degree = (min_bound*3 + max_bound) / 4;
      if (check_degree(cells, nr_healthy, test_degree)) {
        max_bound = test_degree;
        last_check = true;
      }
      else {
        min_bound = test_degree + 1;
        last_check = false;
      }
    }

    if (min_bound == 30 && !last_check) {
      cout << "Impossible!" << endl;
    }
    else {
      cout << min_bound << endl;
    }
  }

  return 0;
}
```

8.14 Island Hopping

Keywords: ACM, Dynamic Programming

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

using namespace std;

int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        int island_count, attack_radius, attacker_strength_start;
        cin >> island_count >> attack_radius >> attacker_strength_start;

        // read in the defense strength
        vector<int> defense(island_count + 1, 0);
        for(int island_index = 1; island_index < island_count; island_index++) {
            cin >> defense.at(island_index);
        }

        assert(defense.at(0) == 0);
        assert(defense.at(island_count) == 0);

        // keeps track of the path length
        vector<int> path_length(island_count + 1, numeric_limits<int>::max()); // first island is the attacker's island,
             this is just always -inf
        path_length.at(0) = 0;

        // our DP table containing the resulting fighter strength left at an island
        vector<int> table(island_count + 1, 0);
        table.at(0) = attacker_strength_start;

        // DP algo
        for(int cur_island = 0; cur_island < island_count; cur_island++) {
            if(table.at(cur_island) <= 0) {
                continue;
            }

            for(int next_island = cur_island + 1;
                next_island <= cur_island + attack_radius && next_island <= island_count;
                next_island++) {

                // calculate how many attackers are left if we send the people from island `cur_island` to
                // island `next_island`
                int attackers_left = table.at(cur_island) - defense.at(next_island);

                // if this is a greater number than the one already in our DP table, we found a better way
                if(attackers_left > table.at(next_island)) {
                    table.at(next_island) = attackers_left;
                    // don't forget to keep track of the path
                    path_length.at(next_island) = path_length.at(cur_island) + 1;
                } else if(attackers_left == table.at(next_island)) {
                    // maybe we found an other, better path with the same results
                    path_length.at(next_island) = min(path_length.at(cur_island) + 1, path_length.at(next_island));
                }
            }
        }

        if(table.at(island_count) <= 0) {
            cout << "safe" << endl;
        } else {
            cout << path_length.at(island_count) - 1 << " " << table.at(island_count) << endl;
        }
    }

  return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <stack>

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
```

```
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_islands, attack_radius, initial_strength;
    cin >> nr_islands >> attack_radius >> initial_strength;

    vector<int> defence_strengths;
    defence_strengths.reserve(nr_islands + 1);
    defence_strengths.push_back(0);
    for (int i = 1; i < nr_islands; i++) {
      int defence_strength;
      cin >> defence_strength;
      defence_strengths.push_back(defence_strength);
    }
    defence_strengths.push_back(0);

    vector<int> max_attack_forces(nr_islands + 1, 0);
    vector<int> min_islands(nr_islands + 1, 0);
    max_attack_forces[0] = initial_strength;
    for (int i = 0; i < nr_islands; i++) {
      int current_attack_force = max_attack_forces[i];
      if (current_attack_force > 0) {
        for (int j = 1; j <= attack_radius; j++) {
          if (i + j <= nr_islands) {
            int new_attack_force = current_attack_force - defence_strengths[i + j];
            if (new_attack_force > max_attack_forces[i + j]) {

              max_attack_forces[i + j] = new_attack_force;
              min_islands[i + j] = min_islands[i] + 1;
            }
            else if (new_attack_force == max_attack_forces[i + j]) {
              min_islands[i + j] = min(min_islands[i + j], min_islands[i] + 1);
            }
          }
          else {
            break;
          }
        }
      }
    }

    if (max_attack_forces[nr_islands] > 0) {
      cout << min_islands[nr_islands] - 1 << " " << max_attack_forces[nr_islands] << endl;
    }
    else {
      cout << "safe" << endl;
    }
  }

  return 0;
}
```

8.15 Sweepers

Keywords: BGL, Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow, Euler circuit, Components

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

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

using namespace std;
using namespace boost;

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;
typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> SimpleGraph;
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 add_edge, also creates reverse edges with corresponding capacities.
void addEdge(int u, int v, long c, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, Graph &G) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
```

```
}

int main() {
  cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
      int room_count, corridor_count, sweeper_count;
        cin >> room_count >> corridor_count >> sweeper_count;

        int door_count = sweeper_count;

        // create graph for flow analysis
        Graph flow_graph(room_count + 2);
        EdgeCapacityMap capacity = get(edge_capacity, flow_graph);
        ReverseEdgeMap  rev_edge = get(edge_reverse, flow_graph);
        //ResidualCapacityMap res_capacity = get(edge_residual_capacity, flow_graph);

        // create graph with just the rooms, corridors
        SimpleGraph castle_graph(room_count);

        // define source and sink
        int source = room_count;
        int sink = room_count + 1;

        // keep track of vertices degrees
        vector<int> degrees(room_count, 0);
        vector<int> non_modified_degrees(room_count, 0);

        // read in location of sweepers
        vector<int> sweepers(room_count, 0);
        for(int sweeper_index = 0; sweeper_index < sweeper_count; sweeper_index++) {
            int location;
            cin >> location;

            // source -> sweeper location
            addEdge(source, location, 1, capacity, rev_edge, flow_graph);

            degrees.at(location)++;
            sweepers.at(location)++;
        }

        // read in location of doors
        for(int door_index = 0; door_index < door_count; door_index++) {
            int location;
            cin >> location;

            // door location -> sink
            addEdge(location, sink, 1, capacity, rev_edge, flow_graph);

            degrees.at(location)--;
        }

        // read in corridors
        for(int corridor_index = 0; corridor_index < corridor_count; corridor_index++) {
            int from, to;
            cin >> from >> to;

            addEdge(from, to, 1, capacity, rev_edge, flow_graph);
            addEdge(to, from, 1, capacity,rev_edge, flow_graph);

            add_edge(from, to, castle_graph);

            degrees.at(from)++;
            degrees.at(to)++;
            non_modified_degrees.at(from)++;
            non_modified_degrees.at(to)++;
        }

        // so now we have a basic graph, check if all sweepers can reach a door.
        // do this by doing a flow and check if flow result same as sweepers count
        int sweepers_exited = push_relabel_max_flow(flow_graph, source, sink);

        //cout << "sweepers exited: " << sweepers_exited << ", total sweepers: " << sweeper_count << endl;
        if(sweepers_exited != sweeper_count) {
            cout << "no" << endl;
            continue;
        }

        // check for euler circuit
        bool has_even_degree = true;
        for(int vertex = 0; vertex < room_count; vertex++) {
            if(degrees.at(vertex) < 0 || degrees.at(vertex) % 2 != 0) {
                has_even_degree = false;
                break;
            }
        }

        //cout << "even: " << has_even_degree << endl;
        if(!has_even_degree) {
            cout << "no" << endl;
            continue;
        }
```

```
        // make sure every edge, i.e. corridor, was visited at least once
        // first get the connected component information
        vector<int> component(room_count);
        connected_components(castle_graph, &component[0]);
        vector<int> cleaned_by_sweepers(room_count, 0);
        int max_component_nr = 0;
        for(int room_index = 0; room_index < room_count; room_index++) {
            // we're interested that every component is cleaned, so we use this as the index. Keep track of
            // how many components we have
            int component_nr = component.at(room_index);
            max_component_nr = max(max_component_nr, component_nr);

            // if the room has no corridors attached, we assume it is cleaned (only corridors need cleaning,
            // no corridor, no cleaning needed).
            if(non_modified_degrees.at(room_index) == 0) {
                cleaned_by_sweepers.at(component_nr) = 1;
            } else {
                // the degree of the room might be 0, then there is no sweeper added to the component. If never one
                // is added, component not cleaned.
                cleaned_by_sweepers.at(component_nr) += sweepers.at(room_index);
            }
        }

        // make sure every component is cleaned, i.e. contains some sweepers. Other requirements make sure
        // the component is cleaned if it contains sweepers
        bool components_cleaned = true;
        for(int i = 0; i <= max_component_nr; i++) {
            if(cleaned_by_sweepers.at(i) == 0) {
                components_cleaned = false;
                break;
            }
        }

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

    return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <stack>

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

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> SimpleGraph;

inline void add_flow_edge(int start, int end, int c, EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge, Graph& g) {
  Edge e, rev_e;
  bool success;
  tie(e, success) = add_edge(start, end, g);
  if (success) {
    tie(rev_e, success) = add_edge(end, start, g);
    capacity[e] = c;
    capacity[rev_e] = 0;
    rev_edge[e] = rev_e;
    rev_edge[rev_e] = e;
  }
  else {
    capacity[e] += c;
  }
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
```

```
int nr_test_cases;
cin >> nr_test_cases;

for (int test_case = 0; test_case < nr_test_cases; test_case++) {
  int nr_rooms, nr_corridors, nr_sweepers;
  cin >> nr_rooms >> nr_corridors >> nr_sweepers;

  //cout << endl;

  Graph g(nr_rooms + 2);
  SimpleGraph components_g(nr_rooms);
  int source = nr_rooms;
  int sink = nr_rooms + 1;

  EdgeCapacityMap capacity = get(edge_capacity, g);
  ReverseEdgeMap rev_edge = get(edge_reverse, g);

  vector<int> starting_locations(nr_rooms, 0);
  for (int i = 0; i < nr_sweepers; i++) {
    int starting_location;
    cin >> starting_location;

    add_flow_edge(source, starting_location, 1, capacity, rev_edge, g);

    starting_locations[starting_location]++;
  }

  vector<int> outside_doors(nr_rooms, 0);
  for (int i = 0; i < nr_sweepers; i++) {
    int outside_door;
    cin >> outside_door;

    add_flow_edge(outside_door, sink, 1, capacity, rev_edge, g);

    outside_doors[outside_door]++;
  }

  vector<int> vertex_degrees(nr_rooms, 0);
  for (int i = 0; i < nr_corridors; i++) {
    int end_one, end_two;
    cin >> end_one >> end_two;

    vertex_degrees[end_one]++;
    vertex_degrees[end_two]++;

    add_flow_edge(end_one, end_two, 1, capacity, rev_edge, g);
    add_flow_edge(end_two, end_one, 1, capacity, rev_edge, g);

    add_edge(end_one, end_two, components_g);
  }

  //check if all vertices have an even degree
  bool found_uneven = false;
  for (int i = 0; i < nr_rooms; i++) {
    if ((vertex_degrees[i] + outside_doors[i] + starting_locations[i]) % 2 == 1) {
      found_uneven = true;
      break;
    }
  }
  if (found_uneven) {
    //cout << "uneven degree found" << endl;
    cout << "no" << endl;
    continue;
  }

  //check if every component of the graph has at least one assigned sweeper
  vector<int> comp(nr_rooms);
  int num = connected_components(components_g, &comp[0]);
  vector<int> cleaned(num, 0);
  for (int i = 0; i < nr_rooms; i++) {
    if (vertex_degrees[i] == 0) {
      cleaned[comp[i]] = 1;
    }
    else {
      cleaned[comp[i]] += starting_locations[i];
    }
  }
  bool uncleaned_comp = false;
  for (int i = 0; i < num; i++) {
    if (cleaned[i] == 0) {
      uncleaned_comp = true;
      break;
    }
  }

  //cout << "no_sweeper: " << no_sweeper << endl;
  if (uncleaned_comp) {
    //cout << "component without sweeper" << endl;
    cout << "no" << endl;
    continue;
  }

  //check if all sweepers can flee
  int escaped_sweepers = push_relabel_max_flow(g, source, sink);
  if (escaped_sweepers < nr_sweepers) {
    //cout << "max flow to low" << endl;
```

```
      cout << "no" << endl;
      continue;
    }

    cout << "yes" << endl;
  }

  return 0;
}
```

8.16 Clues

Keywords: BGL, CGAL, BGL with CGAL, CGAL with BGL, Point set, Triangulation with info(), Bipartite, Components

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/bipartite.hpp>
#include <boost/graph/connected_components.hpp>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Point_set_2.h>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator Edge_iterator;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Circle_2 Circle;

typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Point_set_2<K, Tds> PSet;
typedef CGAL::Point_set_2<K, Tds>::Vertex_handle Vertex_handle;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;

  for(int test = 0; test < test_count; test++) {
    int station_count, clue_count;
    K::FT range;
    cin >> station_count >> clue_count >> range;

    range = pow(range, 2); // work with squared distances

    // read in station location
    // pair.first: the point where the station is located
    // pair.second: the index of the station
    vector<pair<Point, int> > stations(station_count);
    for(int station_index = 0; station_index < station_count; station_index++) {
      int x, y;
      cin >> x >> y;

      stations.at(station_index) = make_pair(Point(x, y), station_index);
    }

    // read in clue location
    vector<pair<Point, Point> > clues(clue_count);
    for(int clue_index = 0; clue_index < clue_count; clue_index++) {
      // read in from where to where we would like to go
      int start_x, start_y, target_x, target_y;
      cin >> start_x >> start_y >> target_x >> target_y;

      clues.at(clue_index) = make_pair(Point(start_x, start_y), Point(target_x, target_y));
    }

    // create a graph that can be used to check for graph properties
    Graph graph(station_count);

    // calculate the point set so we can find the nearest one for a given point and all stations
```

```
    // inside a circle of given radius
    PSet station_triang;
    station_triang.insert(stations.begin(), stations.end());

    // go trough stations, search for stations that are inside the radius and process
    // them.
    for(int stationA_index = 0; stationA_index < station_count; stationA_index++) {
      Point stationA = stations.at(stationA_index).first;
      Circle stationA_area = Circle(stationA, range);

      // collect all stations inside the circle
      vector<Vertex_handle> stations_in_range;
      station_triang.range_search(stationA_area, back_inserter(stations_in_range));

      // go trough found stations and make sure that each pair is at least 'range' away from each other
      // as long as it's not our stationA one of the pair's stations
      for(int i = 0; i < stations_in_range.size(); i++) {
        for(int j = i + 1; j < stations_in_range.size(); j++) {
          int some_station_1 = stations_in_range.at(i)->info();
          int some_station_2 = stations_in_range.at(j)->info();

          if(some_station_1 != stationA_index && some_station_2 != stationA_index) {
            if(CGAL::squared_distance(stations_in_range.at(i)->point(), stations_in_range.at(j)->point()) <= range) {
              // another pair of stations in reachable station, conflict!
              //cout << "blub" << endl;
              goto not_bipartit;
            }
          }
        }
      }

      // add edges
      for(int i = 0; i < stations_in_range.size(); i++) {
        int to = stations_in_range.at(i)->info();
        if(to != stationA_index) {
          add_edge(stationA_index, to, graph);
        }
      }
    }

    // check if graph is not bipartit, if so, we can stop
    if(!is_bipartite(graph)) {
      //cout << "no blub" << endl;
      not_bipartit:
      for(int clue_index = 0; clue_index < clue_count; clue_index++) {
        cout << "n";
      }
      cout << endl;
    } else {
      // now we have to check which of the clues can be received
      // get connected components information, so we can find out if start and end of a clue is in the same
      // component and therefore reachable
      vector<int> in_component(station_count);
      connected_components(graph, &in_component[0]);

      for(int clue_index = 0; clue_index < clue_count; clue_index++) {
        // get where the clue starts and should end
        Point start = clues.at(clue_index).first;
        Point target = clues.at(clue_index).second;

        // check if start and target of the clue are near each other and communicate directly
        if(range >= CGAL::squared_distance(start, target)) {
          cout << "y";
          continue;
        }

        // get nearest vertex to the start and check if we can reach it
        Vertex_handle near_start_station = station_triang.nearest_vertex(start);
        if(range < CGAL::squared_distance(start, near_start_station->point())) {
          cout << "n"; // next station from start too far away
          continue;
        }

        // now the same for the end station/clue location
        Vertex_handle near_end_station = station_triang.nearest_vertex(target);
        if(range < CGAL::squared_distance(target, near_end_station->point())) {
          cout << "n";
          continue;
        }

        // last part is to check if both are in the same component
        if(in_component.at(near_start_station->info()) == in_component.at(near_end_station->info())) {
          cout << "y";
        } else {
          cout << "n";
        }
      }

      cout << endl;
    }
  }

  return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/bipartite.hpp>
#include <boost/graph/connected_components.hpp>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Point_set_2.h>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef K::Point_2 Point;
typedef K::Circle_2 Circle;

typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Point_set_2<K, Tds> PSet;
typedef CGAL::Point_set_2<K, Tds>::Vertex_handle Vertex_handle;

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

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

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_stations, nr_clues, max_radius;
    cin >> nr_stations >> nr_clues >> max_radius;

    vector<pair<Point, int> > stations;
    stations.reserve(nr_stations);
    for (int i = 0; i < nr_stations; ++i) {
      int station_x, station_y;
      cin >> station_x >> station_y;
      stations.push_back(make_pair(Point(station_x, station_y), i));
    }

    vector<pair<Point, Point> > clues;
    clues.reserve(nr_clues);
    for (int i = 0; i < nr_clues; ++i) {
      int from_x, from_y, to_x, to_y;
      cin >> from_x >> from_y >> to_x >> to_y;
      clues.push_back(make_pair(Point(from_x, from_y), Point(to_x, to_y)));
    }

    PSet pset(stations.begin(), stations.end());
    K::FT max_radius_squared = pow(max_radius, 2);
    Graph g(nr_stations);
    for (int i = 0; i < nr_stations; ++i) {
      Circle c(stations[i].first, max_radius_squared);
      vector<Vertex_handle> result;
      pset.range_search(c, back_inserter(result));

      for (int j = 0; j < result.size(); ++j) {
        for (int k = j + 1; k < result.size(); ++k) {
          if (result[j]->info() != i && result[k]->info() != i) {
            if (CGAL::squared_distance(result[j]->point(), result[k]->point()) <= max_radius_squared) {
              goto not_bipartite;
```

```
          }
        }
      }
    }
    for (int j = 0; j < result.size(); ++j) {
      int to = result[j]->info();
      if (to != i) {
        add_edge(i, to, g);
      }
    }
  }
  if (!is_bipartite(g)) {
    not_bipartite:
    for (int i = 0; i < nr_clues; i++) {
      cout << "n";
    }
    cout << endl;
  }
  else {
    vector<int> component(nr_stations);
    connected_components(g, &component[0]);

    for (int i = 0; i < nr_clues; i++) {
      if (CGAL::squared_distance(clues[i].first, clues[i].second) <= max_radius_squared) {
        cout << "y";
        continue;
      }
      Vertex_handle start_p = pset.nearest_vertex(clues[i].first);
      if (CGAL::squared_distance(start_p->point(), clues[i].first) > max_radius_squared) {
        cout << "n";
        continue;
      }
      Vertex_handle end_p = pset.nearest_vertex(clues[i].second);
      if (CGAL::squared_distance(end_p->point(), clues[i].second) > max_radius_squared) {
        cout << "n";
        continue;
      }
      int start = start_p->info();
      int end = end_p->info();
      if (component[start] == component[end]) {
        cout << "y";
      }
      else {
        cout << "n";
      }
    }
    cout << endl;
  }
 }

 return 0;
}
```

8.17 Radiation 2

Keywords: CGAL, Delaunay Triangulation, Point set, Triangulation with info()

```
//
// ONLY FIRST TWO TEST CASES!
//

#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/Point_set_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;

typedef K::Point_2 Point;
typedef K::Circle_2 Circle;

typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Point_set_2<K, Tds> PSet;
typedef CGAL::Point_set_2<K, Tds>::Vertex_handle PSVertex_handle;

typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef CGAL::Delaunay_triangulation_2<K>::Vertex_handle  TriangVertex_handle;

using namespace std;

// from slides, fun!
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

double ceil_to_double(const K::FT& x)
```

```
{
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

int main() {
  // some basic setup stuff
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);
  cout << fixed << setprecision(0);

  int test_count;
  cin >> test_count;
  for(int test = 0; test < test_count; test++) {
    int healthy_count, cancer_count;
    cin >> healthy_count >> cancer_count;

    // read in healthy and cancer cells
    vector<Point> cancer_cells;
    cancer_cells.reserve(cancer_count);

    vector<Point> healthy_cells;
    healthy_cells.reserve(healthy_count);

    for(int healthy_index = 0; healthy_index < healthy_count; healthy_index++) {
      double x, y;
      cin >> x >> y;

      healthy_cells.push_back(Point(x, y));
    }

    for(int cancer_index = 0; cancer_index < cancer_count; cancer_index++) {
      double x, y;
      cin >> x >> y;

      cancer_cells.push_back(Point(x, y));
    }

    // create a point set of cancer cells, this allows us to find all cells inside a circle quickly (I hope!)
    PSet cancer_triang(cancer_cells.begin(), cancer_cells.end());

    // create a triangulation of healthy cells to find quickly the nearest one to a cancer point
    Triangulation healthy_triang(healthy_cells.begin(), healthy_cells.end());

    // iterate over all cancer cells to search for the best radius
    unsigned long max_cells_killed = 0;
    for(int cancer_index = 0; cancer_index < cancer_count; cancer_index++) {
      // gett the point of the cancer cell
      Point cancer_cell = cancer_cells.at(cancer_index);

      // get the nearest healthy cell, defining by that the radius
      TriangVertex_handle next_healthy = healthy_triang.nearest_vertex(cancer_cell);

      // create the circle which will be radiated
      Circle radiation_region = Circle(cancer_cell, CGAL::squared_distance(cancer_cell, next_healthy->point()));

      // get the cancer cells that would be eliminated
      vector<PSVertex_handle> killed_cancer_cells;
      cancer_triang.range_search(radiation_region, back_inserter(killed_cancer_cells));

      max_cells_killed = max(max_cells_killed, killed_cancer_cells.size());
    }

    cout << max_cells_killed << endl;
  }
}
```

8.18 Knights

Keywords: BGL, Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow

```
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <algorithm>

#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
```

```cpp
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor Edge;

void addEdge(int from, int to, int weight, Graph& graph, EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge) {
  Edge edge, r_edge;
  tie(edge, tuples::ignore) = add_edge(from, to, graph);
  tie(r_edge, tuples::ignore) = add_edge(to, from, graph);
  capacity[edge] = weight;
  capacity[r_edge] = 0;
  rev_edge[edge] = r_edge;
  rev_edge[r_edge] = edge;
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int test_count;
  cin >> test_count;
  for(int test = 0; test < test_count; test++) {
    int width, height, knight_count;
    cin >> width >> height >> knight_count;

    // create graph
    Graph graph(2 * width * height + 2 + knight_count); // don't forget source and sink
    EdgeCapacityMap capacity = get(edge_capacity, graph);
    ReverseEdgeMap rev_edge = get(edge_reverse, graph);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, graph);

    /* Index
     * [0 .. knight_count - 1]: knights
     * [knight_count .. knight_count + width * height - 1]: intersections of set 1 with edges to set 2
     * [knight_count + width * height .. knight_count + 2 * width * height - 1]: intersections of set 2 with edges to set
         1
     */

    // source / sink
    int source = 2 * width * height + knight_count;
    int sink = source + 1;

    // read in knight position
    for(int knight_index = 0; knight_index < knight_count; knight_index++) {
      int x, y;
      cin >> x >> y;

      // source -> sink
      addEdge(source, knight_index, 1, graph, capacity, rev_edge);
      //cout << "source -> " << knight_index << endl;

      // knight -> starting position in first intersection set
      addEdge(knight_index, knight_count + x + y * width, 1, graph, capacity, rev_edge);
      //cout << knight_index << " -> " << knight_count + x + y * width << endl;
    }

    // add missing edges
    for(int edge_x = 0; edge_x < width; edge_x++) {
      for(int edge_y = 0; edge_y < height; edge_y++) {
        int intersection_offset = edge_x + edge_y * width;

        // edge from first intersection set to second one
        addEdge(knight_count + intersection_offset, knight_count + width * height + intersection_offset, 1, graph,
             capacity, rev_edge);
        //cout << "first set to second set: " <<  intersection_offset << " -> " << intersection_offset << endl;

        // edge from second intersection set to all in set one that can be reached by a path of length 1 in the cave
        if(edge_x - 1 >= 0) {
          addEdge(knight_count + width * height + intersection_offset, knight_count + edge_x - 1 + edge_y * width, 1,
               graph, capacity, rev_edge);
          //cout << "A second set to first set: " << intersection_offset << " -> " << edge_x - 1 + edge_y * width << endl
               ;
        }

        if(edge_x + 1 < width) {
          addEdge(knight_count + width * height + intersection_offset, knight_count + edge_x + 1 + edge_y * width, 1,
               graph, capacity, rev_edge);
          //cout << "B second set to first set: " << intersection_offset << " -> " << edge_x + 1 + edge_y * width << endl
               ;
        }

        if(edge_y - 1 >= 0) {
          addEdge(knight_count + width * height + intersection_offset, knight_count + edge_x + (edge_y - 1) * width, 1,
               graph, capacity, rev_edge);
          //cout << "C second set to first set: " << intersection_offset << " -> " << edge_x + (edge_y - 1) * width <<
               endl;
        }

        if(edge_y + 1 < height) {
          addEdge(knight_count + width * height + intersection_offset, knight_count + edge_x + (edge_y + 1) * width, 1,
               graph, capacity, rev_edge);
          //cout << "D second set to first set: " << intersection_offset << " -> " << edge_x + (edge_y + 1) * width <<
               endl;
        }

        // add sink edge to second intersection set
        if(edge_x == 0 || edge_x + 1 == width || edge_y == 0 || edge_y + 1 == height) {
          addEdge(knight_count + width * height + intersection_offset, sink, 1, graph, capacity, rev_edge);
```

```
          //cout << knight_count + intersection_offset << " -> sink" << endl;
        }
      }
    }

    // do max flow
    int max = push_relabel_max_flow(graph, source, sink);

    cout << max << endl;

  }

  return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/property_map/transform_value_property_map.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

inline void add_flow_edge(int start, int end, int c, EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge, Graph& g) {
  Edge e, rev_e;
  bool success;
  tie(e, success) = add_edge(start, end, g);
  if (success) {
    tie(rev_e, success) = add_edge(end, start, g);
    capacity[e] = c;
    capacity[rev_e] = 0;
    rev_edge[e] = rev_e;
    rev_edge[rev_e] = e;
  }
  else {
    capacity[e] += c;
  }
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int x_dim, y_dim, nr_knights;
    cin >> x_dim >> y_dim >> nr_knights;
    int nr_intersections = x_dim*y_dim;
    int nr_nodes = nr_intersections * 2 + 2;
    int source = nr_intersections * 2;
    int sink = nr_intersections * 2 + 1;

    Graph g(nr_nodes);
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);

    for (int i = 0; i < x_dim; i++) {
      for (int j = 0; j < y_dim; j++) {
        int edge_id = j*x_dim + i;

        //add "only go through intersection once constraint"
        add_flow_edge(edge_id, edge_id + nr_intersections, 1, capacity, rev_edge, g);

        //add horizontal edges
        if (i == 0) {
          add_flow_edge(edge_id + nr_intersections, sink, 1, capacity, rev_edge, g);
        }
```

```
        else {
          add_flow_edge(edge_id + nr_intersections, edge_id-1, 1, capacity, rev_edge, g);
        }

        if (i == x_dim - 1) {
          add_flow_edge(edge_id + nr_intersections, sink, 1, capacity, rev_edge, g);
        }
        else {
          add_flow_edge(edge_id + nr_intersections, edge_id + 1, 1, capacity, rev_edge, g);
        }

        //add vertical edges
        if (j == 0) {
          add_flow_edge(edge_id + nr_intersections, sink, 1, capacity, rev_edge, g);
        }
        else {
          add_flow_edge(edge_id + nr_intersections, edge_id - x_dim, 1, capacity, rev_edge, g);
        }

        if (j == y_dim - 1) {
          add_flow_edge(edge_id + nr_intersections, sink, 1, capacity, rev_edge, g);
        }
        else {
          add_flow_edge(edge_id + nr_intersections, edge_id + x_dim, 1, capacity, rev_edge, g);
        }
      }
    }

    for (int i = 0; i < nr_knights; i++) {
      int knight_x, knight_y;
      cin >> knight_x >> knight_y;

      int edge_id = knight_y*x_dim + knight_x;

      add_flow_edge(source, edge_id, 1, capacity, rev_edge, g);
    }

    int flow = push_relabel_max_flow(g, source, sink);
    cout << flow << endl;
  }

  return 0;
}
```

8.19 Tight Words

Keywords: ACM, Dynamic Programming

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#define MOD_NUM 100003

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int test_count;
  cin >> test_count;

  for(int test = 0; test < test_count; test++) {
    int max_word, max_length;
    cin >> max_word >> max_length;

    if(max_length > 0) {
      vector<vector<int> > table(max_word + 1, vector<int>(max_length + 1, 0));

      // init for length 1
      for(int word = 0; word <= max_word; word++) {
        table.at(word).at(1) = 1;
      }

      // go trough length of the word
      for(int length = 2; length <= max_length; length++) {
        // go trough each additional word
        for(int word = 0; word <= max_word; word++) {
          // add current word to a string ending with the previous word
          if(word - 1 >= 0) { // only if there is a "previous word" of course
            table.at(word).at(length) += table.at(word - 1).at(length - 1);
          }

          // add current word to a string ending with the same word
          table.at(word).at(length) += table.at(word).at(length - 1);
```

```
          // add current word to a string ending with the next word
          if(word + 1 <= max_word) {
            table.at(word).at(length) += table.at(word + 1).at(length - 1);
          }

          table.at(word).at(length) = table.at(word).at(length) % MOD_NUM;
        }
      }

      // now we have to count together all the words possible (it can end with any of the words)
      int solution = 0;
      for(int word = 0; word <= max_word; word++) {
        solution += table.at(word).at(max_length);
      }

      cout << solution % MOD_NUM << endl;
    } else {
      cout << "1" << endl;
    }
  }

  return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>

#define MOD_NUM 100003

using namespace std;

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  //cout << fixed << setprecision(0);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_letters, word_length;
    cin >> nr_letters >> word_length;

    if (word_length > 0) {
      vector<vector<int> > dp(word_length + 1, vector<int>(nr_letters + 1, 0));

      for (int i = 0; i <= nr_letters; i++) {
        dp[1][i] = 1;
      }

      for (int i = 2; i <= word_length; i++) {
        for (int j = 0; j <= nr_letters; j++) {
          if (j - 1 >= 0) {
            //cout << "adding1: " << dp[i - 1][j - 1] << endl;
            dp[i][j] += dp[i - 1][j - 1];
          }
          //cout << "adding2: " << dp[i - 1][j] << endl;
          dp[i][j] += dp[i-1][j];
          if (j + 1 <= nr_letters) {
            //cout << "adding3: " << dp[i - 1][j + 1] << endl;
            dp[i][j] += dp[i - 1][j + 1];
          }
          dp[i][j] = dp[i][j] % MOD_NUM;
          //cout << dp[i][j] << " ";
        }
        //cout << endl;
      }
      int solution = 0;
      for (int i = 0; i <= nr_letters; i++) {
        solution += dp[word_length][i];
      }
      solution = solution % MOD_NUM;
      cout << solution << endl;
    }
    else {
      cout << 1 << endl;
    }

  }

  return 0;
}
```

8.20 Cantonal Courier

Keywords: BGL, Graph with edge capacity, Graph with residual capacity, Graph with reverse edges, Max-flow

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <cmath>
#include <climits>
#include <algorithm>
#include <climits>
#include <string>

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

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, int,
    property<edge_residual_capacity_t, int,
    property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

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;

// Custom add_edge, also creates reverse edges with corresponding capacities.
void addEdge(int u, int v, int c, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, Graph &G) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

int main(void)
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);

    int test_cases;
    cin >> test_cases;

    for(int test = 0; test < test_cases; test++) {
        int zone_count, job_count;
        cin >> zone_count >> job_count;

        // create graph for flow analysis
        Graph graph(zone_count + job_count + 2);

        // helper variables to find
        int zone_offset = 0;
        int job_offset = zone_count;
        int source = zone_count + job_count;
        int sink = source + 1;

        // get graph's properties
        EdgeCapacityMap capacity = get(edge_capacity, graph);
        ReverseEdgeMap rev_edge = get(edge_reverse, graph);

        // read in the cost for a zone ticket
        for(int zone_index = 0; zone_index < zone_count; zone_index++) {
            int cost;
            cin >> cost;

            addEdge(zone_offset + zone_index, sink, cost, capacity, rev_edge, graph);
        }

        // read in job rewards
        int total_reward = 0;
        for(int job_index = 0; job_index < job_count; job_index++) {
            int reward;
            cin >> reward;

            total_reward += reward;

            addEdge(source, job_offset + job_index, reward, capacity, rev_edge, graph);
        }

        // read in needed zones
        for(int job_index = 0; job_index < job_count; job_index++) {
            int zones_needed_count;
            cin >> zones_needed_count;
            for(int col = 0; col < zones_needed_count; col++) {
                int zone;
```

```
                cin >> zone;
                zone--;

                addEdge(job_offset + job_index, zone_offset + zone, numeric_limits<int>::max(), capacity, rev_edge, graph
                    );
            }
        }

        // calculate flow which represents the amount of money we have to spent
        int we_pay = push_relabel_max_flow(graph, source, sink);

        cout << total_reward - we_pay << endl;

    }

    return 0;
}
```

```
#include <iostream>
#include <cassert>
#include <vector>
#include <limits>
#include <algorithm>
#include <queue>
#include <set>
#include <utility>
#include <cmath>
#include <stack>

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

using namespace std;
using namespace boost;

typedef adjacency_list<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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> SimpleGraph;

inline void add_flow_edge(int start, int end, int c, EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge, Graph& g) {
  Edge e, rev_e;
  bool success;
  tie(e, success) = add_edge(start, end, g);
  if (success) {
    tie(rev_e, success) = add_edge(end, start, g);
    capacity[e] = c;
    capacity[rev_e] = 0;
    rev_edge[e] = rev_e;
    rev_edge[rev_e] = e;
  }
  else {
    capacity[e] += c;
  }
}

int main() {
  cin.sync_with_stdio(false);
  cout.sync_with_stdio(false);

  int nr_test_cases;
  cin >> nr_test_cases;

  for (int test_case = 0; test_case < nr_test_cases; test_case++) {
    int nr_zones, nr_jobs;
    cin >> nr_zones >> nr_jobs;

    int nr_vertices = nr_zones + nr_jobs + 2;
    Graph g(nr_vertices);
    int source = nr_vertices - 2;
    int sink = nr_vertices - 1;

    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);

    for (int i = 0; i < nr_zones; ++i) {
      int ticket_cost;
      cin >> ticket_cost;
      add_flow_edge(nr_jobs + i, sink, ticket_cost, capacity, rev_edge, g);
    }
    int sum_job_reward = 0;
    for (int i = 0; i < nr_jobs; ++i) {
      int job_reward;
      cin >> job_reward;
      add_flow_edge(source, i, job_reward, capacity, rev_edge, g);
      sum_job_reward += job_reward;
```

```
    }
    for (int i = 0; i < nr_jobs; ++i) {
      int nr_tickets_for_job;
      cin >> nr_tickets_for_job;
      for (int j = 0; j < nr_tickets_for_job; ++j) {
        int ticket_id;
        cin >> ticket_id;
        ticket_id--;
        add_flow_edge(i, nr_jobs + ticket_id, numeric_limits<int>::max(), capacity, rev_edge, g);
      }
    }
    int cost = push_relabel_max_flow(g, source, sink);

    cout << sum_job_reward - cost << endl;
  }

  return 0;
}
```

9 Useful Snippets and Stuff

9.1 General remarks

- around $10'000'000$ operations/iterations per second (ACM slide below claims its less, but oh well)

Asymptotic Running Time

- Rule of Thumb: Processor can do 1M operations per second, Timelimit is 3 seconds.
- n < 1M: Algorithm should be $O(n)$
- n < 100K: Algorithm should be $O(n \log n)$
- n < 1K: Algorithm should be $O(n^2)$
- n < 100: Algorithm should be $O(n^3)$
- n < 50: Algorithm should be $O(n^4)$
- n < 20: Algorithm should be $O(n^5)$ or $O(2^n)$
- n < 10: Algorithm should be $O(n^6)$ or $O(n!)$

9.2 Custom Sorting

```
// sorting
// ATTENTION: be careful, each container has its own order!
#include <vector>
#include <algorithm>

/*
 * #1: Define custom compare function
 */
struct Edge {
  int from, to, weight;
};

bool compare(const Edge& lhs, const Edge& rhs) {
  return lhs.weight > rhs.weight;
}

// ...
vector<Edge> edges;
// ...
sort(edges.begin(), edges.end(), compare);


/* #2: Define the '<' operator for a struct/class
 * Generally, for some type T define 'bool operator<(T other) const {}'
 */
struct Edge {
  int from, to, weight;

  bool operator<(Edge other) const {
    return weight > other.weight;
  }
};

// ...
vector<Edge> edges;
// ...
sort(edges.begin(), edges.end());

/*
 * #3: Define 'operator()'
 */

struct Edge {
  int from, to, weight;
};
```

```
class Compares {
  int ref_weight;

public:
  Compares(const int& weight) : ref_weight(weight) {}

  bool operator()(const Edge& lhs, const Edge& rhs) const {
    return lhs.weight > rhs.weight && lhs.weight >= ref_weight;
  }
};

// ...
Compares comp(100);
// ...
vector<Edge> edges;
// ...
sort(edges.begin(), edges.end(), comp);
```

9.3 CMake Configuration

Add to `CMakeLists.txt`:

```
project( some-project_)

# ^^ start of the CMakeLists.txt ...

# enable C++11
add_definitions("-std=c++11")

# enable all warnings
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -Wall")
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -Wall")

# enable debug always
set(CMAKE_BUILD_TYPE "Debug")

# additional information regarding the makefile creation, not really useful I think
set(CMAKE_VERBOSE_MAKEFILE ON)

# ... rest of the CMakeLists.txt ...
cmake_minimum_required(VERSION 2.6.2)
```

Debugging can also be enabled by calling `cmake -DCMAKE_BUILD_TYPE=Debug`, however this adds only the `-g` flag!

9.4 CMake and CGAL

```
# Step 0: IMPORTANT: first, create the C++ source code file

# Step 1: call CGAL CMake script
cgal_create_cmake_script

# Step 2: modify CMakeLists.txt if needed (adding C++11 support, debugging, etc.)
vim CMakeLists.txt
# or
nano CMakeLists.txt
# or ...

# Step 3: call CMake, don't forget the dot
cmake .

# Step 4: from now on always enough to call make
make

# Step 5: execute application
```

9.5 BGL

URL for normal graph functions: `https://judge.inf.ethz.ch/doc/boost/libs/graph/doc/graph_concepts.html`. Also good starting point is the "A Quick Tour of the Boost Graph Library" (see TOC).

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

#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp> // tuples::ignore

// BGL algo specific includes ...
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/strong_components.hpp>
// .. end

using namespace std;
using namespace boost;

// Directed graph with int weights on edges.
typedef adjacency_list<vecS, vecS, directedS, no_property, property<edge_weight_t, int> > Graph;
```

```
// ... or one with multiple properties for some fancy algorithm
typedef adjacency_list<vecS, vecS, directedS, property<vertex_name_t, string,
  property<vertex_distance_t, int> > > Graph;

  // and don't forget:
  typedef property_map<Graph, vertex_name_t>::type NameMap;

// Edge type (edge descriptor in BGL speak).
typedef graph_traits<Graph>::edge_descriptor Edge;

// Edge iterator.
typedef graph_traits<Graph>::edge_iterator EdgeIterator;

// Out Edge iterator (directed graph)
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;

// Map edge -> weight.
typedef property_map<Graph, edge_weight_t>::type WeightMap;

// ...

void main() {
  // create graph with `n` vertices
  Graph graph(n);

  // accessing a property map
  NameMap name_map = get(vertex_name, graph_instance);

  // iterate over all outgoing (directed graph) edges
  OutEdgeIterator out_edge_iterator, out_edge_end;
  for(tie(out_edge_iterator, out_edge_end) = out_edges(some_vertices, graph);
    out_edge_iterator != out_edge_end;
    ++out_edge_iterator) {
    // get the other end's vertex
    int other_vertex = target(*out_edge_iterator, graph);
  }
}
```

```
typedef CGAL::Quadratic_program<IT> P;
typedef CGAL::Quadratic_program_solution<ET> S;
```

- The input type `IT`
  - Typically `int` or `double`
- The exact type `ET` (check in the manual)
  - Requirement: input type can be converted to exact type
  - `int` / `double` (not recommended)
  - GMP
    - ⋆ `CGAL::Gmpz` integral numbers
    - ⋆ `CGAL::Gmpq` rational numbers
    - ⋆ `CGAL::Gmpzf` "floating point" numbers
  - `CGAL::MP_Float` "floating point" numbers
- The solution type `CGAL::Quotient<ET>`

Multiplication in `CGAL::MP_Float` has complexity $\Theta(n^2)$, multiplication for the GMP number types uses a faster algorithm (depends on the magnitude), asymptotic runtime $\approx n^{<1.5}$

`QuadraticProgram`

$$\min x^T D x + c^T x + c_0 \quad \text{s.t. } Ax \lesseqgtr b, \quad \ell \le x \le u$$

`NonnegativeQuadraticProgram`

$$\min x^T D x + c^T x + c_0 \quad \text{s.t. } Ax \lesseqgtr b, \quad x \ge 0$$

`LinearProgram`

$$\min c^T x + c_0 \quad \text{s.t. } Ax \lesseqgtr b, \quad \ell \le x \le u$$

`NonnegativeLinearProgram`

$$\min c^T x + c_0 \quad \text{s.t. } Ax \lesseqgtr b, \quad x \ge 0$$

The solvers
`solve_nonnegative_{linear/quadratic}_program()`
will completely ignore any manually set lower or upper bounds ℓ or u.

Debugging:

- Check the dimensions: `lp.get_n()`, `lp.get_m()`
- `CGAL::print_linear_program(std::cerr, lp, "lp");`

Change the pivot rule

- Bland's pivot rule avoids cycling (but it is slower...)

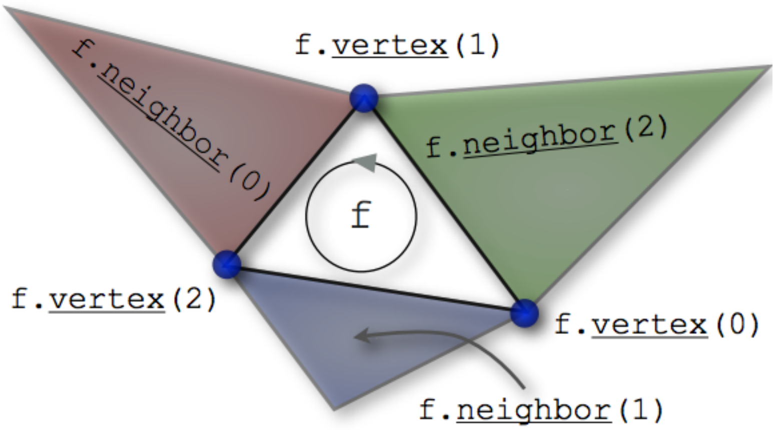
```
CGAL::Quadratic_program_options options;
options.set_pricing_strategy(CGAL::QP_BLAND);
Solution s = CGAL::SOLVER(program, ET(), options);
```

BUG - Don't Forget!

There is a bug in the assignment `operator=` for `Solution` objects...

9.7 CGAL: Approximation (Triangulation)

Nex image '`f`' is a '`face handle`'




```
...
Triangulation::Edge e;
...
// get the vertices of e
Triangulation::Vertex_handle v1 = e.first->vertex((e.second + 1) % 3);
Triangulation::Vertex_handle v2 = e.first->vertex((e.second + 2) % 3);
std::cout << "e = " << v1->point() << " <-> " << v2->point() << std::endl;
...
```

9.8 Matching

- Undirected graph $G = (V, E)$
- Is a subset $M \subseteq E$ of edges
- Each pair of edges of the matching set $(e_1, e_2 \in M)$ don't have a common vertices $v \in V$
- Or based on the vertices: Every pair of vertices $v_1, v_2 \in V$ aren't the start or end point of an edge $e \in M$
- **Maximal Matching**: If it's not possible to add another edge $e \in E$
  M to M and M being a matching
- **Maximum Matching/Maximum Cardinality Matching**: A maximal matching with the largest amount of edges. There might exists multiple maximum matchings.

9.9 Vertex Cover

- Undirected graph $G = (V, E)$
- A vertex cover is a subset $V' \subseteq V$ such that for each edge $(u, v) =: e \in E$ $u \in V'$ and/or $v \in V'$
- **Minimal Vertex Cover**: Generally NP-complete. Is the smallest possible vertex cover.

9.10 König's Theorem

- only for bipartite graphs
- The size of a maximum matching (maximum cardinality matching) is equal to the minimal vertex cover

Algorithm:

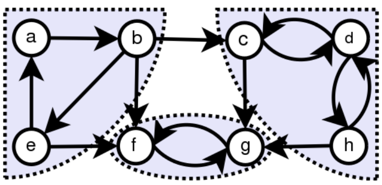
1. Let $G = (V, E)$ be a undirected bipartite graph, i.e. $L \cup R = V \wedge L \cap R = \emptyset$
2. Calculate the Maximum Matching (Maximum Cardinality Matching), resulting in a matching $M \subseteq E$
3. Mark all vertices $v \in L$ that are not in M $(v \notin M)$ as visited
4. Start at visited vertices and do a vertices search (DFS) from L to R along edges from $V\ M$ and R to L along edges from M. Each such visited vertex is marked as visited
5. All unvisited vertices in L and all visited in R are part of the minimal vertex cover

9.11 Connected Component

- For undirected graphs
- Connected component is a subgraph of a graph where any two vertices in the subgraph are connected by a path, but no path exists to vertices to ones outside the subgraph

9.12 Strongly Connected Component

- For directed graphs
- Partitions a graph into subgraphs where each subgraph is strongly connected.
- **Strongly Connected**: A graph is strongly connected iff every vertex of the graph is reachable from every other vertex



From: `https://commons.wikimedia.org/wiki/File:Scc.png`

9.13 Biconnected Component / Articulation Points

See BGL documentation, it's pretty good. Otherwise this might help:

- **Biconnected Graph**: A connected graph is biconnected iff removing any single vertex (and all edges from/to this vertex) can't disconnect the graph. Such a graph doesn't contain *articulation points*!
- **Biconnected Components**: A subset of vertices where removing one of these vertices doesn't result in a disconnected graph inside the subgraph. Vertices can belong to multiple biconnected components!
- **Articulation Points**: These are the vertices that belong to more than one biconnected component. Such vertices are called *articulation points*. Removing such a articulation point would result in an increase of connected components (i.e. increase the number of graphs). No such articulation points means the graph is biconnected!

9.14 BGL: DFS / BFS

```
#include <iostream>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/depth_first_search.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

class BFSVisitorClass: public default_bfs_visitor {
public:
  BFSVisitorClass() {
    cout << "here do some initialisation before passing to breadth_first_search()" << endl;
  }

  void initialize_vertex(Vertex v, const Graph& graph) {
    cout << "initialize_vertex() for vertex: " << v << endl;
  }

  void discover_vertex(Vertex v, const Graph& graph) {
    cout << "discover_vertex() for vertex: " << v << endl;
  }

  // this is called right before all edges are taken and tree_edge() calls done
  void examine_vertex(Vertex v, const Graph& graph) {
    cout << "examine_vertex() for vertex: " << v << endl;
  }

  void examine_edge(Edge e, const Graph& graph) {
    cout << "examine_edge() for edge: " << e << endl;
  }

  // this one is interesting if you want to know the visited/followed edges
  // this edge forms the search tree
  void tree_edge(Edge e, const Graph& graph) {
    cout << "tree_edge() for edge: " << e << endl;
  }

  void non_tree_edge(Edge e, const Graph& graph) {
    cout << "non_tree_edge() for edge: " << e << endl;
  }

  void gray_target(Edge e, const Graph& graph) {
    cout << "gray_target() for edge: " << e << endl;
  }

  void black_target(Edge e, const Graph& graph) {
    cout << "black_target() for edge: " << e << endl;
  }

  void finish_vertex(Vertex v, const Graph& graph) {
    cout << "finish_vertex() for vertex: " << v << endl;
  }
};

class DFSVisitorClass: public default_dfs_visitor {
public:
  DFSVisitorClass() {
    cout << "here do some initialisation before passing to depth_first_search()" << endl;
  }

  void initialize_vertex(Vertex v, const Graph& graph) {
    cout << "initialize_vertex() for vertex: " << v << endl;
  }

  void start_vertex(Vertex v, const Graph& graph) {
    cout << "start_vertex() for vertex: " << v << endl;
  }

  void discover_vertex(Vertex v, const Graph& graph) {
    cout << "discover_vertex() for vertex: " << v << endl;
  }

  void examine_edge(Edge e, const Graph& graph) {
    cout << "examine_edge() for edge: " << e << endl;
  }

  void tree_edge(Edge e, const Graph& graph) {
    cout << "tree_edge() for edge: " << e << endl;
  }

  void back_edge(Edge e, const Graph& graph) {
    cout << "back_edge() for edge: " << e << endl;
  }

  void forward_or_cross_edge(Edge e, const Graph& graph) {
    cout << "forward_or_cross_edge() for edge: " << e << endl;
  }
```

```
  void finish_vertex(Vertex v, const Graph& graph) {
    cout << "finish_vertex() for vertex: " << v << endl;
  }
};

int main() {
  /*
                0
               / \
              /   \
             1     2
            /\     |
           /  \    |
          3    4   5
               |
               6
    */

  Graph graph(8);
  add_edge(0, 1, graph);
  add_edge(1, 3, graph);
  add_edge(1, 4, graph);
  add_edge(2, 5, graph);
  add_edge(4, 6, graph);

  cout << "Depth First Search:" << endl;
  depth_first_search(graph, visitor(DFSVisitorClass()));
  cout << endl << endl;

  cout << "Breath First Search" << endl;
  breadth_first_search(graph, 0, visitor(BFSVisitorClass()));
  //                          ^- source to start from

  return 0;
}
```

```
# OUTPUT
Depth First Search:
here do some initialisation before passing to depth_first_search()
initialize_vertex() for vertex: 0
initialize_vertex() for vertex: 1
initialize_vertex() for vertex: 2
initialize_vertex() for vertex: 3
initialize_vertex() for vertex: 4
initialize_vertex() for vertex: 5
initialize_vertex() for vertex: 6
initialize_vertex() for vertex: 7
start_vertex() for vertex: 0
discover_vertex() for vertex: 0
examine_edge() for edge: (0,1)
tree_edge() for edge: (0,1)
discover_vertex() for vertex: 1
examine_edge() for edge: (1,0)
back_edge() for edge: (1,0)
examine_edge() for edge: (1,3)
tree_edge() for edge: (1,3)
discover_vertex() for vertex: 3
examine_edge() for edge: (3,1)
back_edge() for edge: (3,1)
finish_vertex() for vertex: 3
examine_edge() for edge: (1,4)
tree_edge() for edge: (1,4)
discover_vertex() for vertex: 4
examine_edge() for edge: (4,1)
back_edge() for edge: (4,1)
examine_edge() for edge: (4,6)
tree_edge() for edge: (4,6)
discover_vertex() for vertex: 6
examine_edge() for edge: (6,4)
back_edge() for edge: (6,4)
finish_vertex() for vertex: 6
finish_vertex() for vertex: 4
finish_vertex() for vertex: 1
finish_vertex() for vertex: 0
start_vertex() for vertex: 2
discover_vertex() for vertex: 2
examine_edge() for edge: (2,5)
tree_edge() for edge: (2,5)
discover_vertex() for vertex: 5
examine_edge() for edge: (5,2)
back_edge() for edge: (5,2)
finish_vertex() for vertex: 5
finish_vertex() for vertex: 2
start_vertex() for vertex: 7
discover_vertex() for vertex: 7
finish_vertex() for vertex: 7


Breath First Search
here do some initialisation before passing to breadth_first_search()
initialize_vertex() for vertex: 0
initialize_vertex() for vertex: 1
initialize_vertex() for vertex: 2
initialize_vertex() for vertex: 3
initialize_vertex() for vertex: 4
```

```
initialize_vertex() for vertex: 5
initialize_vertex() for vertex: 6
initialize_vertex() for vertex: 7
discover_vertex() for vertex: 0
examine_vertex() for vertex: 0
examine_edge() for edge: (0,1)
tree_edge() for edge: (0,1)
discover_vertex() for vertex: 1
finish_vertex() for vertex: 0
examine_vertex() for vertex: 1
examine_edge() for edge: (1,0)
non_tree_edge() for edge: (1,0)
black_target() for edge: (1,0)
examine_edge() for edge: (1,3)
tree_edge() for edge: (1,3)
discover_vertex() for vertex: 3
examine_edge() for edge: (1,4)
tree_edge() for edge: (1,4)
discover_vertex() for vertex: 4
finish_vertex() for vertex: 1
examine_vertex() for vertex: 3
examine_edge() for edge: (3,1)
non_tree_edge() for edge: (3,1)
black_target() for edge: (3,1)
finish_vertex() for vertex: 3
examine_vertex() for vertex: 4
examine_edge() for edge: (4,1)
non_tree_edge() for edge: (4,1)
black_target() for edge: (4,1)
examine_edge() for edge: (4,6)
tree_edge() for edge: (4,6)
discover_vertex() for vertex: 6
finish_vertex() for vertex: 4
examine_vertex() for vertex: 6
examine_edge() for edge: (6,4)
non_tree_edge() for edge: (6,4)
black_target() for edge: (6,4)
finish_vertex() for vertex: 6
```

Index