Chapter 30

Sweep line algorithms

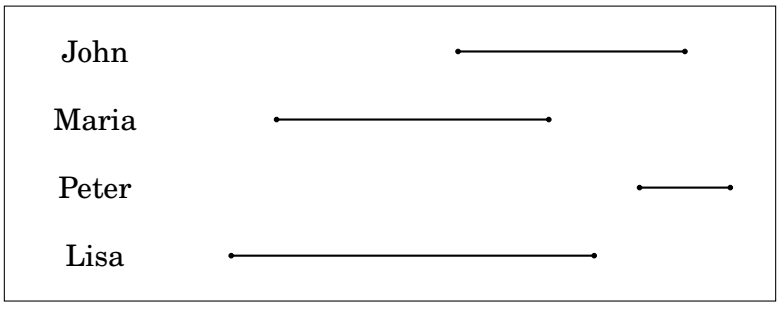
Many geometric problems can be solved using **sweep line** algorithms. The idea in such algorithms is to represent an instance of the problem as a set of events that correspond to points in the plane. The events are processed in increasing order according to their x or y coordinates.

As an example, consider the following problem: There is a company that has n employees, and we know for each employee their arrival and leaving times on a certain day. Our task is to calculate the maximum number of employees that were in the office at the same time.

The problem can be solved by modeling the situation so that each employee is assigned two events that correspond to their arrival and leaving times. After sorting the events, we go through them and keep track of the number of people in the office. For example, the table

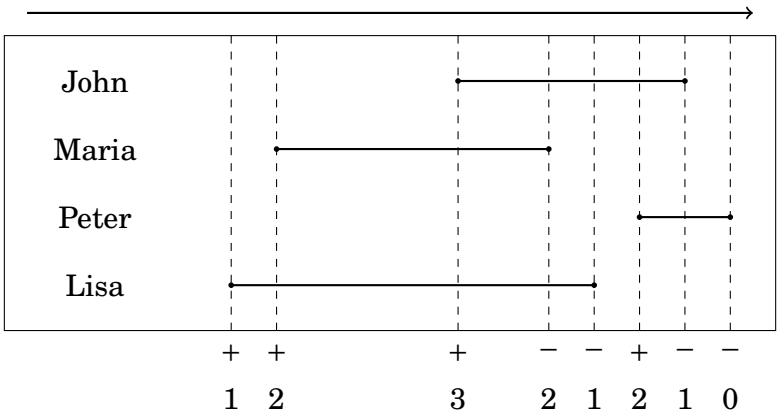
person	arrival time	leaving time
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

corresponds to the following events:



We go through the events from left to right and maintain a counter. Always when a person arrives, we increase the value of the counter by one, and when a person leaves, we decrease the value of the counter by one. The answer to the problem is the maximum value of the counter during the algorithm.

In the example, the events are processed as follows:

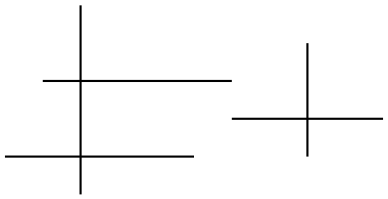


The symbols + and − indicate whether the value of the counter increases or decreases, and the value of the counter is shown below. The maximum value of the counter is 3 between John's arrival and Maria's leaving.

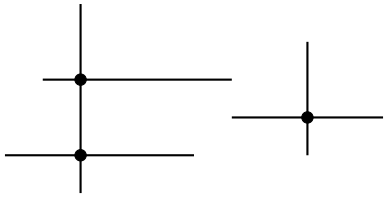
The running time of the algorithm is $O(n \log n)$, because sorting the events takes $O(n \log n)$ time and the rest of the algorithm takes $O(n)$ time.

Intersection points

Given a set of n line segments, each of them being either horizontal or vertical, consider the problem of counting the total number of intersection points. For example, when the line segments are



there are three intersection points:

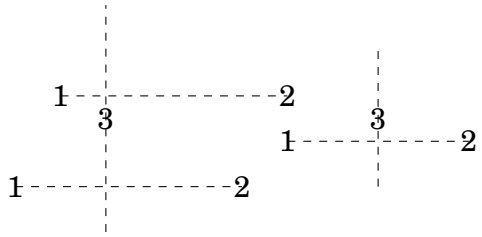


It is easy to solve the problem in $O(n^2)$ time, because we can go through all possible pairs of line segments and check if they intersect. However, we can solve the problem more efficiently in $O(n \log n)$ time using a sweep line algorithm and a range query data structure.

The idea is to process the endpoints of the line segments from left to right and focus on three types of events:

(1) horizontal segment begins
(2) horizontal segment ends
(3) vertical segment

The following events correspond to the example:



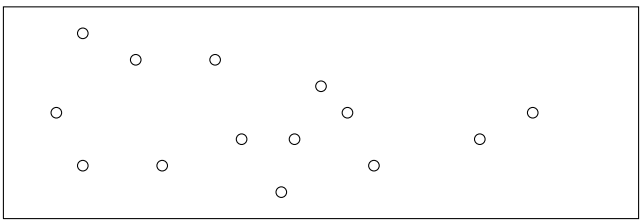
We go through the events from left to right and use a data structure that maintains a set of y coordinates where there is an active horizontal segment. At event 1, we add the y coordinate of the segment to the set, and at event 2, we remove the y coordinate from the set.

Intersection points are calculated at event 3. When there is a vertical segment between points y_1 and y_2, we count the number of active horizontal segments whose y coordinate is between y_1 and y_2, and add this number to the total number of intersection points.

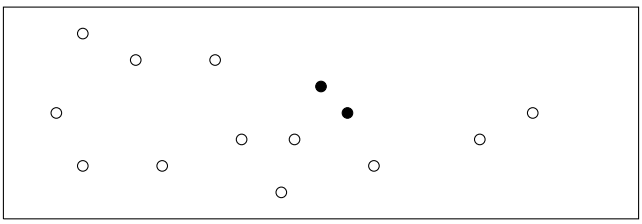
To store y coordinates of horizontal segments, we can use a binary indexed or segment tree, possibly with index compression. When such structures are used, processing each event takes $O(\log n)$ time, so the total running time of the algorithm is $O(n \log n)$.

Closest pair problem

Given a set of n points, our next problem is to find two points whose Euclidean distance is minimum. For example, if the points are



we should find the following points:



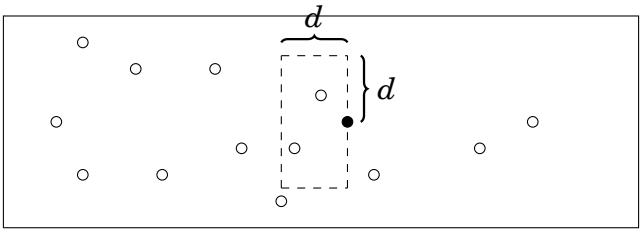
This is another example of a problem that can be solved in $O(n \log n)$ time using a sweep line algorithm[1]. We go through the points from left to right and maintain a value d: the minimum distance between two points seen so far. At

[1]Besides this approach, there is also an $O(n \log n)$ time divide-and-conquer algorithm [56] that divides the points into two sets and recursively solves the problem for both sets.

each point, we find the nearest point to the left. If the distance is less than d, it is the new minimum distance and we update the value of d.

If the current point is (x,y) and there is a point to the left within a distance of less than d, the x coordinate of such a point must be between $[x-d,x]$ and the y coordinate must be between $[y-d,y+d]$. Thus, it suffices to only consider points that are located in those ranges, which makes the algorithm efficient.

For example, in the following picture, the region marked with dashed lines contains the points that can be within a distance of d from the active point:



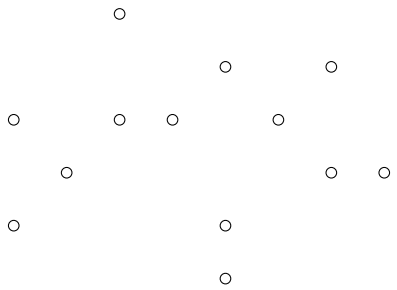
The efficiency of the algorithm is based on the fact that the region always contains only $O(1)$ points. We can go through those points in $O(\log n)$ time by maintaining a set of points whose x coordinate is between $[x-d,x]$, in increasing order according to their y coordinates.

The time complexity of the algorithm is $O(n \log n)$, because we go through n points and find for each point the nearest point to the left in $O(\log n)$ time.

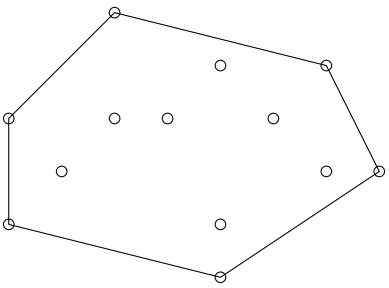
Convex hull problem

A **convex hull** is the smallest convex polygon that contains all points of a given set. Convexity means that a line segment between any two vertices of the polygon is completely inside the polygon.

For example, for the points



the convex hull is as follows:



Andrew's algorithm [3] provides an easy way to construct the convex hull for a set of points in $O(n \log n)$ time. The algorithm first locates the leftmost and rightmost points, and then constructs the convex hull in two parts: first the upper hull and then the lower hull. Both parts are similar, so we can focus on constructing the upper hull.

First, we sort the points primarily according to x coordinates and secondarily according to y coordinates. After this, we go through the points and add each point to the hull. Always after adding a point to the hull, we make sure that the last line segment in the hull does not turn left. As long as it turns left, we repeatedly remove the second last point from the hull.

The following pictures show how Andrew's algorithm works: