ETH_Algorithm_Lab

Code and assessment of the course Algorithm Labs. There is one problem of the week (PoW) and multiple problems during the week. The codes are implemented in C++.

Note that only problems more (or equal) than 3 sub-questions will be tested in the final exam. So I label these more important questions with bold style in the division table.

Divide by Week

Week	Q1	Q2	Q3	Q4	PoW
1	Build the Sum	Dominoes	**Even Pairs**	**Even Matrices**	
2	**Beach Bars**	**Buring Coins**	**The Great Game**	**Defensive Line**	**Deck of Cards**
3	First Hit	Hit	Antenna	Hiking Maps	**From Russia with Love**
4	First Step with BGL	Important Bridges	**Buddy Selection**	**Ant Challenge**	**Fighting Pits of Meereen**
5	**Boats**	**Moving Books**	**Severus Snape**	**Asterix the Gaul**	**Motocycles**
6	What is the Maximum	Diet	Inball	**Lannister**	**Planet Express**
7	Shopping Trip	**Knights**	Coin Tossing Tournament	**London**	**Octopussy**
8	Germs	Bistro	**H1N1**	Light the Stage	**Suez**
9	**Algocoon**	**Real Estate Market**	**Canteen**	Placing Knights	**Kingdom_Defence**
10	**Asterix and the Chariot Race**	Evolution	**WorldCup**	**Asterix in Swizerland**	**Golden Eye**
11	**The Iron Islands**	**Return of the Jedi**	**Idefix**	**Legions**	**Phantom Menace**
12	**On Her Majesty's Secret Service**	**Hong Kong**	**Car Sharing**	**Bonus Level**	**San Francisco**
13	**Hand**	**Hagrid**	Punch	**Ludo Bagman**	**Clues**
14					**India**

Divide by Topic

Topic	Questions
Dynamic Programming	**Bonus Level**, **Asterix and the Chariot Race**, **Even Matrices**, **Fighting Pits of Meereen**, **From Russia with Love**, **Hagrid**, **San Francisco**, **The Great Game**
Linear Programming	**Lannister**, **Legions**, **Suez**, **WorldCup**
Sliding Window	**Beach Bars**, **Deck of Cards**, **Defensive Line**, Hiking Maps, **The Iron Islands**
Greedy	**Asterix the Gaul**, **Boats**, **Moving Books**, **Severus Snape**
Graph	**Ant Challenge**, **Buddy Selection**, **Planet Express**
Max Flow	**Algocoon**, **Canteen**, **Car Sharing**, **India**, **Kingdom_Defence**, **Knights**, **London**, **Ludo Bagman**, **Phantom Menace**, Placing Knights, **Real Estate Market**, **Asterix in Swizerland**
CGAL	**Motocycles**
Triangulation	**Clues**, **Golden Eye**, **H1N1**, **Hand**, **Hong Kong**, **Idefix**, Light the Stage

Useful Codes

- Sliding Window

```
// [) interval
int head = 0, tail = 1, sum = cards[head], best_val = INT_MAX;
pair<int, int> solution = make_pair(head, tail-1);
while(true){
    int val = abs(sum - k);
    if(val < best_val){
        best_val = val;
        solution = make_pair(head, tail-1);
    }
  if(sum==k) break;
  if(sum < k){
    if(tail==cards.size()) break;
    sum += cards[tail++];
  } else {
    sum -= cards[head++];
  }
}
```

```
int sliding_window(vector<int>& costs, vector<int>& water_way, int query){
    int left = 0, right = 0, max_result = -1, sum = 0;
    while(true){
        int num_of_island = right - left;
        if(sum==query){
            max_result = max(max_result, num_of_island);
            sum -= costs[water_way[left++]];
        }
        else if(sum < query ){
            if(right == (int)water_way.size()) break;
            sum += costs[water_way[right++]];
        } else {
            sum -= costs[water_way[left++]];
        }
    }
    return max_result;
}
```

- Binary search
  - upperbound

```
while(l < r){
        int mid = (l + r + 1)/2;
        c_map[boost::edge(v_src, k, G).first] = mid;
        if(feasible(G, v_src, v_tar, mid)){
            l = mid;
        } else {
            r = mid - 1;
        }
    }
```

  - Lower bound

- Lambda:

```
void preprocess(vector<Chamber>& chamber_list, int chamber_id){
    for(auto child : chamber_list[chamber_id].child_list){
        preprocess(chamber_list, child.first);
        chamber_list[chamber_id].number_of_node += chamber_list[child.first].number_of_node;
        chamber_list[chamber_id].time_cost += child.second +
chamber_list[child.first].time_cost;
    }
    sort(chamber_list[chamber_id].child_list.begin(),
chamber_list[chamber_id].child_list.end(),
        [&chamber_list, chamber_id](auto& left, auto& right)-> bool {
            long time_cost1 = chamber_list[left.first].time_cost + left.second;
            long number_of_node1 = chamber_list[left.first].number_of_node;
            long time_cost2 = chamber_list[right.first].time_cost + right.second;
            long number_of_node2 = chamber_list[right.first].number_of_node;
            return time_cost1 * number_of_node2 < time_cost2 * number_of_node1;
        }
    );
}
```

- CGAL:
  - Check Intersection: `CGAL::do_intersect`
  - Find Intersection:

```
auto o = CGAL::intersection(ray,segments[i]);
if (const P* op = boost::get<P>(&*o))
  intersect_point = *op;
else if (const S* os = boost::get<S>(&*o)) {
  if(CGAL::squared_distance(start, os->source()) < CGAL::squared_distance(start, os-
>target()))
    intersect_point = os->source();
  else intersect_point = os->target();
}
```

  - squared distance: `CGAL::squared_distance`
  - check the cross product: `Line.oriented_side(Point)`
  - Floor to double and output:

```
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;
}
cout << fixed << setprecision(0) <<...<< endl;
```

- Triangulation:
  - BFS construction

```
void BFS_construction(Vh vertex, Triangulation& tri, graph& G){
    set<Vh> visited;
    vector<Vh> queue; queue.push_back(vertex); visited.insert(vertex);
    while(!queue.empty()){
        auto next_vertex = queue[queue.size() - 1]; queue.pop_back();
        if(next_vertex != vertex) boost::add_edge(vertex->info(), next_vertex->info(), G);
        auto neighbor = next_vertex->incident_vertices();

        do {
            if (!tri.is_infinite(neighbor) && visited.find(neighbor) == visited.end() && CGAL::squared_distance(vertex->point(), neighbor->point()) <= r_square) {
                visited.insert(neighbor);
                queue.push_back(neighbor);
            }
        } while(++neighbor != next_vertex->incident_vertices());
    }
}
```

  - Dijkstra construction

```
void precompute(Triangulation& tri){
    priority_queue< pair<FT, Face> > q;
    for(auto face = tri.all_faces_begin(); face != tri.all_faces_end(); face++){
        if(tri.is_infinite(face)){
            q.push(make_pair(LONG_MAX, face));
            face->info() = LONG_MAX;
        } else {
            Point outheart = tri.dual(face);
            FT distance = CGAL::squared_distance(outheart, face->vertex(0)->point());
            q.push(make_pair(distance, face));
            face->info() = distance;
        }
    }

    while(!q.empty()){
        FT distance = q.top().first;
        Face current_face = q.top().second;
        q.pop();
        // current face is updated by another better face
        if(distance < current_face->info()) continue;
        for(int i = 0; i < 3; i++){
            Face next_face = current_face->neighbor(i);
            FT edge_length = tri.segment(current_face, i).squared_length();
            FT new_distance = min(current_face->info(), edge_length);
            if(new_distance > next_face->info()){
                q.push(make_pair(new_distance, next_face));
                next_face->info() = new_distance;
            }
        }
    }
}
```

-

Assessment 1: From Russia with Love

Problem Definition

- **Input**:
  - n: number of coins
  - m: number of players
  - k: the player you are intereseted in
  - values $x_0, \ldots, x_{n-1}$ s.t. x_i denotes the price of i-th coin
- **Rule**: m players take turns from player 0, pick up a coin from either the front or the end of the array of coins.
- **Output**: Largest winnings that player k can collect, **regardless of how other players play**. (even all other players play against player k, the maximum value k can get)

Mathematical Concept

- **Recursion**: The problem can be transformed recursively into subproblems.
- **Reformulation**: This problem is similar to a zero-sum game, the difference is that there are more than two players, and all other players play against player k. We define the score that player k can get when it is player p's turn and the remaining coins are from l to r.

$$
f(l,r,p) = \begin{cases} x_l, & l = r \wedge p = k \\ 0, & l = r \wedge p \neq k \\ max[x_l + f(l+1, r, (p+1)\ mod\ m),\ x_r + f(l, r-1, (p+1)\ mod\ m)], & l \neq r \wedge p = k \\ min[f(l+1, r, (p+1)\ mod\ m),\ f(l, r-1, (p+1)\ mod\ m)], & otherwise \end{cases}
$$

Algorithm

- **Recursion**: Just from the mathematical formulation
  - Running time: $O(2^n)$ (For each decision, there are two possibilities)

```
int score(int head, int tail, int player){
    if(head==tail) return (player!=k?0:coins[head]);
    int score1 = (player!=k?0:coins[head]) + score(head+1, tail, (player+1)%m);
    int score2 = (player!=k?0:coins[tail]) + score(head, tail-1, (player+1)%m);
    return (player==k ? max(score1, x):min(score1, score2));
}
```

- **Dynamic Programming (Top-Down)**
  - Since the player is fixed for a specific array, we only need to record the state as `dp[head][tail]`.
  - Running time: $O(n^2)$ (For each `dp[head][tail]`, we only need to compute once. $\binom{n}{2}$ in total.)

```
int score(int head, int tail, int player){
    if(dp[head][tail]!=-1) return dp[head][tail];
    if(head==tail) return (dp[head][tail] = (player!=k?0:coins[head]));
    int score1 = (player!=k?0:coins[head]) + score(head+1, tail, (player+1)%m);
    int score2 = (player!=k?0:coins[tail]) + score(head, tail-1, (player+1)%m);
    return (dp[head][tail] = (player==k ? max(score1, x):min(score1, score2)));
}
```

Implementation & Testing

```
#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;
#define MAX_C 1001
#define MAX_P 501
int t, n, m, k, coins[MAX_C], dp[MAX_C][MAX_C];

int score(int head, int tail, int player){
    if(dp[head][tail]!=-1) return dp[head][tail];
    if(head==tail) return (dp[head][tail] = 0);
    int score1 = (player!=k?0:coins[head]) + score(head+1, tail, (player+1)%m);
    int score2 = (player!=k?0:coins[tail]) + score(head, tail-1, (player+1)%m);
    return (dp[head][tail] = (player==k ? max(score1, score2):min(score1, score2)));
}

int main(){
    ios_base::sync_with_stdio(false);
    cin >> t;
    while(t--){
        cin >> n >> m >> k;
        memset(dp, -1, sizeof(dp));
        for(int i = 0; i < n; i++) cin >> coins[i];
        cout << score(0,n-1,0) <<endl;
    }
    return 0;
}
```

- When $m = 2$, it is equaivalent to a zero-sum game. Use min-max algorithm.

```
#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;
#define MAX_C 1001
#define MAX_P 501
int t, n, m, k, coins[MAX_C], sum[MAX_C], dp[MAX_C][MAX_C];

int score(int head, int tail, int player){
    if(dp[head][tail]!=-1) return dp[head][tail];
    if(head==tail) return (dp[head][tail] = 0);
    int score1 = (player!=k?0:coins[head]) + score(head+1, tail, (player+1)%m);
    int score2 = (player!=k?0:coins[tail]) + score(head, tail-1, (player+1)%m);
    return (dp[head][tail] = (player==k ? max(score1, score2):min(score1, score2)));
}
```

```

int score2(int head, int tail){
    if(dp[head][tail]!=-1) return dp[head][tail];
    if(head==tail) return (dp[head][tail] = 0);
    int scorea = coins[head] + score2(head+1, tail);
    int scoreb = coins[tail] + score2(head, tail-1);
    int total = sum[tail + 1] - sum[head];
    return (dp[head][tail] = total - max(scorea, scoreb));
}

int main(){
    ios_base::sync_with_stdio(false);
    cin >> t;
    while(t--){
        cin >> n >> m >> k;
        memset(sum, 0, sizeof(sum));
        memset(dp, -1, sizeof(dp));
        for(int i = 0; i < n; i++){
            cin >> coins[i];
            sum[i+1] = sum[i] + coins[i];
        }
        cout << ((m==2)?score2(0, n-1):score(0,n-1,0)) <<endl;
    }
    return 0;
}
```

```
#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;
#define MAX_C 1001
#define MAX_P 501
int t, n, m, k, coins[MAX_C], dp[MAX_C][MAX_C];

int score(int head, int tail, int player){
    if(dp[head][tail]!=-1) return dp[head][tail];
    if(head==tail) return (dp[head][tail] = (player!=k?0:coins[head]));
    int score1 = (player!=k?0:coins[head]) + score(head+1, tail, (player+1)%m);
    int score2 = (player!=k?0:coins[tail]) + score(head, tail-1, (player+1)%m);
    return (dp[head][tail] = (player==k ? max(score1, score2):min(score1, score2)));
}

int main(){
    ios_base::sync_with_stdio(false);
    cin >> t;
    while(t--){
        cin >> n >> m >> k;
        memset(dp, -1, sizeof(dp));
        for(int i = 0; i < n; i++) cin >> coins[i];
        cout << score(0,n-1,0) <<endl;
    }
    return 0;
}
```

Overall

- Understand the question: I firstly understand this problem in a different way. (I thought it asks me to find the maximum score that player k can get, which means all the other players are helping him!)
- Find a bug when $m = 2$.

Assessment 2: Lannister

Problem Definition

- **Input**:
  - n noble houses
  - m common houses
  - Two canals (one for sewage one for fresh water) cross at right angles
  - sewage pipe: horizontal, fresh water pipe: vertical
  - Constraints
    - Cersei: noble houses at the LHS of the sewage cannal, common houses at the RHS of the sewage cannal.
    - Tywin: sum of sewage pipe length should be less than a threshold s.
    - Jaime: minimize the length of the longest fresh water pipe.
- **Output**:
  - check the feasibility of Cersei's and Tywin's constraints
  - minimum length of the longest fresh water pipe

Mathematical Concept

- Assume the sewage canal is $ax + by + c = 0$, then the fresh canal can be represented as $bx - ay + d = 0$, which is perpendicular to the sewage canal. (x_i, y_i) is the position of house i.
- Cersei's constraint:

$$\begin{cases} ax_i + by_i + c \leq 0 \ (i \in noble) \\ ax_i + by_i + c \geq 0 \ (i \in common) \end{cases}, \ a > 0 \ (enforce\ the\ LHS\ RHS\ order)$$

- Tywin's constraint:

$$dist_{sewages} = \sum_{i \in houses} |x_i - (-\frac{b}{a}y_i - \frac{c}{a})| = \sum_{i \in houses} |x_i + \frac{b}{a}y_i + \frac{c}{a}| \leq s \tag{2}$$

- Jaime's constraint: if we set the upperbound of lengths of fresh water pipes to be l

$$dist_{fresh}(i) = |y_i - (\frac{b}{a}x_i + \frac{d}{a})| \leq l \quad minmize\ l \tag{3}$$

Algorithm

- Linear programming: we can transform the constraints into a linear programming problem format.
  - object function: l
  - Variables: a, b, c, d , l
- Transformed constraints
  - Cersei's constraint

$$\begin{cases} x_i a + y_i b + c \leq\ 0\ (i\ \in noble) \\ (-x_i)a + (-y_i)b + (-1)c \leq\ 0\ (i\ \in common) \end{cases}$$
$$a > 0\ (enforce\ the\ LHS\ RHS\ order)$$

- Tywin's constraint:

$$\begin{aligned} dist_{sewages} &= \sum_{i\in\ houses} |x_i + \frac{b}{a}y_i + \frac{c}{a}| \\ &= \sum_{i\in\ noble\ houses} -(x_i + \frac{b}{a}y_i + \frac{c}{a}) + \sum_{i\in\ common\ houses} (x_i + \frac{b}{a}y_i + \frac{c}{a}) \\ &= (\sum_{i\in\ common} x_i - \sum_{i\in\ noble} x_i) + \frac{b}{a}((\sum_{i\in\ common} y_i - \sum_{i\in\ noble} y_i)) + \frac{c}{a}(m-n) \leq s \\ a(\sum_{i\in\ common} x_i - \sum_{i\in\ noble} x_i - s) &+ b((\sum_{i\in\ common} y_i - \sum_{i\in\ noble} y_i)) + c(m-n) \leq 0 \end{aligned} \tag{4}$$

- Jaime's constraint:

$$\begin{aligned} dist_{fresh}(i) &= |y_i - (\frac{b}{a}x_i + \frac{d}{a})| \leq\ l \\ max[(\frac{b}{a}x_i + \frac{d}{a}) - y_i,\ y_i - (\frac{b}{a}x_i + \frac{d}{a})] &\leq l \\ (\frac{b}{a}x_i + \frac{d}{a}) - y_i \leq l\ \wedge y_i &- (\frac{b}{a}x_i + \frac{d}{a}) \leq l \\ minmize\ l & \end{aligned} \tag{5}$$

  - just add two constraint to fulfill this constraint.

Implementation & Testing

- Set $a = 1 > 0$

```
lp.set_l(a, true, 1); lp.set_u(a, true, 1);
```

- Tywin's constraint:

$$\begin{aligned} dist_{sewages} = (\sum_{i\in\ common} x_i - \sum_{i\in\ noble} x_i) + b((\sum_{i\in\ common} y_i - \sum_{i\in\ noble} y_i)) + c(m-n) \leq s \\ b((\sum_{i\in\ common} y_i - \sum_{i\in\ noble} y_i)) + c(m-n) \leq s - (\sum_{i\in\ common} x_i - \sum_{i\in\ noble} x_i) \end{aligned} \tag{6}$$

- Jaime's constraint:

$$\begin{aligned} (bx_i + d) - y_i \leq l\ \wedge y_i - (bx_i + d) \leq l \\ (bx_i + d) - l \leq y_i\ \wedge -(bx_i + d) - l \leq -y_i \end{aligned} \tag{7}$$

- Code
  - implement first two constraints then add Jaime's constraint
  - Use of counter to count row in matrix A
  - use `long` instead of `int`

```
///3
#include <iostream>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
```

```cpp
#include <CGAL/Gmpz.h>
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;
typedef long IT;
typedef CGAL::Gmpz ET;
typedef CGAL::Quadratic_program<IT> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int t, n, m;
long s;


void solve(){
    cin >> n >> m >> s;
    vector< pair<long, long> > nobles(n);
    vector< pair<long, long> > commons(m);

    int number_of_constraints = 0;
    long noble_sum_x = 0, noble_sum_y = 0;
    Program lp (CGAL::SMALLER, false, 0, false, 0);
    const int a = 0, b= 1, c = 2, d = 3, l = 4;
    for(int i = 0; i < n; i++){
        cin >> nobles[i].first >> nobles[i].second;
        noble_sum_x += nobles[i].first;
        noble_sum_y += nobles[i].second;
        lp.set_a(a, number_of_constraints, nobles[i].first);
        lp.set_a(b, number_of_constraints, nobles[i].second);
        lp.set_a(c, number_of_constraints, 1);
        number_of_constraints++;
    }
    long common_sum_x = 0, common_sum_y = 0;
    for(int i = 0; i < m; i++){
        cin >> commons[i].first >> commons[i].second;
        common_sum_x += commons[i].first;
        common_sum_y += commons[i].second;
        lp.set_a(a, number_of_constraints, -commons[i].first);
        lp.set_a(b, number_of_constraints, -commons[i].second);
        lp.set_a(c, number_of_constraints, -1);
        number_of_constraints++;
    }
    // set a = 1 to simplify the second constraint
    lp.set_l(a, true, 1); lp.set_u(a, true, 1);
    Solution sol = CGAL::solve_linear_program(lp, ET());
    if(sol.is_infeasible()) {cout << "Yuck!\n"; return;}

    if(s != -1){
        lp.set_a(b, number_of_constraints, common_sum_y - noble_sum_y);
        lp.set_a(c, number_of_constraints, m - n);
        lp.set_b(number_of_constraints, s - common_sum_x + noble_sum_x);
        number_of_constraints++;
        sol = CGAL::solve_linear_program(lp, ET());
        if(sol.is_infeasible()) {cout << "Bankrupt!\n"; return ;}
    }

```

```
    for(int i = 0; i < n; i++){
        lp.set_a(b, number_of_constraints, -nobles[i].first);
        lp.set_a(d, number_of_constraints, -1);
        lp.set_a(l, number_of_constraints, -1);
        lp.set_b(number_of_constraints, -nobles[i].second);
        number_of_constraints++;

        lp.set_a(b, number_of_constraints, nobles[i].first);
        lp.set_a(d, number_of_constraints, 1);
        lp.set_a(l, number_of_constraints, -1);
        lp.set_b(number_of_constraints, nobles[i].second);
        number_of_constraints++;
    }

    for(int i = 0; i < m; i++){
        lp.set_a(b, number_of_constraints, -commons[i].first);
        lp.set_a(d, number_of_constraints, -1);
        lp.set_a(l, number_of_constraints, -1);
        lp.set_b(number_of_constraints, -commons[i].second);
        number_of_constraints++;

        lp.set_a(b, number_of_constraints, commons[i].first);
        lp.set_a(d, number_of_constraints, 1);
        lp.set_a(l, number_of_constraints, -1);
        lp.set_b(number_of_constraints, commons[i].second);
        number_of_constraints++;
    }
    lp.set_l(l, true, 0);
    lp.set_c(l, 1);

    sol = CGAL::solve_linear_program(lp, ET());
    cout << fixed << setprecision(0) << ceil(CGAL::to_double(sol.objective_value())) << endl;
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin >> t;
    while(t--){
        solve();
    }
    return 0;
}
```

Overall

- Understand the question: too long...
- Set up the constraints in linear programming format (need trick for all the constraints)
    - Cersei: linearly separable **but should be on the same side**
    - Tywin : **need to deal with the absolute sign properly.** (by dividing into two cases)
    - Jamie:
        - **need to deal with the absolute sign properly** (by adding two constraints)
        - **need to find special value for a** (otherwise becomes quadratic problem)
- Debug: use `long` instead of `int`

Assessment 3: Kingdom Defense

Problem Definition

- **Inputs**:
  - l: number of locations ($1 \leq l \leq 500$)
    - Each one corresponds to a pair of g (number of solder stationed), d (number of solder needed to defend the city)
  - p: number of paths ($1 \leq p \leq l^2$)
    - min flow c and max flow C
- **Outputs**:
  - For every test case output a single line containing the word `yes`, if the soldiers can be moved such that during the move enough military presence is displayed along every path and after moving every location is well defended, and the word `no` otherwise.

Mathematical Concept & Algorithm

- This problem can be formulated as a max flow problem (Circulation Problem)
  - Connect source to every supply (all the locations) with capacity g
  - Connect every demand (all the locations) to target with capacity d
  - Connect the locations according to the path information
- Minimum edge constraints
  - Assume all the minimum constraints are fulfilled: c soldiers are moved from u to v
  - Simulate the procedure:
    - Generate a flow from sink to source
    - increase the demand of the supply node by c
    - increase the supply of the demand node by c
- Check whether the max flow equals to the demand

Implementation & Testing

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
using namespace std;
// Graph Type with nested interior edge properties for flow algorithms
typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> traits;
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
    boost::property<boost::edge_capacity_t, long,
        boost::property<boost::edge_residual_capacity_t, long,
            boost::property<boost::edge_reverse_t, traits::edge_descriptor > > > > graph;

typedef traits::vertex_descriptor vertex_desc;
typedef traits::edge_descriptor edge_desc;
using namespace std;

// Custom edge adder class, highly recommended
class edge_adder {
    graph &G;
  public:
```

```
    explicit edge_adder(graph &G) : G(G) {}
    void add_edge(int from, int to, long capacity) {
        auto c_map = boost::get(boost::edge_capacity, G);
        auto r_map = boost::get(boost::edge_reverse, G);
        const auto e = boost::add_edge(from, to, G).first;
        const auto rev_e = boost::add_edge(to, from, G).first;
        c_map[e] = capacity;
        c_map[rev_e] = 0; // reverse edge has no capacity!
        r_map[e] = rev_e;
        r_map[rev_e] = e;
    }
};

int t, l, p;
void solve(){
    cin >> l >> p;
    graph G(l);
    const vertex_desc v_src = boost::add_vertex(G);
    const vertex_desc v_tar = boost::add_vertex(G);
    edge_adder adder(G);
    vector< pair<int, int> > node_info(l);
    for(int i = 0; i < l; i++) cin >> node_info[i].first >> node_info[i].second;
    for(int i = 0; i < p; i++){
        int from, to, c, C; cin >> from >> to >> c >> C;
        adder.add_edge(from, to, C-c);
        node_info[from].second += c;
        node_info[to].first  += c;
    }
    long demands_sum = 0;
    for(int i = 0; i < l; i++){
        adder.add_edge(v_src, i, node_info[i].first);
        adder.add_edge(i, v_tar, node_info[i].second);
        demands_sum += node_info[i].second;
    }
    long flow = boost::push_relabel_max_flow(G, v_src, v_tar);
    cout << (flow==demands_sum? "yes\n":"no\n");
}


int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin >> t;
    while(t--){
        solve();
    }
    return 0;
}
```