CHAPTER 1

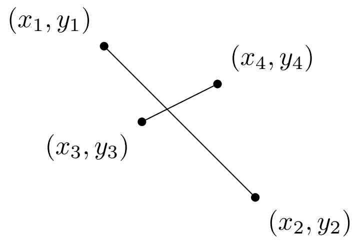
# GEOMETRIC ALGORITHMS

## 1.1 Intersection of Line Segments

In this section, we will look at algorithms to determine whether two line segments intersect, and the geometric interpretation of the algorithm. First, we consider some geometric objects:

- a *point* in 2-dimension is given by $(x, y)$;
- a *line* is an (ordered) pair of points $p$ and $q$, and is denoted by $-p - q-$;
- a *line segment* is a pair of end-points $p$ and $q$, and is denoted by $p - q$;

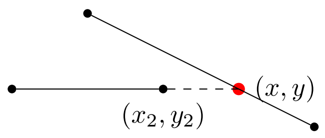
Now, we consider when two line segments $p_1$ and $p_2$ intersect- we do not need to compute the point of intersection. This is possible to do using high school mathematics. For instance, let the line segments be the following:



Then, the equations of the two lines are:

$$(x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) = 0,$$
$$(x_4 - x_3)(y - y_3) - (y_4 - y_3)(x - x_3) = 0.$$

We then find a point of intersection $(p, q)$ by solving the two equations simulatenously, if possible[1]. Finally, to check that the intersection of the two lines is within the line segments, we check that $x_1 \leq p \leq x_2$ and $x_3 \leq p \leq x_4$. The following figure illustrates why this is required:



Using the equation of the two lines, we will find that $(x, y)$ is a solution to the equation. However, we have $x > x_2$, so the intersection does not line on the line segment with $(x_2, y_2)$.

Although possible, this is not the most optimal way of solving the problem. The rearrangement to find the point of intersection involves division, which is

[1]There are two cases when we won't be able to solve this system of equations easily- if the lines are parallel and have different $y$-offset, then they cannot intersect; if the two lines have the same $y$-offset, then they can intersect if the two segments overlap (i.e. $x_1 \leq x_3 \leq x_2$ or $x_1 \leq x_4 \leq x_2$, or vice versa).

susceptible to floating point errors. Moreover, we find the point of intersection even though we just need to find whether the line segments intersect- we are doing more work than required.

We will now look at a more efficient algorithm by breaking the problem down. If we have two line segments $p - q$ and $r - s$, then:

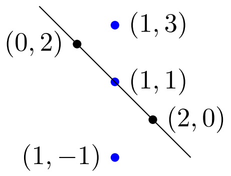
- we first determine whether the two points $p$ and $q$ lie on different sides of the line $r - s$, and vice versa (`onOppositeSides`); and
- we then determine whether the smallest rectangles containing the lines $p - q$ and $r - s$ intersect (`boundingBox`).

The line segments intersect if and only if all the tests pass.

The algorithm `onOppositeSides` takes in two points $a$, $b$ and a line $-p1-p2-$, and determines whether $a$ and $b$ lie on opposite sides of the line. The equation of the line is given by

$$(l.p2.x - l.p1.x)(y - l.p1.y) - (l.p2.y - l.p1.y)(x - 1.p1.x) = 0.$$

When we substitute $(x, y)$ as the points $a$ and $b$, if the value at the LHS of the equation have different signs, then the two points lie on opposite sides. To see this, consider the following example:



Then, the equation of the line is $y + x - 2 = 0$. We can substitute the three blue points to the left hand side of the equation to get a value:

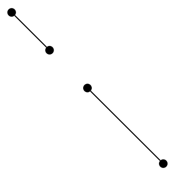
- if $(x, y) = (1, 3)$, then $y + x - 2 = 2 > 0$;
- if $(x, y) = (1, 1)$, then $y + x - 2 = 0$;
- if $(x, y) = (1, -1)$, then $y + x - 2 = -2 < 0$.

In the figure above, all 3 pairs of points are considered to be on opposite sides- we consider $(1, 1)$ to be both above and below the line since it lies on the line. An easy way to check this is if the product of the two values to be non-negative. Note that we are meant to accept the case when both the lines are $(1, 1)$- this corresponds to the trivial line segment that intersects the line given at $(1, 1)$.

The algorithm `onOppositeSides` is given below.

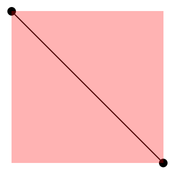
```
bool onOppositeSides(Line line, Point a, Point b) {
    double v1 = (line.p2.x - line.p1.x) * (a.y - line.p1.y) -
                (line.p2.y - line.p2.y) * (a.x - line.p1.x);
    double v2 = (line.p2.x - line.p1.x) * (b.y - line.p1.y) -
                (line.p2.y - line.p2.y) * (b.x - line.p1.x);
    return v1 * v2 <= 0;
}
```

The condition `onOppositeSides` is not enough to check that the two line segments intersect. For example, consider the following line segments:

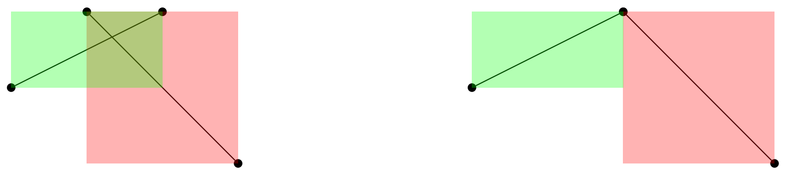


Since the two line segments are part of the same line, the result of `onOppositeSides` in both cases will be `true`. But, since the two line segments do not overlap, they do not intersect. For this reason, we require a third check- the bounding box.

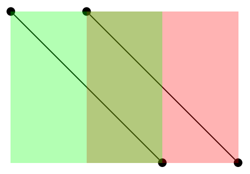
The *bounding box* of a line segment $p - q$ is the smallest rectangle that contains the line segment $p - q$ whose sides are parallel to the $x$ and the $y$-axes. For example, below is a line segment and its bounding box:



For two line segments to intersect, we require the bounding boxes to intersect. We illustrate this in the examples below:

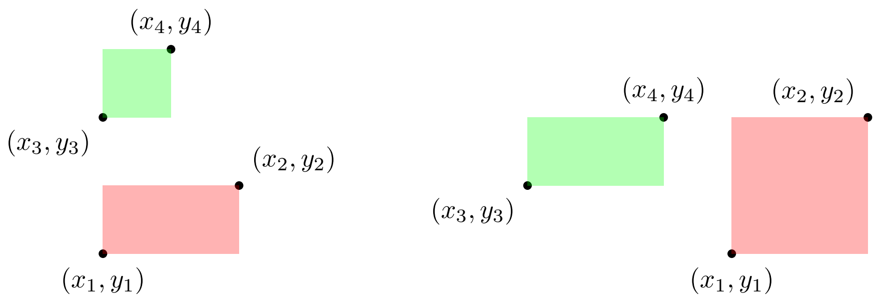


We see that when the two line segments intersect, their bounding boxes intersect. Also, if the bounding boxes do not intersect, then the two lines cannot intersect. Nonetheless, it is possible for two line segments to intersect and there to still not be an intersection- consider the following example.



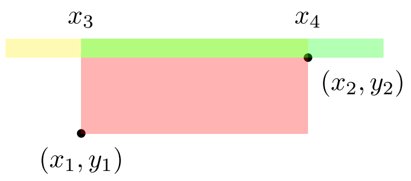
In this case, although the two lines do not intersect, their bounding boxes still intersect. The issue here is that in both line segments, the points are both on the same side of the other line segment.

Now, we consider how to define the `boundingBox` algorithm. The problem is essentially about when two rectangles intersect. So, consider the following two pairs of rectangles:



The rectangles on the left do not intersect since $y_3 > y_2$ and $x_3 > x_2$. On the other hand, the rectangles on the right do not intersect since $x_1 > x_4$ and $y_1 > y_4$. These give us four potential conditions required for rectangles to intersect- $x_3 \leq x_2, y_3 \leq y_2, x_1 \leq x_4, y_1 \leq y_4$.

Assume now we fix one of the rectangles and consider what happens when the $x$-coordinates satisfy this condition:



The value $x_3$ comes to the left of $x_2$, while the value $x_4$ comes to the right of $x_1$. In particular, $x_3$ can be at any point within the yellow region, and $x_4$ on the green region. This ensures that the two rectangles do intersect with respect to the $x$-coordinate. So, if we further require $y_3 \leq y_2$ and $y_1 \leq y_4$, the rectangles must intersect in both $x$- and $y$-coordinate, which means that they intersect in 2D. Therefore, these 4 conditions are sufficient. Hence, the algorithm `boundingBox` is the following.

```
bool boundingBox(LineSegment l1, LineSegment l2) {
    Rectangle r1 = new Rectangle(l1);
    Rectangle r2 = new Rectangle(l2);

    return  r2.bottomLeft.x <= r1.topRight.x &&
            r1.bottomLeft.x <= r2.topRight.x &&
            r2.bottomLeft.y <= r1.topRight.y &&
            r1.bottomLeft.y <= r2.topRight.y;
}
```

Putting this all together, we find that for line segments $p - q$ and $r - s$ intersect if and only if

$$\texttt{onOppositeSides(p, q, r-s)} \wedge \texttt{onOppositeSides(r, s, p-q)} \\ \wedge \texttt{boundingBox(p-q, r-s)}.$$

If two line segments intersect, then we know that for each of the line segment, the two points lie on opposite sides of the other one, and that their bounding boxes must intersect. So, this is a necessary condition for the line segments to intersect. Moreover, if two line segments do not intersect and their bounding boxes intersect, then one of them must be below the other, meaning that it is not on opposite sides- these conditions are sufficient as well.

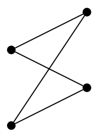
Both the methods `onOppositeSides` and `boundingBox` are $O(1)$, so the intersection algorithm is also $O(1)$. Moreover, the algorithm does not involve division.

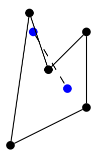
## 1.2 Simple Polygons and Convex Hulls

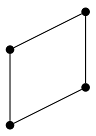
In this section, we will look at constructing different polygons given a set of points. These polygons we construct will have various applications that we will consider later. First, we consider some further geometric objects:

- a *path* is a sequence of distinct points $p_1$ to $p_n$.
- a *polygon* is a path where $p_1 = p_n$ (i.e. $p_1$ to $p_{n-1}$ are distinct, but $p_1 = p_n$);
- a *simple polygon* is a polygon which does not intersect itself- it is characterised by partitioning the space into its interior and exterior;
- a *convex polygon* is a simple polygon where for any two points $p$ and $q$ inside the polygon, the line segment $p - q$ is also inside it.

Below are some polygons that illustrate simple and convex polygons.


(a) A polygon that is not simple. The polygon crosses itself at the middle.

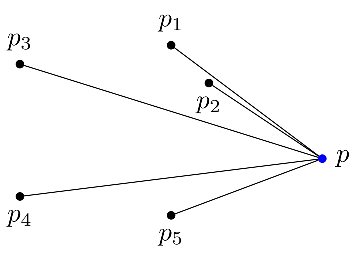

(b) A simple polygon that is not convex. The dashed line is not contained within the polygon.

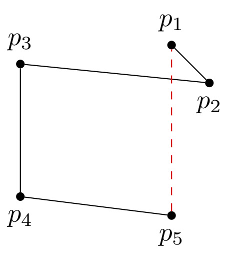

(c) A convex polygon.

### Simple Polygon

We will now consider how to construct a simple polygon given a set of points. In the example above, we saw that it is not possible to construct a simple polygon by arbitrarily joining the set of points. So, we will need to reorder the points in the set so that the resulting set of points can be joined together pairwise to get a simple polygon.

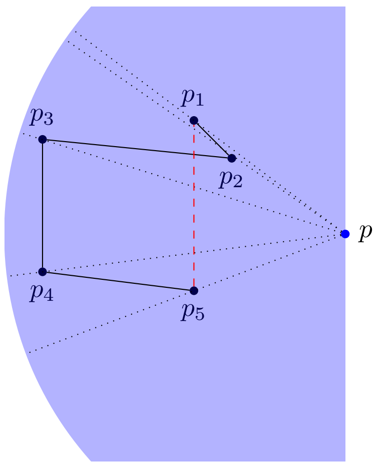
To do so, we can choose a point on the plane as a pivot. Then, we scan the set of points in an anti-clockwise manner from the pivot to find the order in which we should construct the simple polygon. We illustrate this with an example.





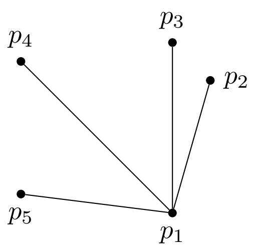
On the left, we use the pivot $p$ to scan and label the points going in an anti-clockwise fashion. That way, when we join the points $p_1 - p_2 - \cdots - p_5$, we do not have any self-intersection. However, when we join $p_5$ and $p_1$, we do get a self-intersection.

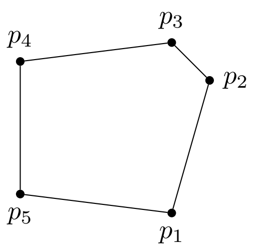
We can understand why there is a self-intersection using the figure below.



The image represents a semi-circle centered at the pivot $p$. When we label each point, we partition the semi-circle into different sectors. And, when we connect two points, we go from one sector to another. If we exclude the last connection, there is only one line from one sector to another.

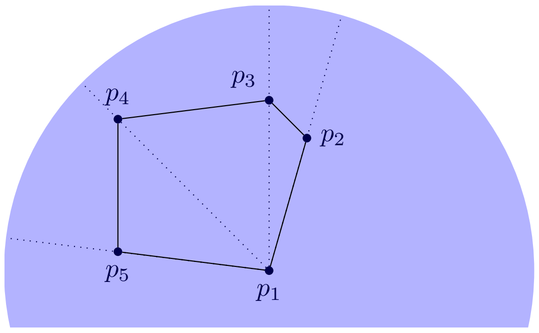
To avoid the intersection, we should choose the pivot as one of the points that we were given. The pivot is an extreme point- one way to choose the point would be to take the one with the highest $x$-coordinate (and lowest $y$-coordinate, if necessary). We first illustrate this with an example.





Here, the pivot was $p_1$. Using this pivot, we labelled the other points depending on when we encountered it with respect to the pivot, turning anti-clockwise. As we can see, this does not lead to a self-intersection at the end.

We will now use a similar diagram to above to show that this method always works.



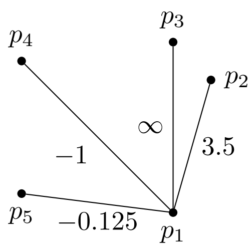
Like before, we do not expect there to be an intersect going from a point to a non-final point. Moreover, at the end, we are traveling along the radius- this ensures that we do not move to another sector and get a self-intersection.

Next, we consider the algorithm. To label the points, we sort them with respect to the angle it makes with the pivot- the smaller the angle, the earlier it is placed. The following is the algorithm:

```
void simplePolygon(List<Point> points) {
    // find the pivot
    int i  = 0;
    Point pivot = points[0];
    for (int j = 1; j < points.length; j++) {
        // Change the pivot if this point has x-coordinate bigger
        // or if this point has the same x-coordinate, and the
        // y-coordinate is smaller.
        if (pivot.x < points[j].x ||
            (pivot.x == points[j].x && pivot.y < points[j].y)) {
            pivot = points[j];
            i = j;
        }
    }

    // move the pivot to the start
    points[i] = points[0];
    points[0] = pivot;

    // list of angles and distances from the starting point
    List<double> angles = List.filled(points.length, null);

    // sort the points wrt angle it makes with the pivot
    points.sort((p1, p2) {
        double angle1 = angles[p1] ??
                        atan2(pivot.x-p1.x, pivot.y-p1.y);
        angles[p1] = angle1;

        double angle2 = angles[p2] ??
                        atan2(pivot.x - p2.x, pivot.y - p2.y);
        angles[p2] = angle2;

        return angle1.compareTo(angle2);
    });
}
```

The function `atan2` takes in the numerator and the denominator of the `arctan` function as two different arguments and gives the right `arctan` value, e.g. if the denominator is 0 or if the angle is greater than $\pi$.

We now consider the complexity of the algorithm- if there are $n$ points, then the loop to compute angles is $O(n)$, and the sorting is $O(n \log n)$. Therefore, the total complexity is $O(n \log n)$. So, this is a very efficient algorithm. It does use the `arctan` function, which is considered expensive. Moreover, there is no reason to explicitly compute the angles. We will now illustrate how to label the points without explicitly compute the arctangent value. So, consider the previous pivot labelling:



The labelled numbers are the gradients of the line going between $p_1$ and $p_i$ for $2 \leq i \leq 5$. So, given two gradients $m_1$ and $m_2$, we compare them as follows:

- if $m_1$ and $m_2$ have different signs, then the positive one comes first;
- if $m_1$ and $m_2$ have the same sign, then the smaller one comes first.

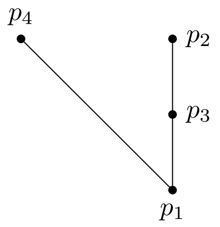
Using this pattern, we can sort the values by just comparing the gradients. This can be done without any division too, i.e. if we have

$$m_1 = \frac{dy_1}{dx_1}, \qquad m_2 = \frac{dy_2}{dx_2},$$

then we can cross multiply to find out whether $m_1$ should come before $m_2$- we need to pay attention to the sign of the values being multiplied to return the correct result.

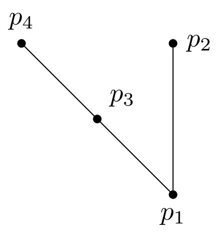
Although the algorithm works in most cases, it does not consider the following exceptional cases:

- If all the points are collinear with respect to the pivot, then there is no enclosing polygon.
- If points are collinear with respect to the pivot, they can be sorted in any order. However, we would require them to be sorted in the 'right' order. For instance, consider the following example:



Here, the pivot is $p_1$. Since $p_2$ and $p_3$ make the same angle with the pivot, we would need to choose the right order- the current ordering does not give us a simple polygon since $p_3$ should be the top right point. We mitigate this by further sorting the points by the distance it makes with the pivot.

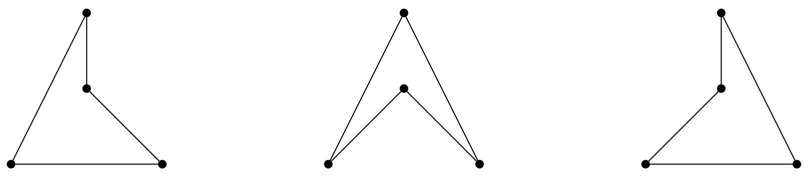
- If the final points are collinear to the pivot, then sorting them by increasing distance from the pivot does not work. For instance, consider the following example:



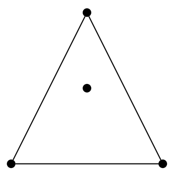
Here, the pivot is $p_1$. Since $p_3$ and $p_4$ make the same angle with the pivot, we would sort $p_3$ and $p_4$ in the order given above by the observation we made in the previous point. However, this does not work when the point is getting connected to the pivot. In that case, we would need to reverse these points, i.e. sort them with respect to the largest distance with the pivot, not the smallest.

## Convex Hull

Now, we will consider the convex hull problem. For a set of points on the plane, the convex hull is the smallest convex polygon that contains all the points. We measure the size of the polygon by its perimeter. So, we expect some of the points to lie on the boundary of this polygon. It is possible that not all of them lie on the boundary, as shown in the figure below.



In the figure above, we have 3 simple polygons from the same set of points. These are the only simple polygons containing all the points. Clearly, none of these are convex polygons. In fact, the convex hull of these 4 points is the following:

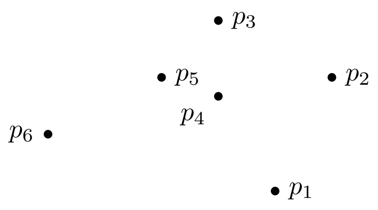


In a given set of points, the point that has the smallest or the largest $x/y$-coordinate will always lie on the convex hull. This implies that the choice of pivot from the simple polygon algorithm above will still work in this case.

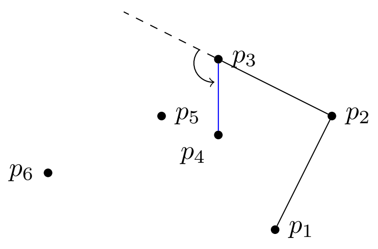
The algorithm for constructing the convex hull is the Graham Scan algorithm. It starts by running the simple polygon algorithm to sort the set of points with respect to the angle they make with the pivot. We then add these points in order to the convex hull, and remove any points as we realise that shouldn't lie on it. One way of finding out whether a point should be included is to:

- choose the pivot to be the point with the largest $x$-coordinate (and smallest $y$-coordinate); and
- choose to keep a point $p_i$ if, when adding the next point $p_{i+1}$, the triplet $(p_{i+1}, p_i, p_{i-1})$, the anti-clockwise angle between the two lines is less than $\pi$.

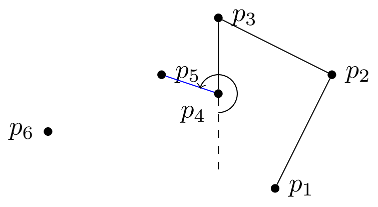
We illustrate the procedure with an example. Consider the following set of points.



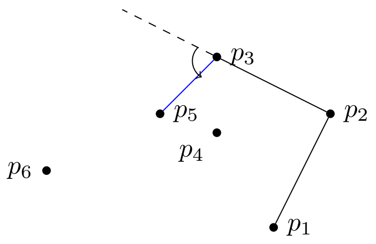
We have already applied the simple polygon algorithm to label the points with respect to the pivot $p_1$. We first connect the first three points- it is guaranteed that the angle between $-p_1 - p_2-$ and $-p_2 - p_3$ is less than $\pi$, since the angles have been sorted in increasing order with respect to $p_1$.



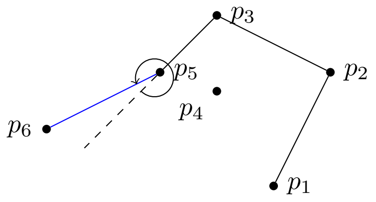
The next point to add is $p_4$. The angle between the two lines $-p_2 - p_3-$ and $-p_3 - p_4-$ is less than $\pi$, so we do not remove $p_3$.



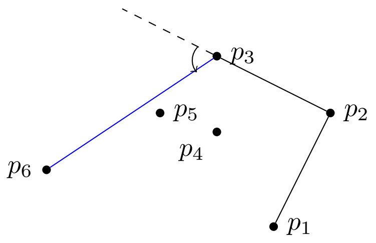
The next point to add is $p_5$. However, the corresponding angle is clearly bigger than $\pi$, so we will first need to discard $p_4$.



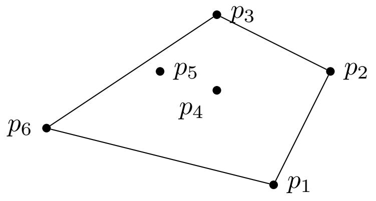
Now, the angle is less than $\pi$. So, we add $p_5$ to the convex hull.



This angle is also more than $\pi$, so we have to discard $p_5$ too.



The angle the lines $-p_2 - p_3-$ and $-p_3 - p_6-$ make is less than $\pi$, so $p_6$ gets added to the convex hull. Since it is the last point, we have constructed the convex hull:



We are guaranteed to make the right angle when we connect the last point with the pivot, as long as the pivot belongs to the convex hull.

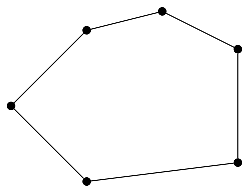
The algorithm for the Graham Scan is given below.

```
List<Point> grahamScan(List<Point> points) {
    simplePolygon(points);
    List<Point> cHull = [points[0], points[1], points[2]];

    // add all the other points to the list => remove
    // if the last 3 form a left angle
    for (var i = 3; i < points.length; i++) {
        while (angle(cHull[-2], cHull[-1], points[i]) > pi) {
            cHull.pop();
        }
        cHull.add(points[i]);
    }

    return cHull;
}
```

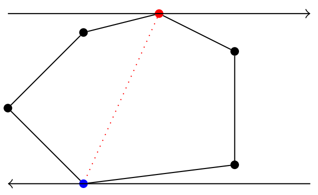
Now, we consider the complexity of the algorithm. The algorithm uses simple polygon construction, which has $O(n \log n)$ complexity. We can check whether the turn is anti-clockwise in $O(1)$ time. Now, the while loop within the for loop could imply that the complexity is $O(n^2)$. However, since a point is eliminated at most once from the list, the complexity of the entire loop is $O(n)$. Hence, the algorithm has complexity $O(n \log n)$.

Next, we will consider some applications of the convex hull algorithm. A basic usage might be for a robot to avoid some obstacles, while taking the shortest path. That is, the robot is at position $A$, wants to get to position $B$, and there are some obstacles in the middle. Then, we consider the set of points containing the obstacles and the positions $A$ and $B$. We construct a convex hull for this set of points. The shortest path avoiding the obstacles will then be either the clockwise or the anti-clockwise path from $A$ to $B$.

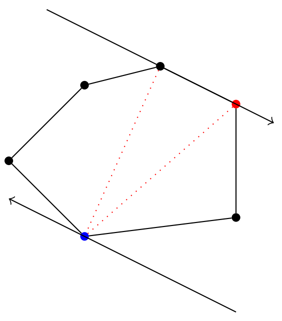
Another application is to compute the furthest pair of points in a set of points. A naive method would check each pair of points- that is $O(n^2)$. However, we can use the convex hull and compute the furthest pair of points in $O(n \log n)$ time- this is because the convex hull contains the pair of points that are furthest from each other.



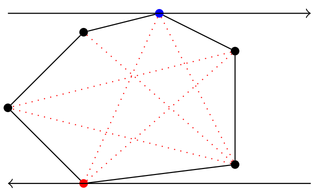
We first select 2 points- one that has a maximum $x$-coordinate, and a minimum $y$-coordinate.



We keep track of the distances between any two points we encounter. We rotate in clockwise direction, until we hit another point.



We continue this until the two lines have flipped.



Then, we choose the pair such that the distance is the longest- this is the longest distance of the entire set of points, not just the convex hull. This algorithm is called the rolling calipers.

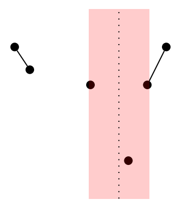
## 1.3 Closest Pairs

In this section, we will look at how to find the closest pair of points from a set, using a divide-and-conquer method.

We are given a set of points, and we want to compute the closest pair of points in the set. A naive algorithm would go through each pair of points and find the minimum distance- this is $O(n^2)$. We will use a divide-and-conquer method to improve the complexity to $O(n \log n)$.

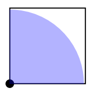
The closest pairs algorithm starts by sorting the given points with respect to the $x$-coordinate. We will then divide the set into 2 equally-sized subsets. This is the divide step. We will compute the closest pair of points in both subsets, and then use that to find the closest pair of points in the entire set. This is the conquer step.

We will now look at the combining step. Assume that the two subsets are $P$ and $Q$, and we have a closest pair of points $(p_1, p_2)$ in $P$ of distance $d_1$, and a closest pair of points $(q_1, q_2)$ in $Q$ of distance $d_2$. Let $d$ be the minimum of $d_1$ and $d_2$. Now, the closest pair of points in entire set is either going to be $(a, b)$, which has distance $d$ and both the points are from either $P$ or $Q$, or it will be $(x, y)$, with $x \in P$ and $y \in Q$, where the distance between the points $x$ and $y$ is strictly less than $d$. Hence, we only need to look at points within $d$ of the the midpoint separating $P$ and $Q$. This is shown in the figure below:



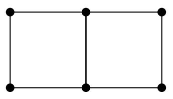
The left subsection is $P$, and the right subsection is $Q$. The closest pair of points in $P$ and $Q$ are also shown. The pink strip represents points that are within $d$ of the the midpoint, with respect to the $x$-coordinate. So, if there are two points $(x, y)$ with $x \in P$ and $y \in Q$ and distance between $x$ and $y$ less than $d$, then they both must lie on the pink strip.

It is possible that most of the points lie on the pink strip, so the algorithm would still be $O(n^2)$ overall. We will improve the algorithm by sorting the coordinates with respect to the $y$-coordinate as well. This means that we can just look at the points that come above a point on the pink strip. Moreover, we only need to look at a constant number of points above the point. We will illustrate this with a figure:



Above, we have a square of size $d \times d$, where $d$ is the smaller distance between the closest pair of points in $P$ and $Q$. We have further restricted the strip above, since a $y$-coordinate greater than $d$ cannot have distance smaller than $d$. If we have a point at the left corner of the square, we find that any point on the blue arc will have distance less than $d$ both in $P$- this is a contradiction.

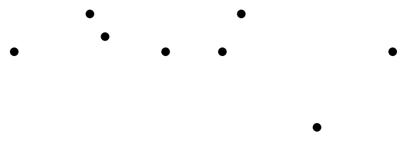
Hence, if there is another point in $P$ lying on the square, it must be outside of the blue arc. In the worst case, we will only have 4 points in $P$- one at each of the corners. When we extend this to a $2d \times d$ rectangle for both $P$ and $Q$, this corresponds to the following figure:



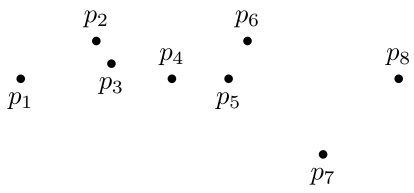
So, for a given point, we just need to consider 5 points on the strip that are above the point.

To check the points, we still have to sort the points with respect to the $y$-coordinate during the combining stage. This is possible by sorting the points every time, but that would lead to the complexity $O(n \log^2 n)$. We can do better by implicitly applying merge sort within the combination phase. So, we assume that the points in $P$ and $Q$ are also sorted by the $y$-coordinate. So, we can then merge the two lists during the combination phase.

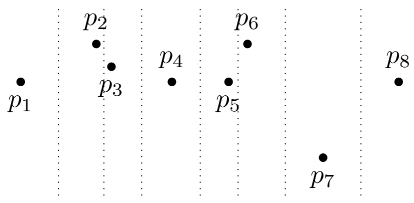
We will now illustrate the algorithm with an example. So, assume that we are given the following points.



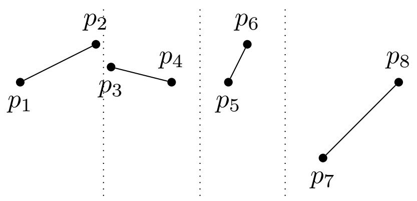
First, we sort the points with respect to their $x$-coordinate. We will label them using this order.



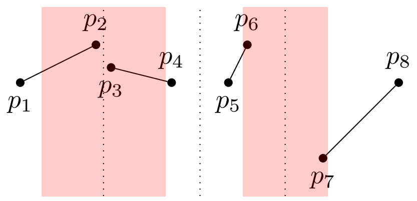
Next, we split the set into subsets- we will do this until each subset has just one point.



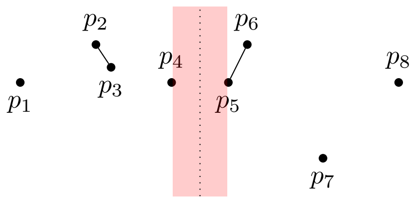
At the base case, the shortest pair of points will be the two points that we are given.



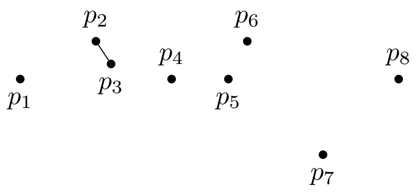
Right now, we have 4 closest pair of points. We will combine two pairs to get just 2 closest pairs.



In the left case, the points $p_2$ and $p_3$ lie in the strip. Clearly, the distance between $p_2$ and $p_3$ is smaller than the minimum distance, so we make that the shortest distance in the left subsection. In the right subsection, the points $p_6$ and $p_7$ lie in the strip, but the distance between them is longer than the minimum distance of the closest pairs. Hence, the minimum distance is the one between $p_5$ and $p_6$.



We shall now combine the two subsections to find the closest pair in the entire section. In this case, no points lies on the strip, so it is not possible to improve the distance.



Hence, the closest pair of points is $(p_2, p_3)$.

The following is the algorithm to find the distance between the closest pair of points:

```
double closestPair(List<Point> points) {
    // sort wrt x-coordinate
    points.sort((Point p1, Point p2) => p1.x.compareTo(p2.x));
    // run the closest pair recursive algorithm
    return sqrt(closestPairRec(points, 0, points.length-1));
}

double closestPairRec(List<Point> points, int i, int k) {
  // equal index => base index; maximum possible index
  if (i == k) {
    return double.infinity;
  }

  int j = (i + k) / 2;
  double mid = (points[j].x + points[j+1].x) / 2;

  int d1 = closestPairRec(points, i, j);
  int d2 = closestPairRec(points, j+1, k);
  int d = min(d1, d2);

  // sort w.r.t y-coordinate (assuming each sublist is sorted)
  merge(points, i, j, k);

  // filter points lying on the strip
  List<Point> filteredPoints = filterPoints(
    points, i, k, sqrt(d), mid
  );

  // Improving best distance using the filtered points
  for (int a=0; a < filteredPoints.length; a++) {
    for (int b=a+1; b<=min(a+5, filteredPoints.length-1); b++) {
      int newD = Point.d2(filteredPoints[a], filteredPoints[b]);
      if (newD < d) {
        d = newD;
      }
    }
  }

  return d;
}
```

The function `Point.d2` computes the squared distance between two points. The `merge` function merges the `points` array between indices `i` and `k`, assuming it sorted between `i` and `j`, and `j` and `k`.

Next, we consider the complexity of the closest pairs algorithm. Let $f(n)$ be the complexity of the algorithm when we are given $n$ pairs. The initial sorting takes $O(n \log n)$. Then, we have a divide-and-conquer segment, within which we make 2 recursive calls to sublists, and extra $O(n)$ time to process the shortest closest pair and merge the coordinates with respect to the $y$-value. Hence,

$$f(n) \leq O(n \log n) + 2f(n/2) + O(n), \qquad f(1) = O(1).$$

Without loss of generality, assume that $n = 2^k$.[2] Then, we can simplify the expression as follows:

$$\begin{aligned} f(n) &\leq O(n \log n) + 2f(n/2) + O(n) \\ &\leq O(n \log n) + 2 \cdot 2f(n/4) + O(n) + O(n) \\ &= O(n \log n) + 2^2 f(n/4) + 2O(n) \\ &\dots \\ &\leq O(n \log n) + 2^k f(1) + kO(n) \\ &= O(n \log n) + O(n) + O(n \log n) = O(n \log n). \end{aligned}$$

[2]We want to study the asymptotic behaviour of $n$, so if $n$ is not of the form $2^k$, we can just increase the value of $n$ to the next power of 2.

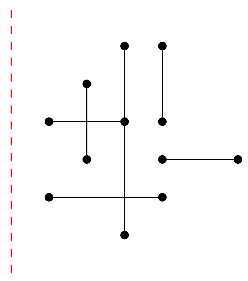
## 1.4 Vertical and horizontal lines

In this section, we will be considering how to find all the points of intersections between a set of horizontal and vertical lines.

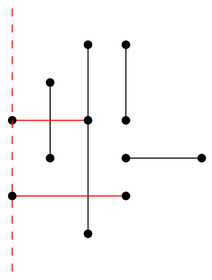
We are given a set of $h$ horizontal and $v$ vertical line segments in the plane, and we need to compute all the intersections. The naive method takes $O(hv)$ time. It is not possible to improve this complexity all the time- we might have that all the horizontal and the vertical line segments intersect, i.e. $hv$ intersections. We will aim to find an algorithm that has better time complexity when there are only few intersections- it has complexity $O(n \log n + p)$, where $n = h+v$ and $p$ is the number of intersections. This is the line-sweep algorithm.

In the algorithm, we represent a horizontal line (segment) by both of its endpoints, and represent a vertical line by one of its endpoints. So, we have a list of $2h + v$ points. Next, we sort this list with respect to the $x$-coordinate- this takes $O(n \log n)$ time. We will assume that no two horizontal lines, or vertical lines, intersect each other- we can just combine them into a single horizontal/vertical line so that they don't intersect. We will then scan left to right, keeping track of horizontal lines whose start we have encountered but not the end, and then check whether it intersects with a vertical line when we encounter one.

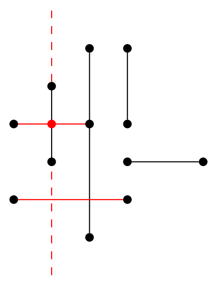
We will now illustrate the algorithm. Assume we are given the following lines. We will illustrate the algorithm using a sweep line. At the start, it is left to all the points.



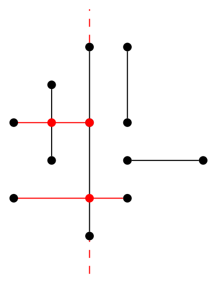
We move the sweep rightwards. First, we encounter two horizontal points.



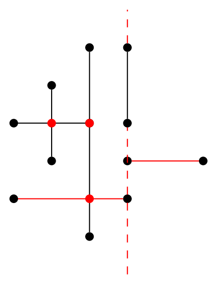
So, we add 2 horizontal lines to the candidate set. We move the sweep again.



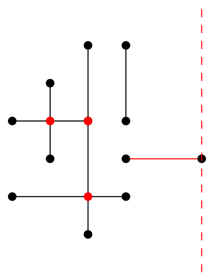
We now encounter a vertical line. So, we check whether any horizontal lines in the candidate set have $y$-coordinate between the start and the end of the vertical line. In this case, only the top horizontal line satisfies this. We move the sweep forward.



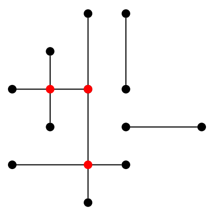
We now encounter the end of a horizontal line and a vertical line. We count the intersection between the ending horizontal line and the vertical line. So, the ending horizontal point should come after the vertical line in the sorted list so that the intersection counts. After the intersection, we remove the horizontal line from the candidate set. We move the sweep further.



We now encounter another vertical line. We count the intersection between the starting horizontal line and the vertical line. So, the starting horizontal point should come before the vertical line in the sorted list. This means that the horizontal line gets added to the candidate set so that it can be considered when we search which lines the vertical line intersects. We move the sweep to the right.



We now encounter the end of the horizontal line. So, we remove it from the candidate set, and we have come to the end of the sorted list. Hence, we have found all the intersections. These are the following:

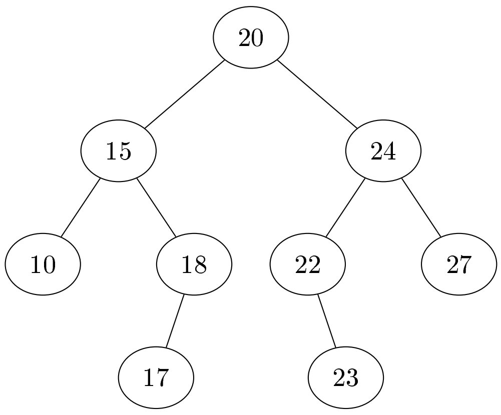


When we encounter a vertical line in the list, we want our algorithm to efficiently compute the intersections it has with the candidate set- it is possible for all the horizontal line segments to be candidates for this intersection, so we cannot use a linear scan to find all the intersections and for the algorithm to be better than the naive algorithm.

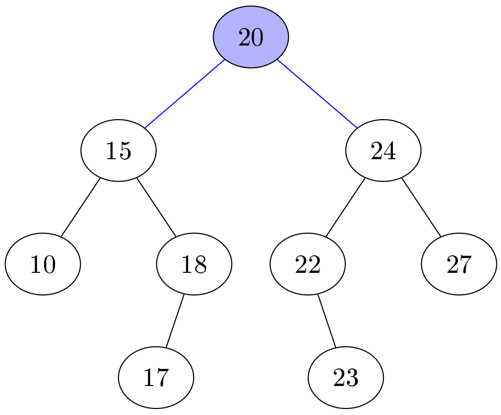
Instead, we need to use a binary search algorithm. To do so, the list of candidates points must be represented in a way that supports range searching- we need to find all those lines in the candidate set such that their $y$-coordinate is between the start and the end of the vertical line. Moreover, we need to be able to efficiently add and remove entries from the data structure. So, a good choice for this would be AVL trees.

AVL trees are binary search trees. They are balanced, which means that for any subtree, the length of the left and the right child differs by at most 1. So, the height of the tree is $O(\log h)$, where $h$ is the number of nodes in the tree. Hence, search, insertion and deletion are $O(\log h)$. Moreover, a range search is $O(k + \log h)$, where $k$ is the number of elements that lie in the range.

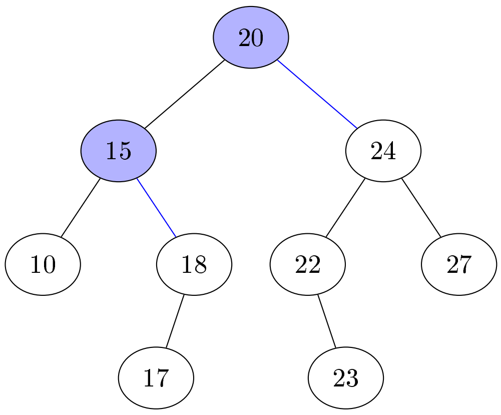
We illustrate range searching with an example. Assume we are given the following AVL Tree.



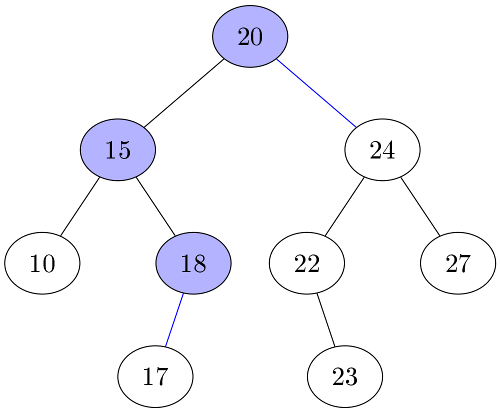
We will range search between 15 and 21. We start the search from the root.



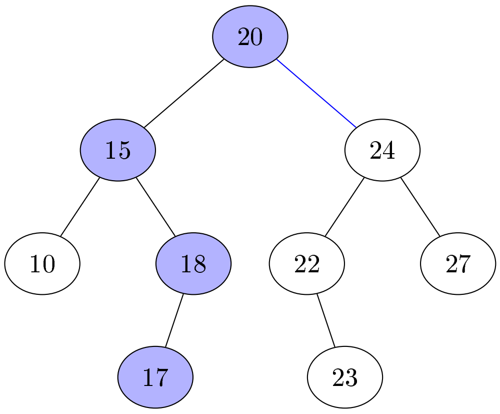
The root value is 20, so it lies within 15 and 21. So, we go down both the subtrees, starting with the left one.



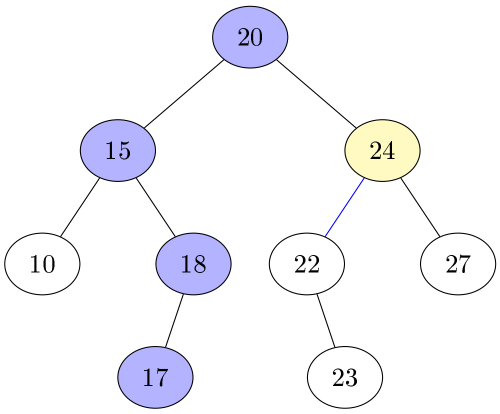
The value 15 is between 15 and 21. However, we only go down the right subtree since any number below 15 won't be in the range.



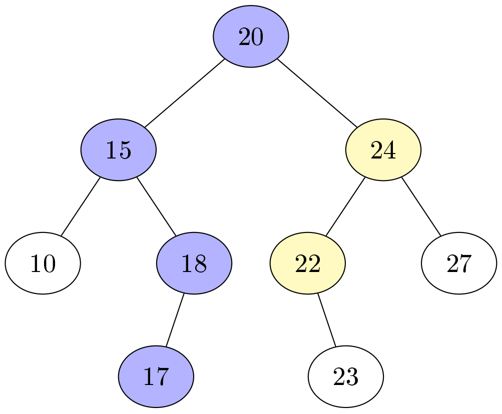
The next value is 18, which lies between 15 and 21. We go down both the left subtree, but there is no right subtree, so we cannot go down right.



The value 17 is between 15 and 21, but we cannot go down anymore. So, we now go back to the right subtree of the root.



The value 24 does not lie between 15 and 21. So, we just go down the left subtree.



The value 22 does not lie between 15 and 21, so we would go down the left subtree. However, there is no left subtree, so the range search is complete.

Below is the algorithm for range search in an AVL tree.

```
int rangeSearch(BNode<double> node, double start, double end) {
  if (node == null) {
    return 0;
  }

  int match = node.value >= start && node.value <= end ? 1 : 0;

  if (node.value <= start) {
    return match + rangeSearch(node.right, start, end);
  } else if (node.value >= end) {
    return match + rangeSearch(node.left, start, end);
  }

  return 1 + rangeSearch(node.left, start, end)
           + rangeSearch(node.right, start, end);
}
```

We will now justify the complexity of the algorithm. We need to find the $k$ values that lie in the range. Moreover, at each level, we need to visit at most 1 node out of the range to the left and the right, so it will take $O(\log h)$ complexity when traversing the values that don't lie in the range. IN total, the algorithm has $O(k + \log h)$ complexity.

Next, the algorithm to compute the number of intersections is the following:

```
int intersection(List<HLine> horizontals, List<VLine> verticals) {
  // get all the points, and sort them.
  List<Point> points = horizontals.map(hor => hor.start) +
                       verticals.map(ver => ver.start) +
                       horizontals.map(hor => hor.end);
  // need to take care of horizontal and vertical points with the
    same x-coordinate
  points.sort((Point p1, Point p2) => p1.x.compareTo(p2.x));

  // candidates will be sorted wrt y-coordinate
  BTree<HLine> candidates = BTree(
    (HLine l1, HLine l2) => l1.start.y.compareTo(l2.start.y)
  );
  int count = 0;

  for (int i=0; i<points.length; i++) {
    if (points[i].isHorizontal && candidates.has(points[i])) {
      candidates.remove(points[i]);
    } else if (points[i].isHorizontal) {
        candidates.add(points[i]);
    } else {
        count += candidates.rangeSearch(
          points[i].start.y,
          points[i].end.y
        );
    }
  }
  return count;
}
```

We can easily modify this to return the list of intersection points.

We will now consider the complexity of the line sweep. The list of points has length $2h + v = O(n)$, so when we sort them by the $x$-coordinate, it has $O(n \log n)$ complexity. At each horizontal point, we have to insert or delete it, which has complexity $O(h \log h)$. Moreover, at each vertical point, we need to do a range search, which has complexity $O(p + v \log h)$, where $p$ is the total number of intersections. Hence, the total complexity of the algorithm is $O(p + n \log n)$.