


[Register Now](#)

Competitions

[TopCoder Networks](#)
[Events](#)
[Statistics](#)
[Tutorials](#)
[Forums](#)
[Surveys](#)
[My TopCoder](#)
[Help Center](#)
[About TopCoder](#)


Member Search:

 Handle:
[Advanced Search](#)


Forums

[Round Tables](#)
[New s Discussions](#)
[Algorithm Matches](#)
[Marathon Matches](#)
[NASA Tournament Lab](#)
[Software Forums](#)
[TopCoder Cookbook](#)
[High School Matches](#)
[Sponsor Discussions](#)
[Search](#)
[Watch Thread](#) | [My Post History](#) | [My Watches](#) | [User Settings](#)

 View: [Flat](#) (newest first) | [Threaded](#) | [Tree](#)
[Previous Thread](#) | [Next Thread](#)
[Forums](#) ▶ [TopCoder Cookbook](#) ▶ [Algorithm Competitions - New Recipes](#) ▶ [Line sweep algorithms, part 1](#)
[Line sweep algorithms, part 1](#) | [Reply](#)

Mon, Aug 23, 2010 at 3:47 PM EDT


bmerri
[316 posts](#)
Name: Line sweep algorithms

Problem: Geometry problems often have a solution that is simple, but an order of magnitude too slow for the constraints.

Solution: A powerful tool for geometry problems is a sweep line: a vertical line that is conceptually "swept" across the plane. In practise, of course, we cannot simulate all points in time and so we consider only some discrete points. At each point that is processed, we make use of the previous processing to compute a result.

In most cases, there will be a fixed set of X values that are of interest, and the sweep can be simulated by sorting them and then processing them in order. In more complex cases, the set of interesting X values is not known in advance, and a priority queue can be used to direct the simulation.

Discussion: The concept is best illustrated by example. We'll start with the following problem: given a set of N points, find the pair that is closest together. The obvious solution (testing all pairs) takes $O(N^2)$ time, but a line sweep can reduce this to $O(N \log N)$.

Suppose that we have processed points 1 to $i - 1$ (ordered by X) and the shortest distance we have found so far is h. We now process point i and try to find a point closer to it than h. We maintain a set of all already-processed points whose X coordinates are within h of point N, as shown in the light grey rectangle. As each point is processed, it is added to the set, and when we move on to the next point or when h is decreased, points are removed from the set. The set is ordered by Y coordinate. A balanced binary tree is suitable for this, and accounts for the log N factor.

 [Insert image at <http://www.topcoder.com/i/education/lineSweep/closest.png>]

Here are example implementations in C++ and Java:

```
typedef complex<double> pnt;    // stores X in real part, Y in imaginary part

struct compare_x    // orders by X, breaking ties by Y
{
    bool operator()(const pnt &a, const pnt &b) const
    {
        if (a.real() != b.real())
            return a.real() < b.real();
        else
            return a.imag() < b.imag();
    }
};

struct compare_y    // orders by Y, breaking ties by X
{
    bool operator()(const pnt &a, const pnt &b) const
    {
        if (a.imag() != b.imag())
            return a.imag() < b.imag();
        else
            return a.real() < b.real();
    }
};

// Returns the closest distance between two points in S
static double closest(const vector<pnt> &S)
{
    int N = S.size();
    vector<pnt> Sx = S;    // to be sorted by X coordinate
    set<pnt, compare_y> Sy; // active points, ordered by Y coordinate
    int tail = 0;          // points in Sx in the range [tail, i) are in Sy

    sort(Sx.begin(), Sx.end(), compare_x());
    double h = HUGE_VAL;
    for (int i = 0; i < N; i++)
    {
        // erase points whose X value is too small to even consider
        while (Sx[i].real() - Sx[tail].real() > h)
        {
```

```

        Sy.erase(Sx[tail]);
        tail++;
    }

    set<pnt, compare_y>::const_iterator y1, y2; // range in Sy within h of current point
    y1 = lower_bound(Sy.begin(), Sy.end(), pnt(-HUGE_VAL, Sx[i].imag() - h), compare_y());
    y2 = upper_bound(Sy.begin(), Sy.end(), pnt( HUGE_VAL, Sx[i].imag() + h), compare_y());

    for (set<pnt, compare_y>::const_iterator j = y1; j != y2; ++j)
        h = min(h, abs(Sx[i] - *j));

    Sy.insert(Sx[i]); // add current point to active set
}
return h;
}

public class Closest
{
    private static class Point
    {
        public double x;
        public double y;

        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public double distance(Point b)
        {
            return Math.hypot(b.x - x, b.y - y);
        }
    }

    private static class CompareX implements Comparator<Point>
    {
        public int compare(Point a, Point b) // orders by X, breaking ties by Y
        {
            if (a.x < b.x) return -1;
            else if (a.x > b.x) return 1;
            else if (a.y < b.y) return -1;
            else if (a.y > b.y) return 1;
            else return 0;
        }
    }

    private static class CompareY implements Comparator<Point>
    {
        public int compare(Point a, Point b) // orders by Y, breaking ties by X
        {
            if (a.y < b.y) return -1;
            else if (a.y > b.y) return 1;
            else if (a.x < b.x) return -1;
            else if (a.x > b.x) return 1;
            else return 0;
        }
    }

    public static double closest(List<Point> S)
    {
        CompareX compareX = new CompareX();
        CompareY compareY = new CompareY();
        Point[] Sx = S.toArray(new Point[0]); // Take a copy we will sort by X
        int N = Sx.length;
        SortedSet<Point> Sy = new TreeSet<Point>(compareY); // Active points
        int tail = 0; // points in Sx in the range [tail, i) are in Sy

        Arrays.sort(Sx, compareX);
        double h = Double.POSITIVE_INFINITY;

        for (int i = 0; i < N; i++)
        {
            // erase points whose X value is too small to even consider
            while (Sx[i].x - Sx[tail].x > h)
            {
                Sy.remove(Sx[tail]);
                tail++;
            }

            // Identify points in Sy within h vertically of Sx[i]
            SortedSet<Point> range = Sy.subSet(
                new Point(Double.NEGATIVE_INFINITY, Sx[i].y - h),
                new Point(Double.POSITIVE_INFINITY, Sx[i].y + h));
            for (Point p : range)
            {
                h = Math.min(h, Sx[i].distance(p));
            }
        }
    }
}

```

```

    }

    Sy.add(Sx[i]); // Add latest point to active set
}

return h;
}
}

```

Line sweep algorithms, part 2 (response to [post](#) by [bmerry](#)) | [Reply](#)

Mon, Aug 23, 2010 at 3:48 PM EDT



bmerry
[316 posts](#)

That is just one example of how a sweep line can be used. Let's consider another problem: given a set of line segments, find all intersections. Once again, there is a relatively obvious $O(N^2)$ solution (test all pairs), and a more efficient solution based on a sweep line.

We'll start by considering the problem of returning all intersections in a set of horizontal and vertical line segments. Since horizontal lines don't have a single X coordinate, we have to abandon the idea of sorting objects by X. Instead, we have the idea of an *event*: an X coordinate at which something interesting happens. In this case, the three types of events are: start of a horizontal line, end of a horizontal line, and a vertical line. As the sweep line moves, we'll keep an active set of horizontal lines cut by the sweep line, sorted by Y value (the red lines in the figure).

[Insert image at <http://www.topcoder.com/i/education/lineSweep/closest.png>]

To handle either of the horizontal line events, we simply need to add or remove an element from the set. Again, we can use a balanced binary tree to guarantee $O(\log N)$ time for these operations. When we hit a vertical line, a range search immediately gives all the horizontal lines that it cuts. If horizontal or vertical segments can overlap there is some extra work required, and we must also consider whether lines are considered to include their endpoints, but none of this affects the computational complexity. This algorithm requires $O(N \log N + I)$ time to return I intersections, but with a suitable tree structure (one which tracks the size of each subtree) it is possible to count the intersections in $O(N \log N)$ time.

Here are example implementations in C++ and Java:

```

typedef complex<int> pnt; // stores X in real part, Y in imaginary part

enum event_type
{
    EVENT_END,          // ordered this way so that events with same x sort this way
    EVENT_VERTICAL,
    EVENT_START
};

struct event
{
    event_type type;
    int x;
    int line;           // index into list of lines

    event() {}
    event(event_type type, int x, int line) : type(type), x(x), line(line) {}

    // sort by X then by type
    bool operator <(const event &b) const
    {
        if (x != b.x) return x < b.x;
        else return type < b.type;
    }
};

struct line
{
    int x1, y1, x2, y2;

    line() {}
    line(int x1, int y1, int x2, int y2) : x1(x1), y1(y1), x2(x2), y2(y2) {}
};

// Returns all intersections between lines. The lines must all be either
// horizontal and vertical, and only horizontal-vertical intersections are
// counted (i.e. not overlaps). Lines are considered to exclude their
// endpoints. Also, each line must have x1 <= x2 and y1 <= y2.
static vector<pnt> hv_intersections(const vector<line> &lines)
{
    int L = lines.size();
    vector<event> events;
    vector<pnt> ans;

    // Y coordinates of active lines
    multiset<int> active;
    // Convert lines into events
    for (int i = 0; i < L; i++)
    {
        if (lines[i].y1 != lines[i].y2)
            events.push_back(event(EVENT_VERTICAL, lines[i].x1, i));
        else if (lines[i].x1 != lines[i].x2)
        {
            events.push_back(event(EVENT_START, lines[i].x1, i));
            events.push_back(event(EVENT_END, lines[i].x2, i));
        }
    }
}

```

```

// Sort events by X
sort(events.begin(), events.end());

// Process events
for (vector<Event>::const_iterator e = events.begin(); e != events.end(); ++e)
{
    switch (e->type)
    {
        case EVENT_START:
            active.insert(lines[e->line].y1);
            break;
        case EVENT_END:
            active.erase(active.find(lines[e->line].y1));
            break;
        case EVENT_VERTICAL:
            {
                // Iterate over all y values for intersecting horizontal lines
                multiset<int>::const_iterator first, last, i;
                first = active.upper_bound(lines[e->line].y1);
                last = active.lower_bound(lines[e->line].y2);
                for (i = first; i != last; ++i)
                    ans.push_back(pnt(e->x, *i));
            }
            break;
    }
}

return ans;
}

public class Lines1
{
    private static class Point
    {
        public int x;
        public int y;

        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }

    private static enum EventType
    {
        EVENT_END,
        EVENT_VERTICAL,
        EVENT_START
    }

    private static class Line
    {
        public int x1, y1, x2, y2;

        public Line(int x1, int y1, int x2, int y2)
        {
            this.x1 = x1; this.y1 = y1;
            this.x2 = x2; this.y2 = y2;
        }
    }

    private static class Event implements Comparable<Event>
    {
        public EventType type;
        public int x;
        public Line line;

        public Event(EventType type, int x, Line line)
        {
            this.type = type;
            this.x = x;
            this.line = line;
        }

        // sorts by x then by type.
        public int compareTo(Event b)
        {
            if (x != b.x) return x - b.x;
            else return type.compareTo(b.type);
        }
    }

    private static List<Point> hvIntersections(List<Line> lines)
    {
        ArrayList<Point> ans = new ArrayList<Point>();
        // number of active horizontal lines with each y value
        SortedMap<Integer, Integer> active = new TreeMap<Integer, Integer>();
        ArrayList<Event> events = new ArrayList<Event>();

```

```

for (Line line : lines)
{
    if (line.y1 != line.y2)
        events.add(new Event(EventType.EVENT_VERTICAL, line.x1, line));
    else
    {
        events.add(new Event(EventType.EVENT_START, line.x1, line));
        events.add(new Event(EventType.EVENT_END, line.x2, line));
    }
}
Collections.sort(events);

for (Event e : events)
{
    switch (e.type)
    {
        case EVENT_START:
        {
            // Increment count of lines with y == e.line.y1
            Integer count = active.get(e.line.y1);
            active.put(e.line.y1, count == null ? 1 : count + 1);
        }
        break;
        case EVENT_END:
        {
            // Decrement count of lines with y == e.line.y1
            int count = active.get(e.line.y1) - 1;
            if (count > 0) active.put(e.line.y1, count);
            else active.remove(e.line.y1);
        }
        break;
        case EVENT_VERTICAL:
        {
            // Iterate over active horizontal lines with suitable y values
            SortedMap<Integer, Integer> view = active.subMap(e.line.y1 + 1, e.line.y2);
            for (Map.Entry<Integer, Integer> i : view.entrySet())
            {
                for (int j = 0; j < i.getValue(); j++)
                    ans.add(new Point(e.line.x1, i.getKey()));
            }
        }
        break;
    }
}
return ans;
}
}

```

Line sweep algorithms, part 3 (response to [post](#) by [bmerry](#)) | [Reply](#)

Mon, Aug 23, 2010 at 3:49 PM EDT



bmerry
[316 posts](#)

Let's consider some contest problems, starting with [PowerSupply](#). For each power line orientation, consider a sweep line oriented in that direction, and sweep it in the perpendicular direction. Consumers are added D units ahead of the sweep and dropped D units behind the sweep. Here is an implementation in C++ and Java:

```

class PowerSupply
{
public:
    int maxProfit(vector<int> x, vector<int> y, int D)
    {
        const int xscale[4] = {0, 1, 1, 1};
        const int yscale[4] = {1, 0, 1, -1};

        int ans = 0;
        for (int pass = 0; pass < 4; pass++)
        {
            int xs = xscale[pass];
            int ys = yscale[pass];
            // Account for non-unit scaling along diagonals
            double max_sep = 2 * D * sqrt(xs * xs + ys * ys);

            int N = x.size();
            vector<int> points; // measured along sweep direction
            for (int i = 0; i < N; i++)
                points.push_back(xs * x[i] + ys * y[i]);
            sort(points.begin(), points.end());

            int first = 0;
            for (int last = 0; last < N; last++)
            {
                while (points[last] - points[first] > max_sep)
                    first++;
                ans = max(ans, last - first + 1);
            }
        }
        return ans;
    }
};

```

```

public class PowerSupply
{
    public int maxProfit(int[] x, int[] y, int D)
    {
        final int[] xscale = {0, 1, 1, 1};
        final int[] yscale = {1, 0, 1, -1};

        int ans = 0;
        for (int pass = 0; pass < 4; pass++)
        {
            int xs = xscale[pass];
            int ys = yscale[pass];
            // Account for non-unit scaling along diagonals
            double max_sep = 2 * D * Math.sqrt(xs * xs + ys * ys);

            int N = x.length;
            int[] points = new int[N]; // measured along sweep direction
            for (int i = 0; i < N; i++)
                points[i] = xs * x[i] + ys * y[i];
            Arrays.sort(points);

            int first = 0;
            for (int last = 0; last < N; last++)
            {
                while (points[last] - points[first] > max_sep)
                    first++;
                ans = Math.max(ans, last - first + 1);
            }
        }
        return ans;
    }
}

```

Line sweep algorithms, part 4 (response to [post by bmerri](#)) | [Reply](#)[1 edit](#) | Mon, Aug 23, 2010 at 3:49 PM EDT

bmerri
[316 posts](#)

Now let's see what a sweep line can do for [ConvexPolygons](#). Consider a vertical sweep-line sweeping horizontally. The events of interest are the vertices of the two polygons, and the intersection points of their edges. Between consecutive events, the section cut by the sweep line varies linearly. Thus, we can sample the cut area at the mid-point X value of each of these regions to get the average for the whole region. Sampling at these mid-points also eliminates a lot of special-case handling, because the sweep line is guaranteed not to pass through a vertex.

The last detail is measuring the portion of a specific sweep line covered by the intersection. This can again be done by a sweeping algorithm. Consider a point that sweeps vertically along the sweep line, keeping track of whether it is inside or outside each of the polygons. The events of interest are now the Y values at which either polygon cuts the sweep line; every time a polygon edge is crossed, the sweep point moves either into or out of the polygon.

Here are implementations in C++ and Java (the `Point` class and `intersect` method are left out for brevity).

```

class ConvexPolygons
{
public:
    double overlap(vector<string> polygon1, vector<string> polygon2)
    {
        vector<string> polygons[2] = {polygon1, polygon2};
        vector<pnt> vertices[2];
        int V[2];
        vector<double> xs;
        double ans = 0.0;

        for (int p = 0; p < 2; p++)
        {
            V[p] = polygons[p].size();
            for (int i = 0; i < V[p]; i++)
            {
                int x, y;
                sscanf(polygons[p][i].c_str(), "%d %d", &x, &y);
                vertices[p].push_back(pnt(x, y));
                xs.push_back(x);
            }
            vertices[p].push_back(vertices[p][0]);
        }

        for (int i = 0; i < V[0]; i++)
            for (int j = 0; j < V[1]; j++)
            {
                pnt cut;
                if (intersect(vertices[0][i], vertices[0][i + 1],
                             vertices[1][j], vertices[1][j + 1],
                             cut))
                {
                    xs.push_back(cut.real());
                }
            }

        sort(xs.begin(), xs.end());
        // Eliminate duplicates
        xs.erase(unique(xs.begin(), xs.end()), xs.end());

        int E = xs.size();
    }
}

```

```

for (int i = 0; i + 1 < E; i++)
{
    double x = (xs[i] + xs[i + 1]) * 0.5;
    pnt sweep0(x, 0);
    pnt sweep1(x, 1);
    // pair of (y, coverage mask delta)
    vector<pair<double, int> > events;

    for (int p = 0; p < 2; p++)
    {
        for (int j = 0; j < V[p]; j++)
        {
            pnt cut;
            intersect(vertices[p][j], vertices[p][j + 1], sweep0, sweep1, cut);
            double y = cut.imag();
            if (vertices[p][j].real() < x && vertices[p][j + 1].real() > x)
            {
                events.push_back(make_pair(y, 1 << p));
            }
            else if (vertices[p][j].real() > x && vertices[p][j + 1].real() < x)
            {
                events.push_back(make_pair(y, -(1 << p)));
            }
        }
    }
    sort(events.begin(), events.end());

    double a = 0.0;
    int mask = 0;
    for (size_t j = 0; j < events.size(); j++)
    {
        if (mask == 3)
            a += events[j].first - events[j - 1].first;
        mask += events[j].second;
    }
    ans += a * (xs[i + 1] - xs[i]);
}
return ans;
};

public class ConvexPolygons
{
    private static class Event implements Comparable<Event>
    {
        public double y;
        public int mask_delta;

        public int compareTo(Event b)
        {
            if (y < b.y) return -1;
            else if (y > b.y) return 1;
            else return mask_delta - b.mask_delta;
        }

        public Event(double y, int mask_delta)
        {
            this.y = y;
            this.mask_delta = mask_delta;
        }
    }

    public double overlap(String[] polygon1, String[] polygon2)
    {
        String[][] polygons = { polygon1, polygon2 };
        ArrayList<ArrayList<Point> > vertices = new ArrayList<ArrayList<Point> >();
        vertices.add(new ArrayList<Point>());
        vertices.add(new ArrayList<Point>());

        int[] V = {0, 0};
        SortedSet<Double> xs = new TreeSet<Double>();
        double ans = 0.0;

        for (int p = 0; p < 2; p++)
        {
            V[p] = polygons[p].length;
            for (int i = 0; i < V[p]; i++)
            {
                String[] xy = polygons[p][i].split(" ");
                double x = Integer.parseInt(xy[0]);
                double y = Integer.parseInt(xy[1]);
                vertices.get(p).add(new Point(x, y));
                xs.add(x);
            }
            vertices.get(p).add(vertices.get(p).get(0));
        }
    }
}

```

```

for (int i = 0; i < V[0]; i++)
    for (int j = 0; j < V[1]; j++)
    {
        Point cut = new Point(0, 0);
        if (intersect(vertices.get(0).get(i), vertices.get(0).get(i + 1),
            vertices.get(1).get(j), vertices.get(1).get(j + 1),
            cut))
        {
            xs.add(cut.x);
        }
    }

Double[] xsa = xs.toArray(new Double[0]);
int E = xsa.length;
for (int i = 0; i + 1 < E; i++)
{
    double x = (xsa[i] + xsa[i + 1]) * 0.5;
    Point sweep0 = new Point(x, 0);
    Point sweep1 = new Point(x, 1);
    ArrayList<Event> events = new ArrayList<Event>();

    for (int p = 0; p < 2; p++)
    {
        for (int j = 0; j < V[p]; j++)
        {
            Point cut = new Point(0, 0);
            intersect(vertices.get(p).get(j), vertices.get(p).get(j + 1), sweep0, sweep1, cut);
            double y = cut.y;
            double x0 = vertices.get(p).get(j).x;
            double x1 = vertices.get(p).get(j + 1).x;
            if (x0 < x && x1 > x)
            {
                events.add(new Event(y, 1 << p));
            }
            else if (x0 > x && x1 < x)
            {
                events.add(new Event(y, -(1 << p)));
            }
        }
    }
    Collections.sort(events);

    double a = 0.0;
    int mask = 0;
    for (int j = 0; j < events.size(); j++)
    {
        if (mask == 3)
            a += events.get(j).y - events.get(j - 1).y;
        mask += events.get(j).mask_delta;
    }

    ans += a * (xsa[i + 1] - xsa[i]);
}
return ans;
}
}

```

[EDIT: changed subject]

Line sweep algorithms, part 5 (response to [post by bmerry](#)) | [Reply](#)

Mon, Aug 30, 2010 at 10:33 AM EDT



bmerry
[316 posts](#)

Sometimes a line sweep can be useful even when a problem is not obviously geometry-based. [CornersDecoding](#) is a good example of this. It is based on a grid of cells, but a quick look at the constraints shows that simulating the entire grid is infeasible. Instead, one can sweep a horizontal line vertically through the cells. For any position of the sweep line, we can represent the cells it cuts compactly by the boundaries between black and white cells (essentially a 1D equivalent to the corner encoding). From this representation, it is easy to determine the number of black cells in a row.

The total number of black cells is simply the total number of black cells in all rows. However, we do not need to iterate over rows one at a time. Each row is the same as the last, except where there is a corner. Thus, between any two corners, we need only count the black cells per row once, and then multiply this count by the number of rows. To move the sweep-line over a corner, we need only add or remove it from the set of black-white boundaries. Example code follows:

```

long long CornersDecoding::blackPixels(vector<int> rows, vector<int> cols)
{
    int N = rows.size();
    vector<pair<int, int> > corners(N);
    for (int i = 0; i < N; i++)
        corners[i] = make_pair(rows[i], cols[i]);
    sort(corners.begin(), corners.end());
    set<int> xs;

    int p = 0;
    int lasty = 0;
    long long ans = 0;
    while (p < N)
    {
        int horiz = 0;
        // ...
    }
}

```



```

set<int>::const_iterator pos = xs.begin();
while (pos != xs.end())
{
    int l = *pos++;
    int r = *pos++;
    horiz += r - l;
}
ans += (long long) horiz * (corners[p].first - lasty);
lasty = corners[p].first;

int q = p;
while (q < N && corners[q].first == corners[p].first)
{
    if (xs.count(corners[q].second))
        xs.erase(corners[q].second);
    else
        xs.insert(corners[q].second);
    q++;
}
if ((q - p) & 1)
    return -1;
p = q;
}
if (!xs.empty())
    return -1;

return ans;
}

public class CornersDecoding
{
    private static class Corner implements Comparable<Corner>
    {
        int row, col;

        public Corner(int row, int col)
        {
            this.row = row;
            this.col = col;
        }

        public int compareTo(Corner b)
        {
            if (row != b.row) return row - b.row;
            else return col - b.col;
        }
    }

    public long blackPixels(int[] rows, int[] cols)
    {
        int N = rows.length;
        Corner[] corners = new Corner[N];
        for (int i = 0; i < N; i++)
            corners[i] = new Corner(rows[i], cols[i]);
        Arrays.sort(corners);
        SortedSet<Integer> xs = new TreeSet<Integer>();

        int p = 0;
        int lasty = 0;
        long ans = 0;
        while (p < N)
        {
            int horiz = 0;
            int parity = -1;
            for (int x : xs)
            {
                horiz += parity * x;
                parity = -parity;
            }
            ans += (long) horiz * (corners[p].row - lasty);
            lasty = corners[p].row;

            int q = p;
            while (q < N && corners[q].row == corners[p].row)
            {
                if (xs.contains(corners[q].col))
                    xs.remove(corners[q].col);
                else
                    xs.add(corners[q].col);
                q++;
            }
            if (0 != ((q - p) & 1))
                return -1;

            p = q;
        }
        if (!xs.isEmpty())
            return -1;
        return ans;
    }
}

```

Re: Line sweep algorithms, part 2 (response to [post](#) by [bmerri](#)) | [Reply](#)

Fri, Oct 25, 2013 at 5:10 AM EDT

haniDaher
[1 post](#)

Hello bmerri,

I actually have a question concerning the sweep line algorithm. I wanted to add an extra condition to the Intersection, which is represented by the distance between the horizontal lines.

What I mean is that if there is an intersection between 2 lines and the distance is less than a given threshold then we consider that the 2 horizontal segments have intersection points.

Should the distance be considered as an EVENT? and Can I add this condition to the sweep line algorithm?

Thank you.

[Forums](#) ▶ [TopCoder Cookbook](#) ▶ [Algorithm Competitions - New Recipes](#) ▶ [Line sweep algorithms, part 1](#)[Previous Thread](#) | [Next Thread](#)[RSS](#)[Home](#) | [About TopCoder](#) | [Press Room](#) | [Contact Us](#) | [Careers](#) | [Privacy](#) | [Terms](#)
[Competitions](#) | [Cockpit](#)

Copyright TopCoder, Inc. 2001-2014