```
// INDEX
// BIT ........................................ pág.   1
// DISJOINT-SET DATA STRUCTURE ................. pág.   1
// BINARY HEAP ................................ pág.   2
// SEGMENT TREE ............................... pág.   3
// BINOMIAL COEFFICIENT ....................... pág.   5
// LIS NLogN .................................. pág.   5
// GEOMETRY ................................... pág.   5
// BELLMAN FORD ............................... pág.   9
// DIJKSTRA ELogV ............................. pág.  10
// DIJKSTRA V^2 ............................... pág.  11
// FLOYD WARSHALL ............................. pág.  11
// FORD-FULKERSON ............................. pág.  12
// MINCOST-MAXFLOW ............................ pág.  14
// BITS & SETS ................................ pág.  15
// NUMBER THEORY .............................. pág.  16
// SQUARE ROOT ................................ pág.  18
// UTILITIES .................................. pág.  18
// 2-SAT AND STRONGLY CONNECTED COMPONENTS....... pág.  19
// GEOMETRY ON JAVA ........................... pág.  21
// SEGMENT TREE FOR SQUARES AREA ............... pág.  23

// Implementation of a Binary Indexed Tree.
// It supports these two operations:
// Modify the value of one element of the array in O(log n).
// Query the sum in a range [a, b] in O(log n).
const int MAX_BIT_SIZE = 1024;
template <class T>
class BIT {
 public:
  BIT(int n, T z) :  size(n), zero(z) {
    for (int i = 0; i < n; ++i) bit_array[i] = zero;
  }
  ~BIT() {}

  T QueryBIT(int a, int b) {
    if (a == 0) {
      T sum = zero;
      for (; b >= 0; b = (b & (b + 1)) - 1) sum += bit_array[b];
      return sum;
    } else return QueryBIT(0, b) - QueryBIT(0, a - 1);
  }

  void IncreaseBIT(int k, T inc) {
    for (; k < size; k |= k + 1) bit_array[k] += inc;
  }

 private:
  int size;
  T bit_array[MAX_BIT_SIZE];
  T zero;
};

// Disjoint-Set Data Structure with a forest using union
// by rank and path compression.  With these, the amortized
// time per operation is only O(a(n)) where a(n) is the
// inverse of the function A(n, n), and A is the extremely
// quickly-growing Ackermann function.  Since a(n) is its
// inverse, it's less than 5 for all remotely practical
// values of n.
const int DISJOINT_SIZE = 1000010;
int parent[DISJOINT_SIZE];
int rank[DISJOINT_SIZE];

void Initialize() {
  for (int i = 0; i < DISJOINT_SIZE; ++i) {
    parent[i] = i;
```

```
    rank[i] = 0;
  }
}

// Returns the representative for the set that x is in.
int Find(int x) {
  if (parent[x] == x) return x;
  return parent[x] = Find(parent[x]);
}

// Merge the sets that contain x and y.
void Merge(int x, int y) {
  int x_root = Find(x);
  int y_root = Find(y);
  if (rank[x_root] > rank[y_root]) parent[y_root] = x_root;
  else if (rank[x_root] < rank[y_root]) parent[x_root] = y_root;
  else if (x_root != y_root) {
    parent[y_root] = x_root;
    ++rank[x_root];
  }
}
// End of the Disjoint-Set Data Structure.

// Implementation of a binary heap.  Usually for Dijkstra in
// O(ElogV).

const int MAX_HEAP_SIZE = 1024;
template <class T>
class Heap {
 public:
  Heap() :  size(0) {
    for (int i = 0; i < MAX_HEAP_SIZE; ++i) qp[i] = -1;
  }
```

```
~Heap() {}

void Swim(int e) {
  for (int i = qp[e], j = (i - 1) / 2, t; v[pq[i]] < v[pq[j]];
       i = j, j = (i - 1) / 2) {
    t = pq[i]; pq[i] = pq[j]; pq[j] = t;
    qp[pq[i]] = i; qp[pq[j]] = j;
  }
}

void Sink(int e) {
  for (int i = qp[e], j = 2 * i + 1, t; j < size;
       i = j, j = 2 * i + 1) {
    if (j + 1 < size && v[pq[j + 1]] < v[pq[j]]) ++j;
    if (v[pq[j]] >= v[pq[i]]) break;
    t = pq[i]; pq[i] = pq[j]; pq[j] = t;
    qp[pq[i]] = i; qp[pq[j]] = j;
  }
}

void DeleteElement(int e) {
  int ori = qp[e];
  pq[ori] = pq[--size]; qp[e] = -1;
  if (size) qp[pq[ori]] = ori;
  Sink(pq[ori]);
}

void InsertOrModify(int e, T p) {
  bool decrease = false;
  if (qp[e] < 0 || v[e] > p) decrease = true;
  if (qp[e] < 0) qp[pq[size] = e] = size++;
  v[e] = p;
  if (decrease) Swim(e); else Sink(e);
}
```

```
  int Top() { return pq[0]; }

  bool Empty() { return size == 0; }

  int GetSize() { return size; }

  T GetValue(int e) { return v[e]; }

  int Pop() {
    int ret = pq[0];
    DeleteElement(ret);
    return ret;
  }

 private:
  T v[MAX_HEAP_SIZE];
  int pq[MAX_HEAP_SIZE];
  int qp[MAX_HEAP_SIZE];
  int size;
};

//Testing
int main() {
  Heap <int> heap;
  for (int i = 0; i < 11; ++i) heap.InsertOrModify(i, 10 - i);
  while (!heap.Empty()) cout << heap.Pop() << " ";
  cout << endl;
  for (int i = 0; i < 11; ++i) heap.InsertOrModify(i, 10 - i);
  heap.InsertOrModify(0, 0);
  heap.InsertOrModify(10, 8);
  while (!heap.Empty()) cout << heap.Pop() << " ";
  cout << endl;
  heap.InsertOrModify(0, 10);
  heap.InsertOrModify(0, 5);
  cout << heap.GetValue(heap.Top()) << endl;
  heap.InsertOrModify(0, 8);
  cout << heap.GetValue(heap.Top()) << endl;
}

// //////////////////
// // SEGMENT TREE //
// //////////////////

const int MAX_SIZE = 100000;
template <class T>
class SegmentTree {
 public:
  SegmentTree(T array[], int n) :  elements(n) {
    InitializeSegmentTree(0, 0, elements - 1, array);
  }

  SegmentTree(int n) :  elements(n) {}

  ~SegmentTree() {}

  void AddElement(int or_index, T val, int index = 0,
        int left = 0, int right = -1) {
    if (right == -1) right = elements - 1;
    if (left == right) {
      tree[index] += val; return;
    }
    int mid = (left + right) / 2;
    int lnode = 2 * index + 1, rnode = lnode + 1;
    if (or_index >= left && or_index <= mid)
      AddElement(or_index, val, lnode, left, mid);
    else
      AddElement(or_index, val, rnode, mid + 1, right);
```

```
    tree[index] = tree[lnode] + tree[rnode];
  }

  void AddElements(int a_left, int a_right, T val,
        int index = 0, int left = 0, int right = -1) {
    if (right == -1) right = elements - 1;
    if (a_left > right || a_right < left) return;
    if (left >= a_left && right <= a_right) {
      tree[index] += T (right - left + 1) * val;
      add[index] += val; return;
    }
    int mid = (left + right) / 2;
    int lnode = 2 * index + 1, rnode = lnode + 1;
    AddElements(a_left, a_right, val, lnode, left, mid);
    AddElements(a_left, a_right, val, rnode, mid + 1, right);
    tree[index] = tree[lnode] + tree[rnode] +
      T (right - left + 1) * add[index];
  }

  T Query(int q_left, int q_right, int index = 0, int left = 0,
          int right = -1) {
    if (right == -1) right = elements - 1;
    if (q_left > right || q_right < left) return 0;
    if (left >= q_left && right <= q_right) {
      T ret = tree[index]; if (!index) return ret;
      int parent = (index - 1) / 2;
      while (parent >= 0) {
        ret += T (right - left + 1) * add[parent];
        if (!parent) break;
        parent = (parent - 1) / 2;
      }
      return ret;
    }
    int mid = (left + right) / 2;
    int lnode = 2 * index + 1, rnode = lnode + 1;
    T r1 = Query(q_left, q_right, lnode, left, mid);
    T r2 = Query(q_left, q_right, rnode, mid + 1, right);
    return r1 + r2;
  }

 private:
  void InitializeSegmentTree(int index, int left, int right,
        T array[]) {
    if (left == right) {
      tree[index] = array[left]; return;
    }
    int mid = (left + right) / 2;
    int lnode = 2 * index + 1, rnode = lnode + 1;
    InitializeSegmentTree(lnode, left, mid, array);
    InitializeSegmentTree(rnode, mid + 1, right, array);
    tree[index] = tree[lnode] + tree[rnode];
  }
  // An upper bound for the capacity given by
  // 2 * 2 ^ ([logn] + 1)
  T tree[4 * MAX_SIZE];
  // Extra tree that contains the additions that should be made
  // to the given node and all of its children.
  T add[4 * MAX_SIZE];
  int elements;
};

int main() {
  SegmentTree <long long> *tree;
  tree = new SegmentTree <long long>(10);
  for (int i = 0; i < 10; ++i) tree->AddElement(i, 5);
  tree->AddElements(0, 9, 5);
  cout << tree->Query(0, 9);
}
```

```
// DYNAMIC PROGRAMMING

// The Binomial coefficient, very useful for DP and
// combinatorics problems.

const int TAM = 30;
long long nCr[TAM][TAM];

void CalcChoose() {
  memset(nCr, 0, sizeof(nCr));
  for (int i = 0; i < TAM; ++i) {
    nCr[i][0] = nCr[i][i] = 1;
    for (int j = 1; j < i; ++j)
       nCr[i][j] = nCr[i-1][j-1] + nCr[i-1][j];
  }
}

// Longest Increasing Subsequence in NLogN.

public class LIS {
  public static int LongestIncSubsequenceNlogN(int[] array) {
    int[] A = new int[array.length + 1];
    Arrays.fill(A, Integer.MAX_VALUE);
    int longest = 0;
    for (int i = 0; i < array.length; ++i) {
      int low = 0;
      int high = array.length - 1;
      while (low < high) {
        int mid = low + (high - low + 1) / 2;
        // If it is not strict, change >= with >
        if (A[mid] >= array[i]) {
          high = mid - 1;
        } else {
          low = mid;
        }
      }
      if (A[low + 1] > array[i]) {
        A[low + 1] = array[i];
        longest = (low + 1 > longest) ?  low + 1 :  longest;
      }
    }
    return longest;
  }
}

// GEOMETRY

// Comparison function required to compare points with double
// coordinates.
const double EPS = 1e-10;
inline int cmp(double x, double y = 0, double tol = EPS) {
  return (x <= y + tol) ?  (x + tol < y) ?  -1 :  0 :  1;
}
// Useful geometry utilities.
// Point class with operator overloading.
class Point {
 public:
  Point(double x_ = 0.0, double y_ = 0.0) :  x(x_), y(y_) {}
  Point operator +(const Point &o) const {
    return Point(x + o.x, y + o.y); }
  Point operator -(const Point &o) const {
    return Point(x - o.x, y - o.y);
  }
  Point operator *(const double &m) const {
    return Point(m * x, m * y); }
  Point operator /(const double &m) const {
    return Point(x / m, y / m); }
```

```
// Dot Product
double operator *(const Point &o) const {
  return x * o.x + y * o.y; }
// Cross Product
double operator ^(const Point &o) const {
  return x * o.y - y * o.x; }

int cmp(Point o) const {
  if (int t = ::cmp(x, o.x)) return t;
  return ::cmp(y, o.y);
}
bool operator ==(const Point &o) const {
  return cmp(o) == 0; }
bool operator !=(const Point &o) const {
  return cmp(o) != 0; }
bool operator < (const Point &o) const {
  return cmp(o) < 0; }

// Euclidean distance between two points.
double Distance(const Point &o) const {
  double d1 = x - o.x, d2 = y - o.y;
  return sqrt(d1 * d1 + d2 * d2);
}

// Calculates the distance between the point and the line
// specified by the two points given; if isSegment is true,
// we treat those two points are the endpoints of a segment.
double Distance(const Point &p1, const Point &p2,
      const bool &isSegment) const {
  double dist = ((p2 - p1) ^ (*this - p1)) / p2.Distance(p1);
  if (isSegment) {
    double dot1 = (*this - p2) * (p2 - p1);
    if (::cmp(dot1) > 0)
      return sqrt((p2 - *this) * (p2 - *this));
    double dot2 = (*this - p1) * (p1 - p2);
    if (::cmp(dot2) > 0)
      return sqrt((p1 - *this) * (p1 - *this));
  }
  return abs(dist);
}

friend ostream& operator <<(ostream &o, Point p) {
  return o << "(" << p.x << ", " << p.y << ")";
}

double x, y;
static Point pivot;
};

Point Point::pivot(0, 0);

// Calculates the angle between the two vectors defined by
// p - r and q - r.  The formula comes from the definition of
// the dot and the cross product:
//
// A . B = |A||B|cos(c)
// A x B = |A||B|sin(c)
//
// sin(c)   A x B
// ------ = ------ = tan(c)
// cos(c)   A . B
inline double angle(const Point &p, const Point &q,
                    const Point &r) {
  Point u = p - r, v = q - r;
  return atan2(u ^ v, u * v);
}

// Calculates sign of the turn between the two vectors defined
```

```
// by <p-r> and <q-r>.
//
// Just to remember, the cross product is defined by (x1 * y2) -
// (x2 * y1) and is negative if it is a right turn and positive
// if it is a left turn.  e.g.
//
//            .p3
//            ^
//           /
//      .p2 /
//      ^  /
//      | /
//      .p1
// The cross product between the vectors <p2-p1> and <p3-p1>
// is negative, that means it is a right turn.
inline int turn(const Point &p, const Point &q,
                const Point &r) {
  return ::cmp((p - r) ^ (q - r));
}

// Decides if the point r is inside the segment defined by the
// points p and q.  To do this, we have to check two conditions:
// 1.  That the turn between the two vectors formed by p - q and
// r - q is zero (that means they are parallel).
// 2.  That the dot product between the vector formed by p - r
// and q - r (that means the testing point as the initial point
// for both vectors) is less than or equal to zero (that means
// that the two vectors have opposite direction).
inline bool between(const Point &p, const Point &q,
                    const Point &r){
  return turn(p, r, q) == 0 && ::cmp((p - r) * (q - r)) <= 0;
}

// Returns 0, -1 or 1 depending if p is in the exterior, the
// frontier or the interior of the given polygon respectively,
// the polygon must be in clockwise or counterclockwise
// order [MANDATORY!!].  The idea is to iterate over each of
// the points in the polygon and consider the segment formed by
// two adyacent points, if the test points is inside that
// segment, the point is in the frontier, if not, we add the
// angles inside the vectors formed by the two points of the
// polygon and the test point.  For a point outside the
// polygon this sum is zero because the angles cancel
// themselves.
int InPolygon(const Point &p, const vector<Point> &T) {
  double a = 0; int N = T.size();
  for (int i = 0; i < N; ++i) {
    if (between(T[i], T[(i + 1) % N], p)) return -1;
    a += angle(T[i], T[(i + 1) % N], p);
  }
  return ::cmp(a) != 0;
}

// Comparator to be used in the sorting for the convex hull.
// We sort the points based on the cross product between them
// (the direction of the turn) and if the are colinear, we order
// them based on their distances to the origin.  At the end,
// the comparison based on the direction of the turn is the same
// as the comparison based on the angles.
bool RadialComp(const Point &p, const Point &q) {
  Point P = p - Point::pivot, Q = q - Point::pivot;
  double R = P ^ Q;
  if (::cmp(R)) return R > 0;
  return ::cmp(P * P, Q * Q) < 0;
}

// Returns the number of the quadrant that the point is in.  The
// point (0,0) is clasified as in the fifth quadrant because it
```

```
// really doesn't belong to any.
int Quadrant(const Point &p) {
  if (::cmp(p.x) == 0 && ::cmp(p.y) == 0) return 5;
  if (::cmp(p.y) == 1) {
    if (::cmp(p.x) == 1) return 1;
    return 2;
  }
  if (::cmp(p.y) == 0) {
    if (::cmp(p.x) == 1 || ::cmp(p.x) == 0) return 1;
    return 3;
  }
  if (::cmp(p.x) == -1) return 3;
  return 4;
}

// Comparator to sort the points by their angle without
// calculating their angles it is different from the
// RadialComp because the cross product only works if the two
// points to be compared are in two contiguous quadrants.  The
// idea of the comparator is to find out the quadrant first
// and if it is the same, then it calculates the cross product.
bool PolarComp(const Point &p, const Point &q) {
  Point P = p - Point::pivot, Q = q - Point::pivot;
  int q1 = Quadrant(P), q2 = Quadrant(Q);
  if (q1 != q2) return q1 < q2;
  double R = P ^ Q;
  if (::cmp(R)) return R > 0;
  return ::cmp(P * P, Q * Q) < 0;
}

// Convex Hull, The vector of points can't be passed by
// reference since we manipulate it here and it gets changed.
//
// We set the pivot as the minimum point in the polygon (that in
// the cmp function for the point is the point with the lowest
// x coordinate and in case of tie with the lowest y
// coordinate).  We then sort the points with the radial
// comparator based on the pivot point.
//
// But there is a problem with the points with the same x
// coordinate than the pivot point, they are ordered in
// ascending order of y coordinate, so we first find those
// points and reverse them.
//
// The final step is to consider each point of the polygon,
// add it to the convex hull and check if that addition implies
// or not an exclusion of the previously added points
// aplying the cross product (to see if it is a counterclockwise
// (include) or a clockwise (exclude) turn).
vector <Point> ConvexHull(vector <Point> T) {
  int j = 0, k, n = T.size(); vector <Point> U(n);
  Point::pivot = *min_element(T.begin(), T.end());
  sort(T.begin(), T.end(), RadialComp);
  for (k = n - 2; k >= 0 && turn(T[0], T[n - 1], T[k]) == 0;
       --k);
  reverse((k + 1) + T.begin(), T.end());
  for (int i = 0; i < n; ++i) {
    // Change >= for > to keep the colinear points.
    while (j > 1 && turn(U[j - 1], U[j - 2], T[i]) > 0) --j;
    U[j++] = T[i];
  }
  U.erase(j + U.begin(), U.end());
  return U;
}

// Calculates the area of the given Polygon, it has to be given
// in clockwise or counterclockwise order [MANDATORY!].  The
// idea is to triangulate the polygon based on an initial point;
```

```
// the cross product will be of the opposite sign if the area of
// the triangle must be added or substracted to the total area
// of the polygon.  At the end, the number can be negative
// if the points were given in clockwise order so we return
// the absolute value.  As we know that the cross product is
// the area of the paralelogram formed by the vectors, then,
// the area of the triangle is the cross product divided by
// two.
double Area(const vector<Point> &T) {
  double area = 0.0;
  //We will triangulate the polygon
  //into triangles with points p[0],p[i],p[i+1]
  for(int i = 1; i + 1 < T.size(); i++){
    area += (T[i] - T[0]) ^ (T[i + 1] - T[0]);
  }
  return abs(area / 2.0);
}

// Returns the point of intersection between the lines defined
// by p, q and r, s.
Point Intersection(const Point &p, const Point &q,
      const Point &r, const Point &s) {
  Point a = q - p, b = s - r, c = Point(p ^ q, r ^ s);
  return Point(Point(a.x, b.x) ^ c,
             Point(a.y, b.y) ^ c) / (a ^ b);
}

// GRAPHS

// //////////////////////////////////
// // BELLMAN FORD'S SHORTEST PATH //
// //////////////////////////////////
// Takes a directed graph where each edge has a weight and
// returns the shortest path from s to any other vertex.  This
// algorithm has the quality that works even if the graph has
// negative cicles (it detects them).  Is useful for sparse
// graphs due to its complexity; for dense graphs better use
// Floyd-Warshall that gives more information for the same cost.
//
// PARAMETERS:
// - deg (global):  (out-)degree of each vertex
// - adj (global):  Adjacency list.  For each u,
//                  adj[u][0..deg[u]] are the neighbours.
// - cost (global):  Costs list.  For each u,
//                   costs[u][0..deg[u]] is the cost of the edge
//                   between u and adj[u][0..deg(u)].
// - n (global):  The number of vertices ([0, n-1] are
//                considered as vertices).
// - INF (global):  As its name.
// - s:  source vertex.
// RETURNS:
// - d[] contains the minimum path from s to any other vertex.
// - prev[] contains the path predecessors.
// - neg is true if the graph has a negative cycle.
// COMPLEXITY:
// - Slow.  O(V * E).
// REQUIRES:
// - vector
// FIELD TESTING:
//

using namespace std;

const int INF = 0X3F3F3F3F;
const int MAX_NODES = 1000;
vector <int> adj[MAX_NODES];
vector <int> cost[MAX_NODES];
int deg[MAX_NODES];
```

```
int d[MAX_NODES];
int prev[MAX_NODES];
int n;
bool neg;

void BellmanFord(int s) {
  for (int i = 0; i < n; ++i) { d[i] = INF; prev[i] = -1; }
  d[s] = 0; neg = false;
  for (int i = 0; i < n + 1; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < deg[j]; ++k) {
        if (d[adj[j][k]] > d[j] + cost[j][k]) {
          if (i >= n) {
            neg = true;
          }
          prev[adj[j][k]] = j;
          d[adj[j][k]] = d[j] + cost[j][k];
        }
      }
    }
  }
}

// //////////////////////////////
// // DIJKSTRA'S SHORTEST PATH //
// //////////////////////////////
//
// Takes a directed graph where each edge has a weight and
// returns the shortest path from s to any other vertex.
//
// PARAMETERS:
// - deg (global):  (out-)degree of each vertex
// - adj (global):  Adjacency list.  For each u,
// adj[u][0..deg[u]] are the neighbours.
// - cost (global):  Costs list.  For each u,
// costs[u][0..deg[u]] is the cost of the edge between u
// and adj[u][0..deg(u)].
// - n (global):  The number of vertices ([0, n-1] are
// considered as vertices).
// - INF (global):  As its name.
// - s:  source vertex.
// RETURNS:
// - d[] contains the minimum path from s to any other vertex.
// - prev[] contains the path predecessors.
// COMPLEXITY:
// - Fast.  O(E * log(V)).
// REQUIRES:
// - vector
// FIELD TESTING:
//
const int INF = 0X3F3F3F3F;
const int MAX_NODES = 1000;
vector <int> adj[MAX_NODES];
vector <int> cost[MAX_NODES];
int deg[MAX_NODES];
int d[MAX_NODES];
int prev[MAX_NODES];
int n;
void Dijkstra(int s) {
  for (int i = 0; i < n; ++i) { d[i] = INF; prev[i] = -1; }
  Heap <int> heap;
  d[s] = 0;
  heap.InsertOrModify(s, 0);
  while(!heap.Empty()) {
    int vertex = heap.Pop();
    int dist = heap.GetValue(vertex);
    for (int ii = 0; ii < deg[vertex]; ++ii) {
      int vertex2 = adj[vertex][ii], c = cost[vertex][ii];
```

```
      if(d[vertex2] > d[vertex] + c) {
        d[vertex2] = d[vertex] + c;
        prev[vertex2] = vertex;
        heap.InsertOrModify(vertex2, d[vertex2]);
      }
    }
  }
}

// //////////////////////////////
// // DIJKSTRA'S SHORTEST PATH //
// //////////////////////////////
//
// Takes a directed graph where each edge has a weight and
// returns the shortest path from s to any other vertex.  This
// version is better for very dense graphs since its complexity
// is lower than the Dijkstra with heap if E = V^2.
//
// PARAMETERS:
// - graph (global):  Adjacency matrix.  The value graph[i][j]
is
// the cost of the edge between the nodes i and j.
// - n (global):  The number of vertices ([0, n-1] are consid-
ered
// as vertices).
// - INF (global):  As its name.
// - s:  source vertex.
// RETURNS:
// - d[] contains the minimum path from s to any other vertex.
// - prev[] contains the path predecessors.
// COMPLEXITY:
// - Fast.  O(E + V^2).
// FIELD TESTING:
//
const int INF = 0X3F3F3F3F;
const int MAX_NODES = 1000;
int graph[MAX_NODES][MAX_NODES];
int d[MAX_NODES];
int prev[MAX_NODES];
int n;
void Dijkstra(int s) {
  bool can[MAX_NODES];
  for (int i = 0; i < n; ++i) {
    d[i] = INF; prev[i] = -1; can[i] = false;
  }
  d[s] = 0;
  can[s] = true;
  while (true) {
    int h = -1;
    for (int j = 0; j < n; ++j) if (can[j] && (h == -1 || d[j] <
d[h])) h = j;
    if (h == -1) break;
    can[h] = false;
    for (int ii = 0; ii < n; ++ii) {
      int c = graph[h][ii];
      if (c == INF) continue;
      if(d[ii] > d[h] + c) {
        d[ii] = d[h] + c;
        prev[ii] = h;
        can[ii] = true;
      }
    }
  }
}

// ////////////////////////////////////
// // FLOYD-WARSHALL'S SHORTEST PATH //
// ////////////////////////////////////
```

```
//
// Takes a directed graph where each edge has a weight and
// returns the shortest path between all pair for vertices.
// It is very useful if the graph is dense since will get
// more information than Bellman-Ford in the same time.
//
// PARAMETERS:
// - graph (global):  Adjacency matrix.  The value graph[i][j]
// is the cost of the edge between the nodes i and j.
// - n (global):  The number of vertices ([0, n-1] are
// considered as vertices).
// - INF (global):  As its name.
// - NO_EDGE (global):  The value that specifies that there is
// no connection between two vertex.
// RETURNS:
// - d[][] contains the minimum path between every pair of
// vertices.
// - prev[][] contains the path predecessors.
// COMPLEXITY:
// - Slow.  O(V ^ 3).
// REQUIRES:
// - vector

const int INF = 0X3F3F3F3F;
const int MAX_NODES = 1000;
int graph[MAX_NODES][MAX_NODES];
int d[MAX_NODES][MAX_NODES];
int prev[MAX_NODES][MAX_NODES];
int n;
vector <int> path;

void FloydWarshall() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      d[i][j] = graph[i][j]; prev[i][j] = i;
    }
    d[i][i] = 0;
  }
  for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        if (d[i][j] > d[i][k] + d[k][j]) {
          d[i][j] = d[i][k] + d[k][j];
          prev[i][j] = prev[k][j];
        }
      }
    }
  }
}

void ConstructFloydShortestPath(int s, int t) {
  if (s != t) ConstructFloydShortestPath(s, prev[s][t]);
  path.push_back(t);
}

// //////////////////////////////////////
// // FORD-FULKERSON'S MAXFLOW/MINCUT //
// //////////////////////////////////////
//
// Takes a directed graph where each edge has a weight
// (capacity) and returns the value of the maximum flow
// that can be sent from vertex s to vertex t.  This value
// is also the minimum sum of the weight of the edges
// that have to be removed in order to disconnect s and t.
//
// PARAMETERS:
// - capacity (global):  Adjacency matrix.  The value
// capacity[i][j] is the capacity of the edge between the
```

```
// nodes i and j.
// - n (global):  The number of vertices ([0, n-1] are
// considered as vertices).
// - INF (global):  As its name.
// - s:  source vertex.
// - t:  sink vertex.
// RETURNS:
// - max_flow[] contains the max_flow that can be send in
// a given step between the source node and every other node.
// - prev[] contains the path predecessors.
// COMPLEXITY:
// - Slow.  O(V * E ^ 2) As it always finds the shortest path
// (Edmons-Karp).
// REQUIRES:
// - vector
// FIELD TESTING:
// - A Plug for Unix (PKU #1087)
const int INF = 0X3F3F3F3F;
const int MAX_NODES = 500;
int capacity[MAX_NODES][MAX_NODES];
int prev[MAX_NODES];
int max_flow[MAX_NODES];
int n;
int PFS(int s, int t) {
  bool can[MAX_NODES];
  int max_capacity = 0;
  for (int i = 0; i < n; ++i) {
    max_flow[i] = 0; prev[i] = -1; can[i] = false;
  }
  max_flow[s] = INF;
  can[s] = true;
  while(true) {
    int h = -1;
    for (int j = 0; j < n; ++j)
      if (can[j] && (h == -1 || max_flow[j] > max_flow[h]))
        h = j;
    if (h == -1) break;
    can[h] = false;
    if (h == t) {
      max_capacity = max_flow[h];
      break;
    }
    for (int ii = 0; ii < n; ++ii) {
      int c = capacity[h][ii];
      if (c == 0) continue;
      if (min(max_flow[h], c) > max_flow[ii]) {
        max_flow[ii] = min(max_flow[h], c);
        prev[ii] = h;
        can[ii] = true;
      }
    }
  }
  return max_capacity;
}

int MaximumFlow(int s, int t) {
  int result = 0;
  while (true) {
    int cap = PFS(s, t);
    if (cap == 0) break;
    int where = t;
    while (where != s) {
      int last = prev[where];
      capacity[last][where] -= cap;
      capacity[where][last] += cap;
      where = last;
    }
    cout << endl;
```

```
    result += cap;
  }
  return result;
}

// ////////////////////////////////////
// // MINCOST-MAXFLOW //
// ////////////////////////////////////

const int INF = 0x3F3F3F3F;
const int MAX_NODES = 500;
int capacity[MAX_NODES][MAX_NODES];
int cost[MAX_NODES][MAX_NODES];
int d[MAX_NODES];
int prev[MAX_NODES];
int n;
int max_flow;
// This variation of Dijkstra works with negative edges.
// Its complexity is different but as long as we don't have
// a table, we can keep relaxing edges.
void Dijkstra(int s) {
  bool can[MAX_NODES];
  for (int i = 0; i < n; ++i) {
    d[i] = INF; prev[i] = -1; can[i] = false;
  }
  d[s] = 0;
  can[s] = true;
  while (true) {
    int h = -1;
    for (int j = 0; j < n; ++j)
      if (can[j] && (h == -1 || d[j] < d[h]))
        h = j;
    if (h == -1) break;
    can[h] = false;
    for (int ii = 0; ii < n; ++ii) {
      int c = cost[h][ii];
      if (capacity[h][ii] == 0) continue;
      if(d[ii] > d[h] + c) {
        d[ii] = d[h] + c;
        prev[ii] = h;
        can[ii] = true;
      }
    }
  }
}

int MinCostMaximumFlow(int s, int t) {
  int min_cost = 0;
  max_flow = 0;
  while (true) {
    Dijkstra(s);
    if (d[t] > INF / 2) break;
    int where = t, neck = INF;
    while (where != s) {
      int last = prev[where];
      neck = min(neck, capacity[last][where]);
      where = last;
    }
    where = t;
    while (where != s) {
      int last = prev[where];
      capacity[last][where] -= neck;
      capacity[where][last] += neck;
      min_cost += cost[last][where] * neck;
      where = last;
    }
    max_flow += neck;
  }
```

```
  return min_cost;
}

class RadarGuns {
 public:
  vector <int> getRange(vector <int> enterTimes,
      vector <int> exitTimes, int speedTime, int fineCap);
};

vector <int> RadarGuns::getRange(vector <int> enterTimes,
    vector <int> exitTimes, int speedTime, int fineCap) {
  n = enterTimes.size() * 2 + 2;
  int mid = enterTimes.size();
  // Always remember to set the cost[i][j] on the edges
  // that are in the graph and in the BACK EDGES of the
  // augmented graph.
  for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) {
    cost[i][j] = INF; capacity[i][j] = 0;
  }
  for (int i = 0; i < mid; ++i) {
    for (int j = 0; j < mid; ++j) {
      if (exitTimes[j] > enterTimes[i]) {
        capacity[i][mid + j] = 1;
        cost[i][mid + j] = 0;
        cost[mid + j][i] = 0;
        if (speedTime > exitTimes[j] - enterTimes[i]) {
          int x = speedTime - (exitTimes[j] - enterTimes[i]);
          cost[i][mid + j] = min(x * x, fineCap);
          // It is important!
          cost[mid + j][i] = -min(x * x, fineCap);
        }
      }
    }
  }
  for (int i = 0; i < mid; ++i) {
    capacity[mid + mid][i] = 1;
    cost[mid + mid][i] = 0;
    cost[i][mid + mid] = 0;
    capacity[i + mid][mid + mid + 1] = 1;
    cost[i + mid][mid + mid + 1] = 0;
    cost[mid + mid + 1][i + mid] = 0;
  }
  int temp[MAX_NODES][MAX_NODES];
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) temp[i][j] = capacity[i][j];
  vector <int> ret(2);
  ret[0] = MinCostMaximumFlow(mid + mid, mid + mid + 1);
  if (max_flow < mid) return vector <int>();
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) capacity[i][j] = temp[i][j];
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      if (cost[i][j] != INF) cost[i][j] = -cost[i][j];
  ret[1] = -MinCostMaximumFlow(mid + mid, mid + mid + 1);
  return ret;
}

// MATHEMATICS

// Bits & Sets

template<class T>
inline int CountBit(T n) {
  return (n == 0) ?  0 :  (1 + CountBit(n & (n - 1)));
}

template<class T>
```

```
inline bool IsSubset(T set, T subset) {
  return (set & subset) == subset;
}

// If you need the empty subset, change the subset > 0 to
// subset >= 0.
template<class T>
inline void IterateSubsets(T set) {
  for (int subset = set; subset > 0;
       subset = (subset - 1) & set) {
    // TODO
  }
}

template<class T>
inline bool Contains(T set, T bit) {
  return (set & (1 << bit)) != 0;
}

// Number Theory

// Utility type used by the Extended Euclidean Algorithm.
template <class T>
struct Triple {
  T d_, x_, y_;
  Triple(T d, T x, T y) :  d_(d), x_(x), y_(y) {}
};

// Euclidean algorithm.  Calculates the Greatest Common Divisor
// between two numbers.  This algorithm works on non-negative
// integers.
template <class T>
T GreatestCommonDivisor(T a, T b) {
  return (b == 0) ?  a :  GreatestCommonDivisor(b, a % b);
}

// Extended GCD. Given non-negative a and b, computes
// d = gcd(a, b) along with integers x and y, such that
// d = ax + by and returns the triple (d, x, y).
// REQUIRES: struct Triple.
template <class T>
Triple <T> ExtendedGCD(T a, T b) {
  if(!b) return Triple <T>(a, T(1), T(0));
  Triple <T> q = ExtendedGCD(b, a % b);
  return Triple <T>(q.d, q.y, q.x - a / b * q.y);
}

// Fast powering.  It calculates x ^ y with a complexity of
// O(log y).
// USED BY: ModularInverse.
template <class T>
T FastPow(T x, T y, T mod = T(0)) {
  if (!y) return T(1);
  T ret = FastPow(x, y/2);
  ret = (ret * ret);
  if (mod) ret %= mod;
  // If the power is odd we have to multiply one more time.
  if (y & 1) {
    ret = (ret * x);
    if (mod) ret %= mod;
  }
  return ret;
}

// Calculates the modular inverse of a number based on Fermat's
// theorem.  The theorem is as follows:  Suppose p is a prime
// and k is not a multiple of p.  Then k ^ (p - 1) ≡ 1 (mod p).
// Now, using this theorem, we know that k ^ (p - 2) * k ≡
```

```
// 1 (mod p), therefore k ^ (p - 2) is the modular inverse of k.
// REQUIRES: FastPow.
template <class T>
T ModularInverse(T x, T mod) {
  return FastPow(x, mod - 2) % mod;
}

// Templated inlined function to factorize a number (can be int,
// long long, etc.)  The idea is to find the prime factors
// bottom up.  This function has a couple of little speed
// improvements (after 3, the primes are separated by 6 and 8
// units, it is inlined, etc.).  This is the fastest we can do
// considering that it is a generalized function.
//
// If we know that a number is a perfect square, we can
// calculate the prime factors of the square root and multiply
// the occurrences of these factors by two (each prime factor
// occurs twice in the square of a number).
//
// BE CAREFUL: If the number to factorize is 1, then it will
// return an empty vector; this can backfire you.  For obvious
// reasons, 1 is not considered as a prime factor.
// USED BY: Phi.

template<class T>
inline void Squeeze(vector <pair <T, int> > &M, T &n, T p) {
  int C = 0;
  for (; n % p == 0; n /= p) ++C;
  if (C != 0) M.push_back(make_pair(p, C));
}

template<class T>
inline vector <pair <T, int> > Factorize(T n) {
  if (n < 0) return Factorize(-n);
  vector <pair <T, int> > M;
  if (n < 2) return M;
  Squeeze(M, n, (T) 2);
  Squeeze(M, n, (T) 3);
  T p = 5;
  while (n > 1) {
    Squeeze(M, n, p);
    Squeeze(M, n, p + 2);
    p += 6;
    if (p * p > n) p = n;
  }
  return M;
}

// Euler's Totient Function.
// REQUIRES: Factorize.
int Phi(int n) {
  vector <pair <int, int> > p;
  p = Factorize(n);
  for(int i = 0; i < (int)p.size(); i++){
    n /= p[i].first;
    n *= p[i].first - 1;
  }
  return n;
}

int main() {
  cout << "Fast Pow testing..." << endl;
  cout << "10 ^ 10 = " << FastPow<long long>(10, 10) << endl;
  cout << "Modular Inverse testing..." << endl;
  cout << "The inverse of 4 mod 5 is " << ModularInverse(4, 5);
  cout << "Euler Totient Function testing..." << endl;
  cout << "Phi(89) = " << Phi(89);
}
```

```
// Square Root

import java.math.BigInteger;
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;

public class SquareRoot {
  // Pell's Algorithm.
  static BigInteger BigSqrt(String n) {
    BigInteger HUNDRED = new BigInteger("100");
    BigInteger TWO = new BigInteger("2");
    BigInteger TWENTY = new BigInteger("20");
    BigInteger TEN = new BigInteger("10");
    List <String> parts = new ArrayList <String>();
    for (int i = n.length(); i > 0; i -= 2)
      parts.add(n.substring(Math.max(i - 2, 0), i));
    BigInteger odd = BigInteger.ZERO;
    BigInteger remain = BigInteger.ZERO;
    BigInteger answer = BigInteger.ZERO;
    for (int i = parts.size() - 1; i >= 0; --i) {
      odd = TWENTY.multiply(answer).add(BigInteger.ONE);
      remain = HUNDRED.multiply(remain).add(
               new BigInteger(parts.get(i)));
      BigInteger count = BigInteger.ZERO;
      while (remain.compareTo(odd) >= 0) {
        count = count.add(BigInteger.ONE);
        remain = remain.subtract(odd);
        odd = odd.add(TWO);
      }
      answer = TEN.multiply(answer).add(count);
    }
    return answer;
  }

  private static BigInteger NewtonSquareRoot(BigInteger n) {
    BigInteger act = BigInteger.ONE;
    BigInteger last = BigInteger.ZERO;
    BigInteger TWO = new BigInteger("2");
    while (act.compareTo(last) != 0) {
      last = act;
      act = act.add(n.divide(act)).divide(TWO);
    }
    return act;
  }
}
```

```
// UTILITIES

// Function to compare two floating point numbers (you can
// specify the epsilon).  The base function is used by the other
// two utility functions, ALWAYS use this NEVER direct
// comparison.  They are all inline to save the overhead of
// calling a function.

const double EPS = 1e-10;

inline int cmp(double x, double y = 0, double tol = EPS) {
  return (x <= y + tol) ?  (x + tol < y) ?  -1 :  0 :  1;
}

inline bool cmp_eq(double x, double y) {
  return cmp(x, y) == 0;
}

inline bool cmp_lt(double x, double y) {
```

```
  return cmp(x, y) < 0;
}

inline bool cmp_lteq(double x, double y) {
  return cmp(x, y) <= 0;
}

// Merge-Sort

long long swaps;
long long array[500000];
long long temp[500000];

void MergeSort(int b, int e) {
  if (e - b == 0) return;
  if (e - b == 1) {
    if (array[e] < array[b]) {
      ++swaps;
      swap(array[e], array[b]);
    }
    return;
  }
  int m = (e + b) / 2;
  MergeSort(b, m);
  MergeSort(m + 1, e);
  int i = b, j = m + 1, t = b;
  while (i <= m && j <= e) {
    if (array[i] < array[j]) {
      temp[t] = array[i];
      ++t; ++i;
    } else {
      swaps += static_cast<long long>(m - i + 1);
      temp[t] = array[j];
      ++t; ++j;
    }
  }
  while (i <= m) temp[t++] = array[i++];
  while (j <= e) temp[t++] = array[j++];
  for (int k = b; k <= e; ++k) {
    array[k] = temp[k];
  }
  return;
}

public class XMart {
  // (a or b) is equivalent to (~a -> b) and (~b -> a).
  // For node i, its negation is node i + 1.
  //
  // p q p -> q
  // 1 1 1
  // 1 0 0
  // 0 1 1
  // 0 0 1
  private static List<Integer>[] graph;
  private static boolean isPossible;
  private static int index;
  private static Stack<Integer> stack;
  private static int[] indexes;
  private static int[] lowLink;
  private static boolean[] inStack;
  private static boolean[] marked;

  private static void TarjanFor2SAT() {
    stack = new Stack<Integer>();
    index = 0;
    indexes = new int[graph.length];
    lowLink = new int[graph.length];
    inStack = new boolean[graph.length];
```

```
    marked = new boolean[graph.length];
    for (int i = 0; i < graph.length; ++i) {
      indexes[i] = lowLink[i] = -1;
    }
    for (int i = 0; i < graph.length; ++i) {
      if (marked[i]) continue;
      TarjanDFS(i);
    }
  }

  private static void TarjanDFS(int node) {
    indexes[node] = index;
    lowLink[node] = index;
    ++index;
    stack.push(node);
    inStack[node] = true;
    marked[node] = true;
    for (Integer suc :  graph[node]) {
      if (indexes[suc] == -1) {
        TarjanDFS(suc);
        lowLink[node] = (lowLink[suc] < lowLink[node]) ?
            lowLink[suc] :  lowLink[node];
      } else if (inStack[suc]) {
        lowLink[node] = (lowLink[suc] < lowLink[node]) ?
            lowLink[suc] :  lowLink[node];
      }
    }
    if (lowLink[node] == indexes[node]) {
      // We found the head of a SCC...
      boolean[] SCC = new boolean[graph.length];
      int act = stack.pop();
      inStack[act] = false;
      while (true) {
        SCC[act] = true;
        if ((act & 1) == 0) {
          if (SCC[act + 1]) isPossible = false;
        } else {
          if (SCC[act - 1]) isPossible = false;
        }
        if (act == node) break;
        act = stack.pop();
        inStack[act] = false;
      }
    }
  }

  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws IOException {
    //BufferedReader reader = new BufferedReader(
                          //new InputStreamReader(System.in));
    BufferedReader reader = new BufferedReader(
                          new FileReader("xmart.in"));
    int C, P;
    String[] parts = reader.readLine().split("[ ]+");
    C = Integer.parseInt(parts[0]);
    P = Integer.parseInt(parts[1]);
    while (C!= 0 && P != 0) {
      graph = (List<Integer>[]) new List[2 * P];
      for (int i = 0; i < graph.length; ++i)
        graph[i] = new ArrayList<Integer>();
      for (int i = 0; i < C; ++i) {
        parts = reader.readLine().split("[ ]+");
        int p1, p2, p3, p4;
        p1 = Integer.parseInt(parts[0]) - 1;
        p2 = Integer.parseInt(parts[1]) - 1;
        p3 = Integer.parseInt(parts[2]) - 1;
        p4 = Integer.parseInt(parts[3]) - 1;
        if (p1 != -1 && p2 != -1) {
```

```java
          graph[2 * p1 + 1].add(2 * p2);
          graph[2 * p2 + 1].add(2 * p1);
        } else if (p1 == -1 && p2 != -1) {
          graph[2 * p2 + 1].add(2 * p2);
        } else if (p2 == -1 && p1 != -1) {
          graph[2 * p1 + 1].add(2 * p1);
        }
        if (p3 != -1 && p4 != -1) {
          graph[2 * p3].add(2 * p4 + 1);
          graph[2 * p4].add(2 * p3 + 1);
        } else if (p3 == -1 && p4 != -1) {
          graph[2 * p4].add(2 * p4 + 1);
        } else if (p4 == -1 && p3 != -1) {
          graph[2 * p3].add(2 * p3 + 1);
        }
      }
      isPossible = true;
      TarjanFor2SAT();
      if (isPossible) {
        System.out.println("yes");
      } else {
        System.out.println("no");
      }
      parts = reader.readLine().split("[ ]+");
      C = Integer.parseInt(parts[0]);
      P = Integer.parseInt(parts[1]);
    }
  }
}
```

```java
import java.awt.geom.Line2D;
public class Geometry {
  private static final double EPS = 1e-10;
  private static int cmp(double x, double y) {
    return (x <= y + EPS) ? (x + EPS < y) ?  -1 :  0 :  1;
  }
  // Immutable Point Class.
  private static class Point implements Comparable<Point> {
    public double x;
    public double y;
    public Point(double x, double y) {
      this.x = x;
      this.y = y;
    }
    public Point() {
      this.x = 0.0;
      this.y = 0.0;
    }
    public double dotProduct(Point o) {
      return this.x * o.x + this.y * o.y;
    }
    public double crossProduct(Point o) {
      return this.x * o.y - this.y * o.x;
    }
    public Point add(Point o) {
      return new Point(this.x + o.x, this.y + o.y);
    }
    public Point substract(Point o) {
      return new Point(this.x - o.x, this.y - o.y);
    }
    public Point multiply(double m) {
      return new Point(this.x * m, this.y * m);
    }
    public Point divide(double m) {
      return new Point(this.x / m, this.y / m);
    }
    @Override
```

```
  public int compareTo(Point o) {
    if (this.x < o.x) return -1;
    if (this.x > o.x) return 1;
    if (this.y < o.y) return -1;
    if (this.y > o.y) return 1;
    return 0;
  }
  // Euclidean distance between two points;
  double distance(Point o) {
    double d1 = x - o.x, d2 = y - o.y;
    return Math.sqrt(d1 * d1 + d2 * d2);
  }
}
private static double angle(Point p, Point q, Point r) {
  Point u = p.substract(r), v = q.substract(r);
  return Math.atan2(u.crossProduct(v), u.dotProduct(v));
}
private static int turn(Point p, Point q, Point r) {
  return cmp((p.substract(r)).crossProduct(
      q.substract(r)), 0.0);
}
private static boolean between(Point p, Point q, Point r) {
  return turn(p, r, q) == 0 &&
      cmp((p.substract(r)).dotProduct(q.substract(r)),
          0.0) <= 0;
}
private static int inPolygon(Point p, Point[] polygon,
      int polygonSize) {
  double a = 0; int N = polygonSize;
  for (int i = 0; i < N; ++i) {
    if (between(polygon[i], polygon[(i + 1) % N], p))
      return -1;
    a += angle(polygon[i], polygon[(i + 1) % N], p);
  }
    return (cmp(a, 0.0) == 0) ?  0 :  1;
  }
  private static Point GetIntersection(Line2D.Double l1,
        Line2D.Double l2) {
    double A1 = l1.y2 - l1.y1;
    double B1 = l1.x1 - l1.x2;
    double C1 = A1 * l1.x1 + B1 * l1.y1;
    double A2 = l2.y2 - l2.y1;
    double B2 = l2.x1 - l2.x2;
    double C2 = A2 * l2.x1 + B2 * l2.y1;
    double det = A1*B2 - A2*B1;
    if(det == 0){
      // Lines are parallel, check if they are on the same line.
      double m1 = A1 / B1;
      double m2 = A2 / B2;
      // Check whether their slopes are the same or not,
      // or if they are vertical.
      if (cmp(m1, m2) == 0 || (B1 == 0 && B2 == 0)) {
        if ((l1.x1 == l2.x1 && l1.y1 == l2.y1) ||
            (l1.x1 == l2.x2 && l1.y1 == l2.y2))
          return new Point(l1.x1, l1.y1);
        if ((l1.x2 == l2.x1 && l1.y2 == l2.y1) ||
            (l1.x2 == l2.x2 && l1.y2 == l2.y2))
          return new Point(l1.x2, l1.y2);
      }
      return null;
    }
    double x = (B2*C1 - B1*C2) / det;
    double y = (A1*C2 - A2*C1) / det;
    return new Point(x, y);
  }
}
```

```java
////////////////////////////////////
// Segment Tree for Squares Area //
////////////////////////////////////
public class Squares {
  private static class Segment implements Comparable&lt;Segment&gt; {
    public int x;
    public int y1, y2;
    public boolean open;
    public Segment(int x, int y1, int y2, boolean open) {
      this.x = x;
      this.y1 = y1;
      this.y2 = y2;
      this.open = open;
    }
    @Override
    public int compareTo(Segment o) {
      return this.x - o.x;
    }
  }
  private static class SegmentTree {
    public static int SIZE = 1 << 17;
    public int balance;
    public int val;
    public SegmentTree left;
    public SegmentTree right;
    public SegmentTree(int size) {
      this.balance = 0;
      if (size == 1) {
        this.val = 0;
      } else {
        this.left = new SegmentTree(size >> 1);
        this.right = new SegmentTree(size >> 1);
        this.val = this.left.val + this.right.val;
      }
    }
    public void add(int y1, int y2, int l, int r, int d) {
      //System.out.println(y1 + " " + y2 + " " + l + " " + r);
      int mid = (r + l) >> 1;
      if (l >= y1 && r <= y2) {
        this.balance += d;
      } else if (y1 >= mid) {
        this.right.add(y1, y2, mid, r, d);
      } else if (mid >= y2) {
        this.left.add(y1, y2, l, mid, d);
      } else {
        this.left.add(y1, y2, l, mid, d);
        this.right.add(y1, y2, mid, r, d);
      }
      if (this.balance > 0) {
        this.val = r - l;
      } else if (r - l > 1) { // To avoid NullPointerException
        this.val = this.left.val + this.right.val;
      } else {
        this.val = 0;
      }
    }
  }
  public static void main(String[] args)
      throws FileNotFoundException {
    System.setIn(new FileInputStream("squares.in"));
    Scanner reader = new Scanner(System.in);
    int cases = Integer.parseInt(reader.nextLine());
    for (int c = 0; c < cases; ++c) {
      int squares = Integer.parseInt(reader.nextLine());
      Segment[] segs = new Segment[squares * 2];
      String[] parts;
      for (int n = 0; n < squares; ++n) {
        int x, y, l;
```

```
      parts = reader.nextLine().split("[ ]+");
      x = Integer.parseInt(parts[0]);
      y = Integer.parseInt(parts[1]);
      l = Integer.parseInt(parts[2]);
      segs[2 * n] = new Segment(x, y, y + l, true);
      segs[2 * n + 1] = new Segment(x + l, y, y + l, false);
    }
    Arrays.sort(segs);
    SegmentTree tree = new SegmentTree(SegmentTree.SIZE);
    int res = 0;
    for (int i = 0; i < segs.length - 1; ++i) {
      int d = (segs[i].open) ?  +1 :  -1;
      tree.add(segs[i].y1, segs[i].y2, 0,
          SegmentTree.SIZE, d);
      res += (segs[i + 1].x - segs[i].x) * tree.val;
    }
    System.out.println(res);
  }
 }
}
```