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

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

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

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

 // to use the Function (add an edge)
 void addEdge(int from, int to, long capacity) {
  Edge e, rev_e;
  bool success;
  boost::tie(e, success) = boost::add_edge(from, to, G);
  boost::tie(rev_e, success) = boost::add_edge(to, from, G);
  capacitymap[e] = capacity;
  capacitymap[rev_e] = 0; // reverse edge has no capacity!
  revedgemap[e] = rev_e;
  revedgemap[rev_e] = e;
 }

};

// Functions // ========= // Function for an individual testcase void testcases() { int num_locations; std::cin >> num_locations; int num_paths; std::cin >> num_paths; std::vector soldiers_here(num_locations); std::vector soldiers_needed(num_locations);

 // Create Graph and Maps
 Graph G(num_locations + 2);
 EdgeCapacityMap capacitymap = boost::get(boost::edge_capacity, G);
```

```
  ReverseEdgeMap revedgemap = boost::get(boost::edge_reverse, G);
  ResidualCapacityMap rescapacitymap = boost::get(boost::edge_residual_capacity, G);
  EdgeAdder eaG(G, capacitymap, revedgemap);
  Vertex source = num_locations;
  Vertex target = num_locations + 1;
  long total_soldiers_needed = 0;


  for(int i = 0; i < num_locations; i++) {
     std::cin >> soldiers_here.at(i);
     std::cin >> soldiers_needed.at(i);
     eaG.addEdge(source, i, soldiers_here.at(i));
     eaG.addEdge(i, target, soldiers_needed.at(i));
     total_soldiers_needed += soldiers_needed.at(i);
  }
  for(int i = 0; i < num_paths; i++) {
     int from; std::cin >> from;
     int to; std::cin >> to;
     int min; std::cin >> min;
     int max; std::cin >> max;
     // add edges
     eaG.addEdge(from, to, max - min);
   eaG.addEdge(source, to, min);
   eaG.addEdge(from, target, min);
   total_soldiers_needed += min;

  }

  // Calculate flow
  long flow1 = boost::push_relabel_max_flow(G, source, target);
  if(total_soldiers_needed == flow1) std::cout << "yes" << std::endl; else std::cout << "no" << std::endl;
```

```
}

// Main function to loop over the testcases int main() {
std::ios_base::sync_with_stdio(false); int T; std::cin >> T; for (; T >
0; --T) testcases(); return 0; }
```