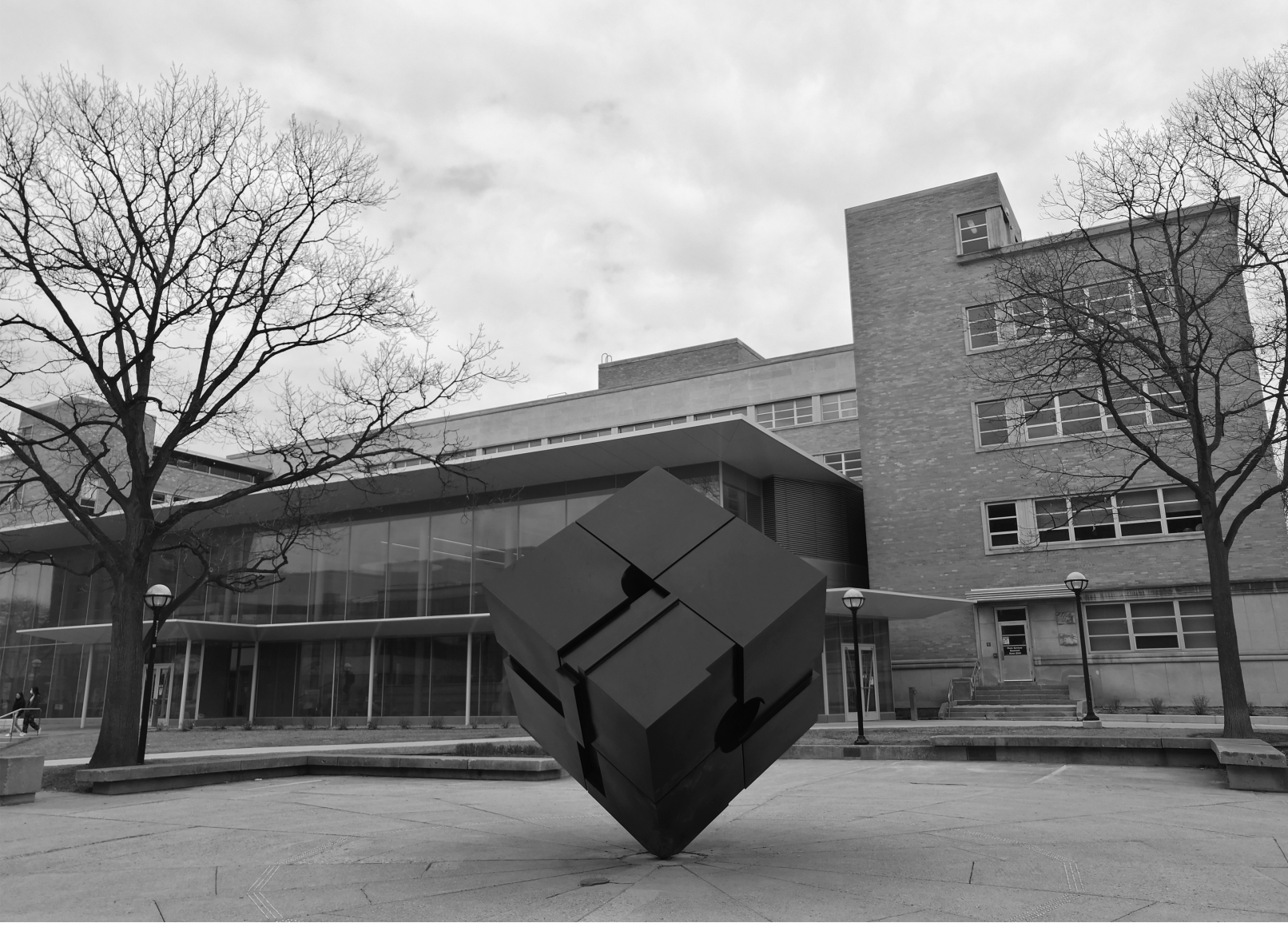
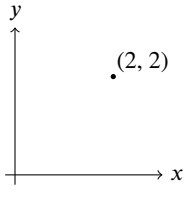


Chapter 26
Computational Geometry

26.1 Points, Segments, and Lines

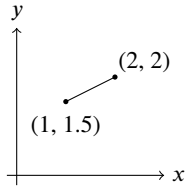
It is quite difficult to discuss any topic without referencing the everlasting presence of geometry in our multidimensional world, and the topic of algorithms is certainly no exception. In this chapter, we will introduce the field of **computational geometry**, which focuses on the design and analysis of algorithms for problems involving geometry, often with inputs and outputs of multiple dimensions. Computational geometry plays an important role in many fields of computer science, such as computer vision, image processing, modeling, robotics, and many more. Although we won't be able to delve into everything this field has to offer in EECS 281, we will still provide a high-level overview of several important computational geometry algorithms in the following sections.

To start off, we will look at how we can represent the simplest geometric components in a program. The first of these components is the **point**, which describes a single position in a dimensional space. Each point can be uniquely identified using a coordinate for each dimension that it resides in. For example, we identify each 2-D point using a pair of coordinates (x, y). In 3-D, we will need to add an additional coordinate to represent the position of a point along the third dimension, i.e., (x, y, z).



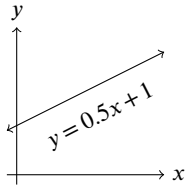

```
1  struct Point {
2    double x;
3    double y;
4  };
5
6  Point pt{2, 2};
```

A **segment** describes the portion of a line between two distinct points, and it can be uniquely identified by its two endpoints.




```
1  struct Segment {
2    Point pt1;
3    Point pt2;
4  };
5
6  Segment sg{Point{1, 1.5}, Point{2, 2}};
```

If we remove the endpoints of a segment and continue both ends infinitely in either direction, we would get a **line**. In two dimensions, a straight line can be uniquely identified using a slope and an y-intercept. The slope of a line is a measure of its steepness (i.e., $\Delta y/\Delta x$, or the change in y for each change in x), and the y-intercept is the value of y at which the line crosses the y-axis. Given a slope m and intercept b, all points (x, y) on the line must satisfy the equation $y = mx + b$.




```
1  struct Line {
2    double slope;
3    double intercept;
4  };
5
6  Line ln{0.5, 1};
```

It should be noted that a segment can also be used to define a line. Given two points (x_0, y_0) and (x_1, y_1), we can solve for the slope (m) and y-intercept (b) of the line that crosses those two points by using the following equations:

$$m = \frac{y_1 - y_0}{x_1 - x_0} \qquad\qquad b = y_0 - mx_0$$

However, different segments may be used to describe the exact same line. For instance, the segment from $(1,1)$ to $(2,2)$ identifies the same line as the segment from $(280,280)$ to $(281,281)$. In fact, there are an infinite number of segments we can choose to represent any given two-dimensional line (but only one slope and y-intercept).[1]

Example 26.1 You are given two straight 2-D lines in the form of a slope and a y-intercept. How would you determine if these two lines intersect, and what the intersection point is if they do intersect?

Let's denote the two lines we are given as $y = ax + b$ and $y = cx + d$, where a and b are the slope and intercept values of the first line and c and d are the slope and intercept values of the second line. First, we would check if the two lines have the same slope, which would indicate that they are parallel. If the two lines do have the same slope, then there are two cases that may arise depending on their intercept values:

1. If their intercepts are equal, then they are the exact same line (and thus they "intersect" at all points).
2. If their intercepts are not equal, then the two lines parallel each other and never intersect.

If the two lines do not have the same slope, then there must exist a single intersection point somewhere, which we will denote as (x_i, y_i). For the two lines to meet at (x_i, y_i), the following equations must be satisfied:

$$\begin{aligned} y_i &= ax_i + b \quad &&\text{(the point } (x_i, y_i) \text{ must be on line 1)} \\ y_i &= cx_i + d \quad &&\text{(the point } (x_i, y_i) \text{ must be on line 2)} \end{aligned}$$

Since the right-hand sides of both equations are equal to y_i, we can set them equal to each other, as shown:

$$ax_i + b = cx_i + d$$

Solving for x_i, we would get the following:

$$x_i = \frac{d - b}{c - a}$$

If we plug the value of x_i into the original equation $y_i = ax_i + b$, we would be able to compute y_i, and thus the intersection point.

Example 26.2 You are given two segments, each represented by the coordinates of its endpoints. How would you determine if these two line segments intersect?

To determine if two segments intersect, we will first compute the lines that the two segments reside on. If the two lines have the same slope but *distinct* intercepts, then the two segments cannot intersect (since they are parallel). If the two lines have the same slope but *equal* intercepts, we would then need to compare the coordinates of the segments' endpoints to determine if the segments themselves overlap. Otherwise, if the two lines have different slopes, we can use the formulas in example 26.1 to calculate the intersection point, and then check if the intersection point lies inside both of the given segments.

[1] We won't be worrying too much about lines in more than two dimensions in this chapter, since EECS 281 isn't a math class.

Example 26.3 You are given three points P_1, P_2, and P_3 on a 2-D plane. How would you determine if these three points all line on the same straight line? (Points lying on the same straight line are defined to be *collinear*.)

To determine if all three points lie on the exact same line, we can check if the lines formed by the segments P_1 — P_2 and P_2 — P_3 are the same. This can be done by using the endpoints of each segment to compute the slope and y-intercept of its corresponding line (via the equations covered previously). If the slope and intercept values of both segments are the same, then both segments lie on the same line, which implies that all three points must also lie on the same line.

Remark: Many of the algorithms that we will discuss in this chapter involve floating point comparisons. However, because of how they are represented in memory, comparing two floating point numbers using `operator==` or `operator!=` can be tricky due to imprecision. Instead, a better convention to is to check if two floating point values are *close enough* rather than exactly equal; if the two values are close enough within a reasonable error, then we consider them to be equal. This margin of allowable error is known as *epsilon*.

```
double a = ..., b = ...;

if (a == b) { /* do stuff */ }  // not ideal

constexpr double epsilon = 0.0001;
if (std::fabs(a - b) < epsilon) { /* do stuff */ }  // better
```

The same idea applies to comparisons. When checking whether one floating-point value is less than or greater than another, it is typically a good idea to add or subtract an epsilon value to handle any imprecision.

```
double a = ..., b = ...;

constexpr double epsilon = 0.0001;
if (a < b + epsilon) { /* do stuff */ }
if (a > b - epsilon) { /* do stuff */ }
```

The C++ `<limits>` library actually provides an epsilon value for floating point types. This value stores the difference between `1.0` and the next smallest floating-point value that can be represented using the floating point type `T`.

```
std::numeric_limits<T>::epsilon()
```

For `double` values, this epsilon value is 2.22×10^{-16}. Depending on the problem you are trying to solve, however, your optimal epsilon value may differ. For instance, if you are working with extremely large values and are relatively lax toward precision differences, a larger epsilon value may be ideal. On the other hand, if even the smallest differences can make a big impact on the problem you are trying to solve, a more fine-grained choice of epsilon may be a better option.

Example 26.4 You are given a vector of points on a Cartesian (x-coordinate, y-coordinate) plane. Each point's `.x` member variable is its x-coordinate, and its `.y` member variable is its y-coordinate (the definition of a `Point` object is provided below). Write a function that counts the number of **unique** rectangles that can be formed by these points. Do not count a rectangle more than once (e.g., a rectangle with points (A, B, C, D) is equivalent to a rectangle with points (D, A, C, B)). **Only consider rectangles whose sides are parallel to the x-axis and y-axis in your count.** All points in the `points` vector are unique, but they may be given in any order.

```
struct Point {
  int32_t x;
  int32_t y;
};
```

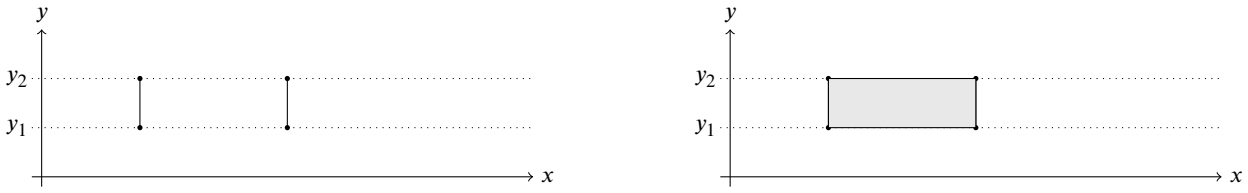
Example: Given the following points: $A = (1,1)$, $B = (2,1)$, $C = (3,1)$, $D = (3,2)$, $E = (2,3)$, $F = (1,3)$, and $G = (1,2)$, you would return `2`, since there are two unique rectangles that can be formed using these points ($ABEF$ and $ACDG$).

```
int32_t count_rectangles(const std::vector<Point>& points);
```

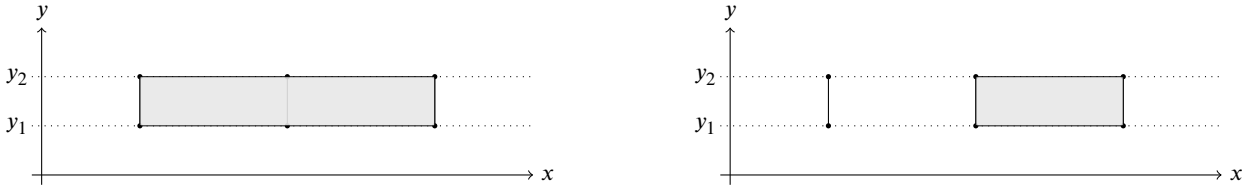
Hint: A rectangle is defined as a quadruple of points $(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)$ where $x_1 \neq x_2$ and $y_1 \neq y_2$. Rectangles can be described using either two vertical segments or two horizontal segments. When counting rectangles, any pair of vertical segments with the same pair of y-coordinates, or any pair of horizontal lines with the same pair of x-coordinates, forms a rectangle.

The key insight is to notice that a rectangle can only be formed if there is a pair of vertical segments with the same y-coordinates, or a pair of horizontal segments with the same x-coordinates, as described by the hint. We can use this observation to our advantage when solving this problem. To begin our approach, it may be helpful to look at an illustration; consider the following two points that form a vertical segment:

How can we determine the number of rectangles that can be formed using this vertical segment? For a rectangle to be constructed, there must exist another pair of points that form a segment with the same y-coordinates, as shown. In this case, we can identify a single rectangle that can be formed using the two segments.

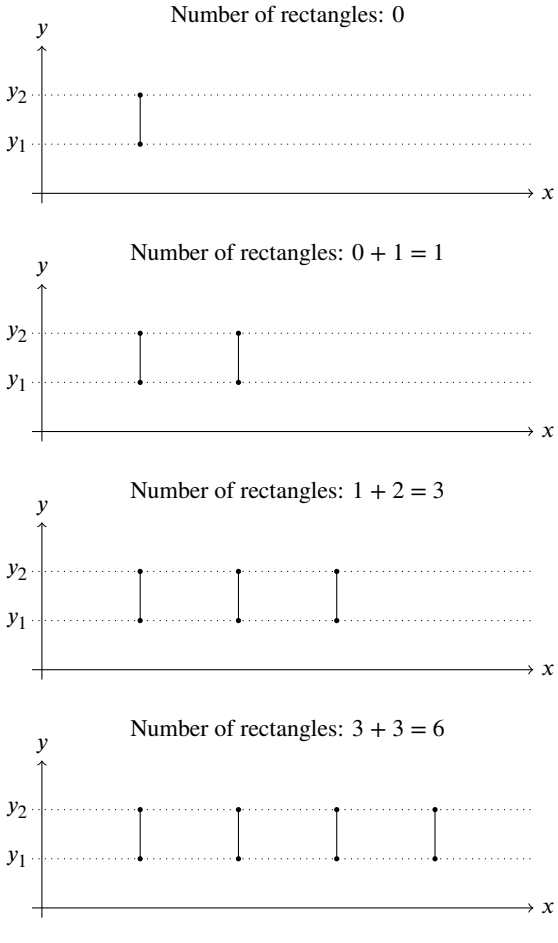


However, what if we encounter another vertical segment with the same pair of y-coordinates? How would that affect our solution? Notice that encountering a third segment ends up increasing the total number of possible rectangles from one to three! This is because this new segment can be used to construct two new, distinct rectangles: one each with the two vertical segments before it.



This observation can be extended with each new vertical segment we discover with the same pair of y-coordinates. If there was a fourth vertical segment with the same pair of y-coordinates, then we would be able to create three new rectangles: one each with the three segments that precede it. If there was a fifth vertical segment with the same pair of y-coordinates, we would be able to add four new rectangles, and so on. In general, encountering an n^{th} vertical segment with a given pair of y-coordinates would allow you to create $n-1$ brand new rectangles.

This idea forms the crux of our solution. To solve the problem, we will iterate over all the vertical segments that can be constructed using the given points (this can be done using a nested loop over the points to identify pairs of points that share the same x-coordinate). For each vertical segment we encounter, we increment a counter by the number of times we have seen another vertical segment with the same pair of y-coordinates. This is shown using an example below (all of the segments below share the same y-coordinates of y_1 and y_2 for simplicity, but this process also works when there are multiple vertical segments that may have different y-coordinates):



We can keep track of vertical segments with the same y-coordinates by using an associative container like a `std::unordered_map<>` or `std::map<>` to map each distinct pair of y-coordinates to the number of vertical line segments that share those coordinates. To avoid having to hash a custom object to keep things simple, we will implement our solution using a `std::map<>` of pairs, where each pair stores the y-coordinates of a vertical segment that can be formed using the given points (to avoid double counting, we will ensure that the smaller y-coordinate will always be first in the pair). An implementation of this solution is shown below:

```
1   int32_t count_rectangles(const std::vector<Point>& points) {
2     // maps each pair of y-coordinates to the number of distinct
3     // vertical segments that have those coordinates
4     std::map<std::pair<int32_t, int32_t>, int32_t> segments;
5
6     int num_rectangles = 0;
7     for (const Point& p1 : points) {
8       for (const Point& p2 : points) {
9         if (p1.x == p2.x && p1.y < p2.y) {
10          std::pair<int32_t, int32_t> vertical_segment{p1.y, p2.y};
11          // add the number of times this pair of y-coordinates was seen before
12          // to the count, then increment the value in the map
13          num_rectangles += segments[vertical_segment]++;
14        } // if
15      } // for p2
16    } // for p1
17
18    return num_rectangles;
19  } // count_rectangles()
```

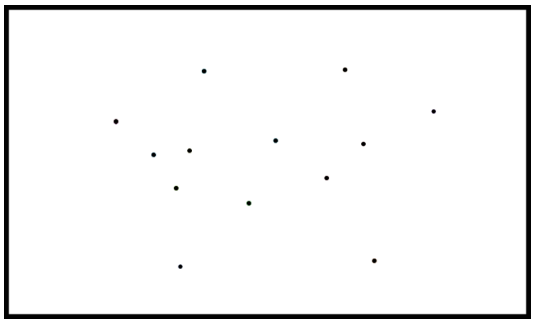
The time complexity of this solution is $\Theta(n^2 \log(n))$ for the number of points n, since we are performing a $\Theta(\log(n))$ map lookup within a nested $\Theta(n^2)$ loop (although it is possible to reduce this time complexity to $\Theta(n^2)$ if we use a hash table instead, but this would require us to implement a custom hashing function). The auxiliary space used by this solution is $\Theta(n^2)$ for the number of pairs we may have to insert into our map.

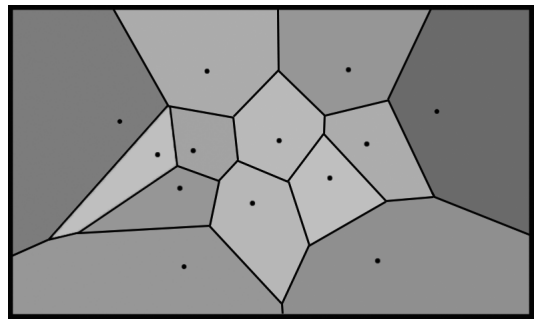
It should be noted that we could have counted the number of horizontal segments with the same x-coordinates to arrive at the same solution. Both methods accomplish the same task of identifying all the rectangles that can be formed, so we only needed implement one of the two. We will look at similar approaches toward solving geometry problems in the next section using a strategy known as the sweep line algorithm.

26.2 Sweep Line Algorithm (⁎)

※ 26.2.1 *Applications of the Sweep Line Algorithm* (⁎)

One of the most important algorithms in computational geometry is the **sweep line algorithm**. As its name implies, the sweep line algorithm solves a geometric problem by sweeping an imaginary line across a plane, stopping at certain points (or "events") to perform necessary computations for the given problem. Once the line has swept through all the objects on the plane, the solution to the problem is obtained. The sweep line algorithm can be used to design efficient implementations for many important computational geometry problems. Just as an example, one important application of the sweep line algorithm comes from the construction of Voronoi diagrams.[2] Given a set of n points on a plane, a *Voronoi diagram* partitions the plane into n convex polygons (often called "cells") that each represent regions of the plane that are closer to one of the given points than all others. An example Voronoi diagram is shown below on the right, for the given set of points on the left.





Voronoi diagrams are useful for a variety of fields, from biology to urban planning. For instance, a Voronoi diagram can be used to devise an algorithm for diverting airplanes to the closest airport in the case of an emergency. One of the most efficient algorithms for building a Voronoi diagram, *Fortune's algorithm*, uses the sweep line technique to accomplish this task in worst-case $\Theta(n \log(n))$ time, given a set of n points.

The Voronoi diagram is just one example of the sweep line algorithm's utility in computational geometry. We will not be discussing Voronoi diagrams and Fortune's algorithm any further in this chapter, since the knowledge required to formulate this algorithm is beyond the scope of the class. However, we will still look at other applications of the sweep line algorithm in this section, and how it can be used to solve simpler computational geometry problems.

[2] You are not responsible for knowing what this is for the class; this is just one example of an application of the sweep line algorithm.

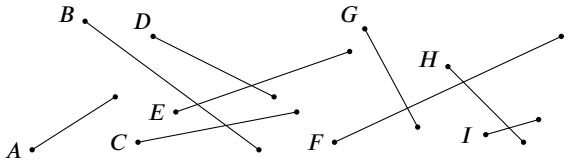
※ 26.2.2 *Solving Problems Using the Sweep Line Algorithm* (✻)

Example 26.5 You are given a collection of n line segments, each represented using the coordinates of its two endpoints. You may make the following assumptions about this collection of segments:

- No line segment is vertical.
- Any two segments intersect in at most one point.
- No three segments intersect at the exact same point.

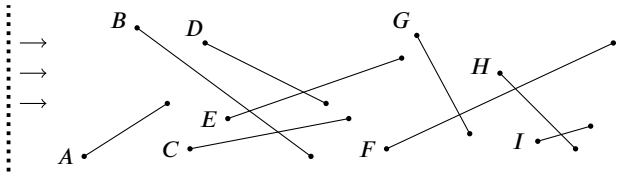
Devise an algorithm that returns the coordinates of all intersection points within this collection of segments.

A naïve approach would be to consider all pairs of line segments and compute if an intersection exists between each pair. However, this would require us to compare $\Theta(n^2)$ pairs of segments, and our algorithm would always run in $\Theta(n^2)$ time. Instead, we can do better if we utilize a sweep line approach. To illustrate how this works, consider the following example:

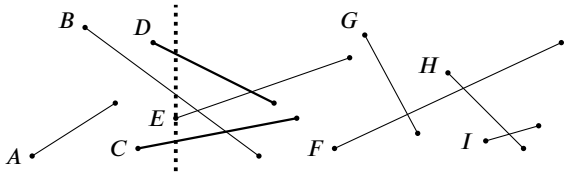


To start our algorithm, we will create a sweep line at the left edge of our plane. This line will then sweep rightwards, stopping only when it experiences the following events:

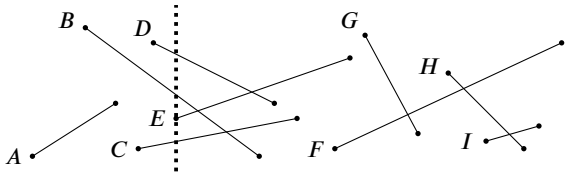
1. The sweep line encounters a brand new segment (by hitting a left endpoint).
2. The sweep line encounters the end of an active segment (by hitting a right endpoint).
3. The sweep line encounters the intersection of two segments.



A key observation to make here is that two segments can only ever intersect if they are adjacent to each other at the same horizontal position. For example, line segments C and D cannot physically intersect at the following position of the sweep line, since they are not adjacent (lines B and E are situated in between them). As a result, there is no need to check if an intersection exists between C and D at this horizontal position of the sweep line. A check only needs to be made if segments B and E terminate before C and D, or if an intersection occurs that changes the relative order of the four segments, since these are the events that could cause C to be adjacent to D (and thus raise the possibility of intersection).

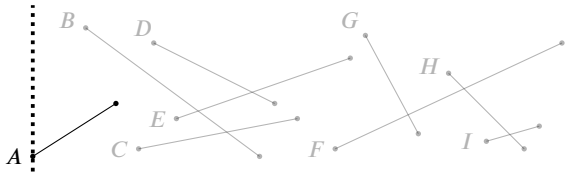


To track this information, we will need a container that can store the relative ordering of the active segments at any point along our algorithm, so that we can easily check if any two segments are adjacent. This can be done using a self-balancing binary search tree (in C++, a `std::set<>`) that stores the active segments at any point in time, ordered by the y-coordinate at which the segment encountered the sweep line. For example, at the position of the sweep line above, the segments in the container should be ordered D, B, E, and C (since this is the vertical order of the segments at this position).



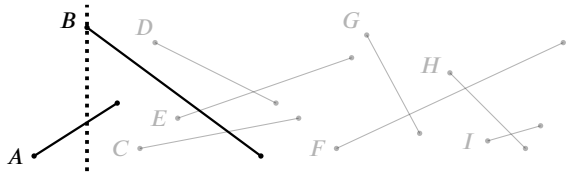
Active Segments	
D	(largest y-position)
B	
E	
C	(smallest y-position)

Let's look at the sweep line in action. We will sweep the line rightward, stopping only if any of the three events occurs (left endpoint, right endpoint, or intersection). At each event, we then check if the segment(s) adjacent to the event generate an intersection. In our example, the first event we encounter is the left endpoint of segment A. Since no other active segments exist at this position, A is inserted into our active segment container, and no intersection checks need to be made.



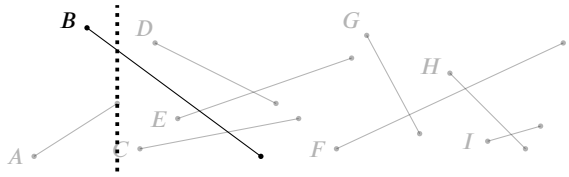
Active Segments	Discovered Intersections
A	

The next event we encounter is the left endpoint of segment B, which we add to the container of active segments — since B's left endpoint had a higher y-position when it encountered the sweep line, it is ordered above A in the container. Since B is a new segment and it is adjacent to segment A, we check if an intersection exists between segments A and B (using the process defined in example 26.2). In this case, A and B do not intersect, so we continue with our sweep.



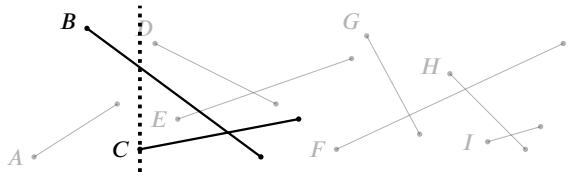
Active Segments	Discovered Intersections
B	
A	

The next event we encounter is the right endpoint of segment A. This indicates that A has terminated, so we can remove A from our container of active segments. B is the only segment remaining in the container, so no intersection checks need to be made.



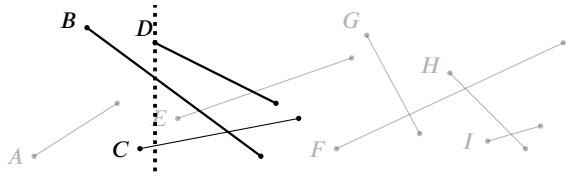
Active Segments	Discovered Intersections
B	

The next event we encounter is the left endpoint of segment C, which we add to our container of active segments. Segment C is adjacent to segment B, so we check if an intersection exists between these two segments. In this case, B and C *do* intersect, so we add the coordinates of the intersection point to our solution.



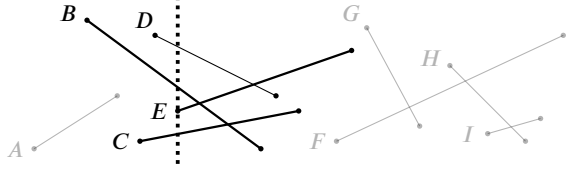
Active Segments	Discovered Intersections
B	(B,C)
C	

The next event we encounter is the left endpoint of segment D, which is added to our container of active segments. Segment D is adjacent to segment B, so we check if an intersection exists between these two segments. In this case, B and D do not intersect, so we continue our sweep.



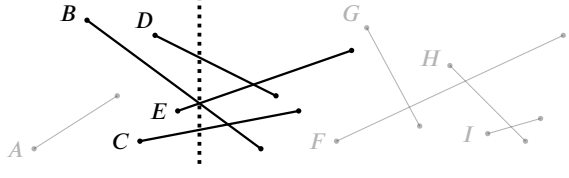
Active Segments	Discovered Intersections
D	(B,C)
B	
C	

The next event we encounter is the left endpoint of segment E, which is added to our container of active segments. Segment E is adjacent to both B and C, so we check if intersections exists between E and both B and C. E and C do not intersect, but E and B do; we therefore add the coordinates of this intersection point to our solution.



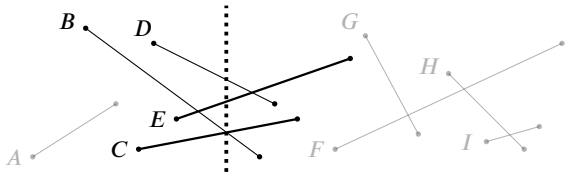
Active Segments	Discovered Intersections
D	(B,C)
B	(B,E)
E	
C	

The next event we encounter is the intersection of segments B and E. At this point, notice that segments B and E switch their vertical positions, so the relative order of these two segments must also be changed in our container of active segments. This swap also creates new adjacent segments: E is now adjacent to D, and B is now adjacent to C. Thus, we will need to check if an intersection exists between these pairs of segments. The intersection between B and C has already been discovered, but the intersection between D and E is brand new. We therefore add the coordinates of this new intersection point to our solution.



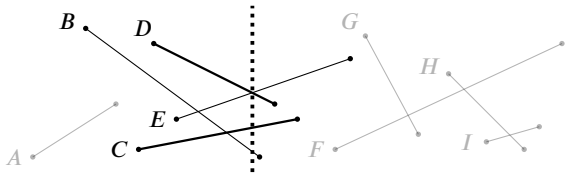
Active Segments	Discovered Intersections
D	(B,C)
E	(B,E)
B	(D,E)
C	

The next event we encounter is the intersection of segments B and C. These two segments switch positions in our active segments container, and any new adjacent pairs of segments are checked for intersection. In this case, C is now adjacent to E, but we already discovered before that C and E do not intersect.



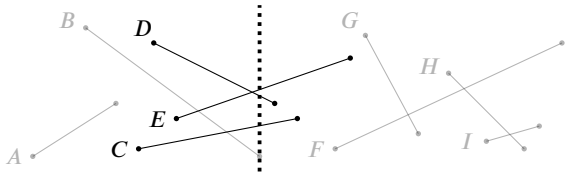
Active Segments	Discovered Intersections
D	(B,C)
E	(B,E)
C	(D,E)
B	

The next event we encounter is the intersection of segments D and E. These two segments switch positions in our container, and the new adjacent segment pair of C and D is checked for intersection. C and D do not intersect, so we continue with our sweep.



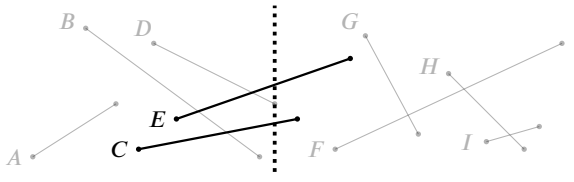
Active Segments	Discovered Intersections
E	(B,C)
D	(B,E)
C	(D,E)
B	

The next event we encounter is the right endpoint of segment B. This indicates that B has terminated, so we remove B from our container of active segments. We then check for intersection between any new pairs of adjacent segments. Since B was the bottom-most segment when it terminated, there are no new adjacent pairs, and no additional check is needed at this event.



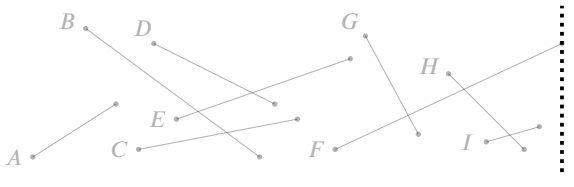
Active Segments	Discovered Intersections
E	(B,C)
D	(B,E)
C	(D,E)

The next event we encounter is the right endpoint of segment D. This indicates that D has terminated, so we remove D from our container of active segments. After D's removal, C and E are now adjacent, so we check if an intersection exists between these two segments. These two segments do not intersect, so we continue with our sweep.



Active Segments	Discovered Intersections
E	(B,C)
C	(B,E)
	(D,E)

If we follow this process for all of the remaining events, we will eventually discover every intersection point once our sweep line reaches the right endpoint of the final segment, and thus the solution to our original problem.



Active Segments	Discovered Intersections
	(B,C) (F,G)
	(B,E) (F,H)
	(D,E) (H,I)

At this point, you may be wondering: how do we instantiate a sweep line and move it across the plane? It turns out that we don't have to create a physical sweep line at all — the line we used in the example was just a visualization tool for understanding how the algorithm works. When it comes to implementing a solution, we do not need to keep track of a line at all; we only need to keep track of the *events* that we need to visit.

This can be done using a min-priority queue of events, where the priority of an event is determined by its horizontal position (i.e., x-coordinate). At the beginning of the algorithm, the priority queue is initialized with the endpoints of all the provided segments. During the algorithm, we also insert any new intersection points that are discovered. By using this priority queue approach, we can easily determine the next event to visit by popping off the element at the top of our priority queue, essentially emulating the process of sweeping a line across the plane.

To summarize, our algorithm to find all segment intersections is as follows:

1. Create a min-priority queue of events based on the x-coordinate of the event, and initialize it with the endpoints of all the given segments. This priority queue will allow us to visit all the events from left to right, akin to sweeping a line.
2. Initialize a self-balancing binary search tree (`std::set<>` in C++) that stores the active segments at the current position of the sweep line, ordered by y-position.
3. While there are still events to visit, pop the next event off the priority queue:
   - If the event is a segment's left endpoint:
     - Add the segment to the balanced binary search tree of active segments.
     - Check segments adjacent to the new segment on the sweep line for intersection.
     - If a new intersection point is discovered, add it to the solution and push it into the event priority queue.
   - If the event is a segment's right endpoint:
     - Remove the segment from the active segments container.
     - If two other segments become adjacent after the removal, check those two segments for intersection.
     - If a new intersection point is discovered, add it to the solution and push it into the event priority queue.
   - If the event is an intersection point:
     - Change the order of the two intersecting segments in the active segment container (this can be done by updating their y-coordinates so that their orderings are switched).
     - If the switch causes two segments to become newly adjacent, check those segments for intersection.
     - If a new intersection point is discovered, add it to the solution and push it into the event priority queue.
4. Once the event priority queue is empty and all segments have been processed, the solution is now complete with all segment intersections.

What is the worst-case time complexity of this algorithm? At every event, we complete the following steps successively:

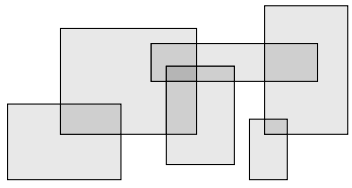
1. We pop from the priority queue (which takes worst-case $\Theta(\log(n))$ time).
2. We may have to check for segment intersection (which takes $\Theta(1)$ time using the strategy covered in example 26.2).
3. We push and pop from the active segment binary search tree (which takes worst-case $\Theta(\log(n))$ time).

Thus, the total work we need to do for each event is bounded by $\Theta(\log(n))$, which is the highest-order term. Since an event can either occur at the endpoints of a segment or an intersection point between two segments, the total number of events we encounter is $2n+k$ if there are n segments and k intersections among these segments. Because there are a total of $\Theta(n+k)$ events, and each event can take up to $\Theta(\log(n))$ time, the worst-case time complexity of the overall algorithm is $\Theta((n+k)\log(n))$.

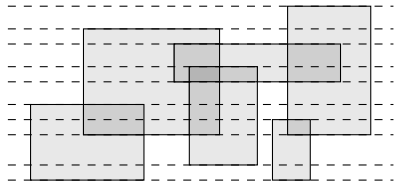
The algorithm we discussed is formally known as the *Bentley-Ottman algorithm*. Although we made some simplifying assumptions for our example, there are additional modifications we can make to support line segments that do not adhere to our initial restrictions. For instance, we can tiebreak two events with the same x-coordinate using their y-coordinates — this allows us to handle multiple events at the same x-coordinate, which in turn allows the algorithm to handle vertical line segments.

Example 26.6 You are given n rectangles whose edges are parallel to the x- and y-axes. Devise an algorithm that can be used to find the total area covered by their union. If more than two rectangles cover the exact same area, that area should not be counted more than once.

This is another problem that can be efficiently solved using a sweep line approach. Consider the following input:

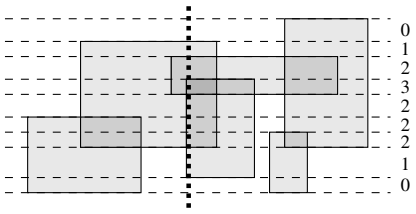


One important detail to notice is that any area covered by a union is bounded above and below by the y-coordinate of one of the given rectangles' horizontal edges. Because of this, we can partition our graph into several intervals using the y-coordinates of our rectangles, as shown:



Similar to the previous example, we start our sweep line at the left end of our plane and sweep it rightwards across the rectangles. For each of the intervals, we keep track of a counter that indicates how many rectangles exist in that interval at the current position of the sweep line. Every time the sweep line crosses a left edge of a rectangle, the intervals that are spanned by that rectangle have their counter(s) incremented; every time the sweep line crosses a right edge, the corresponding counter(s) are decremented. By following this rule, intervals that have a positive counter value at the position of the sweep line must be a part of a rectangle union.

For example, these are the values of the counters when the sweep line is at the following position:



One potential strategy for implementing this solution is as follows:

1. Initialize a running value that stores the area of the rectangle union encountered by the sweep line so far.
2. Sort the left and right edges of each rectangle in ascending order of x-coordinate (or use a priority queue, similar to the previous example). Each of these edges represents an event that our sweep line needs to stop at, and sorting them allows the algorithm to easily determine which event to stop at next.
3. "Sweep" the line through the plane by visiting each of the events one-by-one. At each event:
   - Iterate over the counters to determine the length of the sweep line that currently resides in a rectangle union (i.e., the combined length of intervals whose counter ≥ 1).
   - Multiply this length with the horizontal distance to the previous event (this gets us the total area that the sweep line visited between the previous and current events).
   - Add this result to the running total, and increment or decrement any counters as necessary.
4. After the sweep line reaches the rightmost edge among the given rectangles, the value of the running total is equal to the total area occupied by the rectangle union.

Remark: One data structure that is helpful for solving this problem is the *segment tree*, which is essentially a balanced binary tree that stores information about intervals, allowing you to efficiently query information for a given *range* of values. A segment tree would make it possible to query/update the counters and identify the length of the sweep line inside the rectangle union in $\Theta(\log(i))$ time, where i is the total number of intervals. Since the number of intervals i is on the order of the number of rectangles n, and the number of events is also on the order of n, the overall worst-case time complexity of finding the area of a rectangle union using a segment tree is equal to the number of events $n \times$ the time required for each event $\log(i)$, or $\Theta(n \times \log(i)) = \Theta(n \log(n))$. We will discuss segment trees in greater detail in a later section.

26.3 The Closest Pair of Points Problem

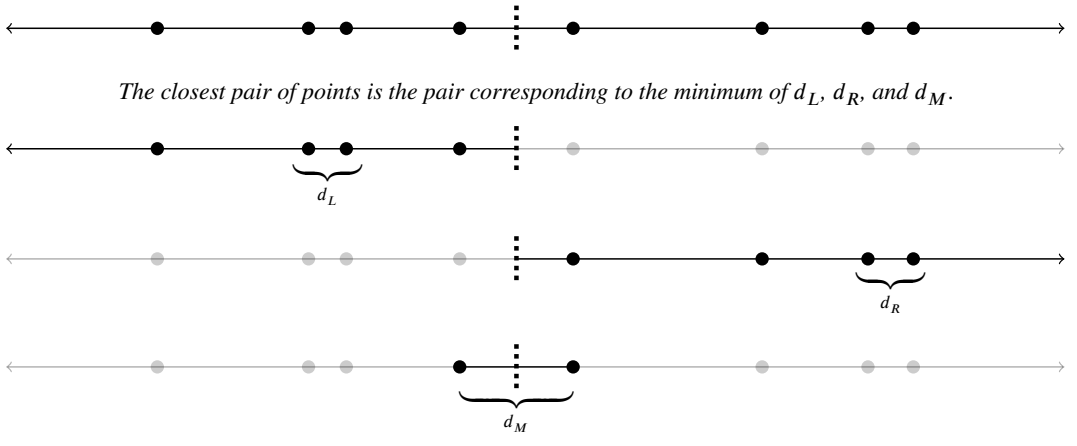
※ 26.3.1 *The Closest Pair of Points Problem in One Dimension*

Given a set of n points on a two-dimensional plane, how can you efficiently find the two points that are separated by the smallest distance? A naïve brute force algorithm would calculate the distance between all pairs of points and return the pair whose distance is minimal. However, the time complexity of this approach would be $\Theta(n^2)$ since there are $\Theta(n^2)$ pairs on the graph. Can we do better?

To solve this problem in better than $\Theta(n^2)$ time, we can use a divide-and-conquer approach. However, the idea behind this improved strategy may not be immediately clear at first. To understand how we can utilize divide-and-conquer to solve this problem, it is helpful to consider the same version of this problem, but on a one-dimensional line rather than a two-dimensional plane. If we are given a set of n points on a line, we can find the closest pair of points by following these steps:

1. Presort all the points in order by their coordinate position.
2. Find the median of all points and partition the points into two halves based on this median.
3. Recursively find the closest pair of points to the left of the median.
4. Recursively find the closest pair of points to the right of the median.
5. Find the closest pair of points that either cross the median (i.e., the rightmost point to the left of the median and the leftmost point to the right of the median) or include the median point (if there is one).

After we recursively compute the closest pair of points (1) to the left of the median, (2) to the right of the median, and (3) among two points that cross the median, the closest pair among these three would also be the closest pair overall. (Notice that this is the same strategy we used to solve the maximum subarray problem using divide-and-conquer in chapter 21.)



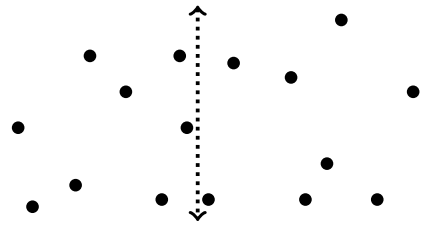
To implement this one-dimensional problem, we will need to first sort the points so that we can find the median; given n points, this step takes $\Theta(n\log(n))$ time if we use a standard comparison-based sorting algorithm. Then, we use divide-and-conquer to recursively split the input in half and compute the closest pair among the three options listed previously; finding this closest pair takes $\Theta(n)$ time if the points are sorted. We can thereby express the divide-and-conquer steps using the following recurrence:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 2 \\ 2T(n/2) + \Theta(n), & \text{if } n > 2 \end{cases}$$

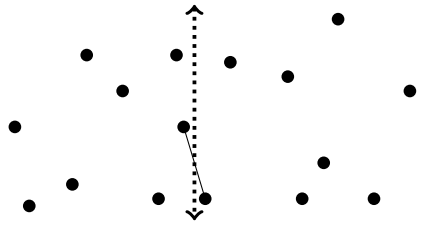
Applying the Master Theorem, we would see that the divide-and-conquer portion of the algorithm takes $\Theta(n\log(n))$ time. If we add this to the $\Theta(n\log(n))$ sorting step, we can conclude that the overall time complexity of the algorithm is also $\Theta(n\log(n))$. It is this approach that we will generalize to solve our original two-dimensional problem.

※ 26.3.2 *The Closest Pair of Points Problem in Two Dimensions*

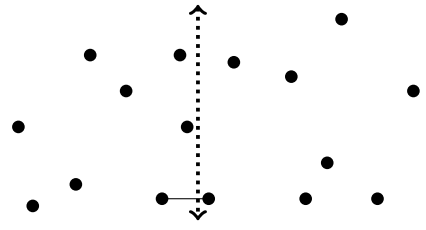
In the one-dimensional case, we were able to divide our points in half using the median *point*. If we move up to two-dimensions, we can achieve similar behavior by splitting the points in half using a median *line*. This is done by sorting all the given points in order by their x-coordinate, and finding the vertical line that lies on the median of these x-coordinates:



We then follow the same procedure as before: we recursively find the closest pair in the left half, the closest pair in the right half, and the closest pair that either crosses the median line or involves a point on the median line. Finding the closest pair to the left and right of the median line is rather straightforward, but finding the closest pair that crosses the median line is trickier. This is because the pair closest to the median line in the x direction may be far away in the y direction (as shown in the example).

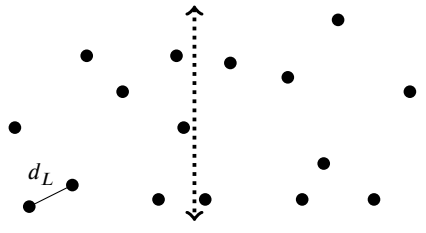


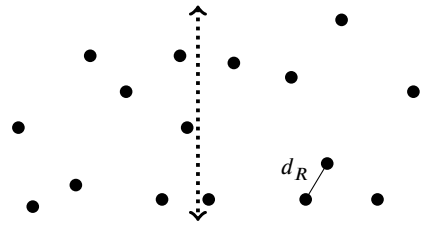
The two connected points above are the closest to the median line.



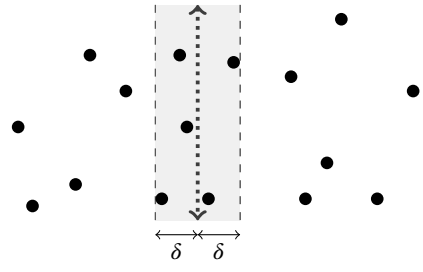
However, they are not the closest pair that crosses the median line!

Because of this, we need to consider *all* pairs that are close enough to the median line, rather than just the two points that are closest to the median line. But how do we define "close enough"? It turns out we can use the closest pair to the left and right of the median line to help us determine how far we need to look around the median line. Let us define d_L as the distance of the closest pair to the left of the median line, and d_R as the distance of the closest pair to the right of the median line, as shown.





Let us define the better of these two values as δ; that is, $\delta = \min(d_L, d_R)$. When looking for pairs that cross the median line, we can ignore any point that is further than δ away from the median line. Why? If a point is more than δ away, the distance required to connect it to a point on the other side of the median line must be worse than either d_L or d_R, and thus cannot be optimal! This is represented by the shaded region below, which we will refer to as the δ-strip.



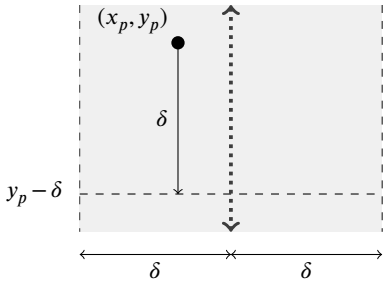
How should we examine the δ-strip? A naïve approach would compute the distance between all pairs of points that cross or involve a point on the median line. However, in the worst case, every single point could reside in the δ-strip! In such a scenario, we would have to compare all pairs of points in the plane. As discussed earlier, comparing all pairs of points takes $\Theta(n^2)$ time, so this leaves us with the following divide-and-conquer recurrence in this worst-case scenario:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 2 \\ 2T(n/2) + \Theta(n^2), & \text{if } n > 2 \end{cases}$$

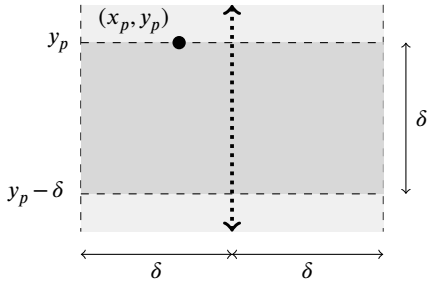
Applying the Master Theorem to this recurrence relation gives us $\Theta(n^2)$, so the worst-case time complexity of our "improved" divide-and-conquer approach is no better than the $\Theta(n^2)$ time complexity of brute force. If we want to achieve a better time complexity in the worst case, we will need to find a more efficient way to examine points near the median line.

Although it is not obvious, there exists a way to find the closest pair that crosses the median line in $\Theta(n)$ time, regardless of how the points are positioned. Let us consider an arbitrary point p in the δ-strip, located at coordinates (x_p, y_p). For the closest pair of points to cross the median line, the distance between the points must be at most δ, since δ is the best distance we know so far from computing the closest pair among points to the left and to the right of the median line.

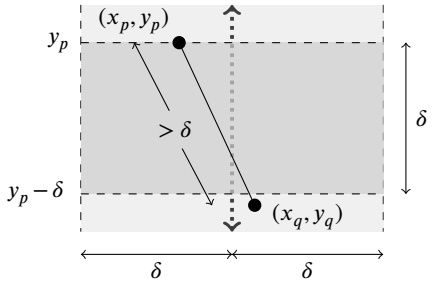
What is the maximum number of points located *below* point p (i.e., with a y-coordinate less than y_p) that can be within a distance of δ away from p? Notice that, in order for a point to be below p but still be within δ away from p, its y-coordinate must reside in the range $[y_p - \delta, y_p]$.



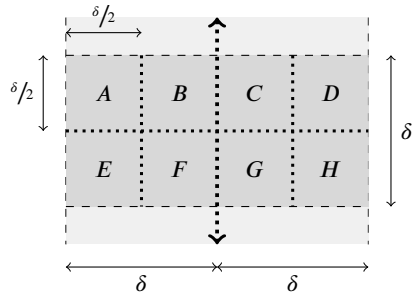
Using this information, we can thereby construct a $\delta \times 2\delta$ rectangle situated below point p, bounded by the left and right boundaries of the δ-strip. This rectangle is shown with a dark gray shading below:



For another point below p to be within δ away from the median line *and* δ away from point p itself, it *must* be situated inside this $\delta \times 2\delta$ rectangle. If a point q exists below this rectangle, then its distance from p must be farther than δ — thus, there would be no reason to compute the distance between p and q, since the pair (p, q) *cannot* be the solution to the problem.



Additionally, we will make the claim that *at most eight points can exist inside the $\delta \times 2\delta$ rectangle*, including point p. To see why this is the case, we will partition this rectangle into eight smaller squares with dimensions $(\delta/2) \times (\delta/2)$.



Only *at most one* point can feasibly exist in each of these eight squares. Why? Recall that we defined δ as the minimum of d_L and d_R, which respectively represent the closest pairs to the left and right of the median line. This means that no pair entirely contained on one side of the median line can have a distance less than δ — otherwise, δ would not be the distance of the closest known pair, contradicting our initial definition. However, there is no way to fit two points in a single $(\delta/2) \times (\delta/2)$ square without their distance being less than δ. Thus, we can conclude that it is impossible to fit more than one point in each of these eight squares, and that our original $\delta \times 2\delta$ rectangle can only fit at most eight points.

In the worst case, a point exists directly on the median line, requiring us to compute its distance to potentially seven other points in the rectangle.[3] Thus, if we want to find the closest pair of points that cross the median line, we only need to compute the distance from each point in our original δ-strip to at most seven other points; since seven is a constant, the computation for each point takes $\Theta(1)$ time. Because there can be at most n points in the δ-strip, the total time complexity of finding the closest pair that crosses the median line is now $n \times \Theta(1) = \Theta(n)$, provided that the points in the strip are sorted beforehand.

In summary, we can find the closest pair of points among a two-dimensional point set using the following algorithm:

1. Presort the set of points by their x-coordinate (so that finding the median line takes constant time) *and* their y-coordinate (so that we can easily iterate over the δ-strip to find the closest pair that crosses the median line for each recursive call). Given n points, this step takes $\Theta(n \log(n))$ time.
2. Compute the median line that splits the points in half by x-coordinate (this takes constant time if the points are presorted by x-coordinate beforehand).
3. Recursively find d_L and d_R, the closest pairs of points to the left and right of the median line. Define $\delta = \min(d_L, d_R)$.
4. Iterate over all points in the δ-strip from top to bottom (this can be done efficiently if the points are presorted by y-coordinate beforehand). For each point p in the δ-strip, compute its distance to the points within the $\delta \times 2\delta$ rectangle situated directly below p. The best distance encountered is stored as d_M, which is the closest pair of points that either crosses the median line or involves a point directly on the median line. This step takes $\Theta(n)$ time.
5. The closest pair of points is the minimum of d_L, d_R, and d_M.

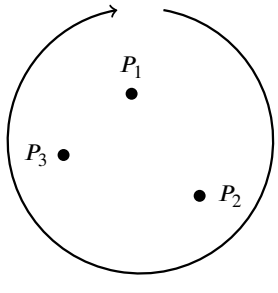
Presorting the points in step 1 takes $\Theta(n \log(n))$ time. Steps 2-5 represent a divide-and-conquer algorithm that can be expressed using the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 2 \\ 2T(n/2) + \Theta(n), & \text{if } n > 2 \end{cases}$$

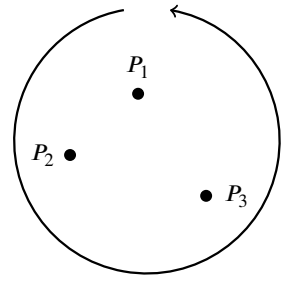
Applying the Master Theorem to this recurrence relation gives us $\Theta(n \log(n))$. We have therefore devised an algorithm that solves the two-dimensional closest pair of points problem in $\Theta(n \log(n))$ time, which is an improvement over the $\Theta(n^2)$ time of the brute force approach. (Although we will not be discussing it in detail here, the divide-and-conquer pattern we used to solve the problem can be generalized to higher dimensions as well, all while retaining its $\Theta(n \log(n))$ time complexity.)

26.4 The Clockwise Test and Triangle Area

Suppose you are given three non-collinear points, P_1, P_2, and P_3 on a two-dimensional plane. How can you determine if these three points are oriented in a *clockwise* or *counterclockwise* direction?



clockwise



counterclockwise

[3] Seven is actually not exact in this case. However, for our analysis, this does not matter, since the exact upper bound is still a constant.

To solve this problem, we will have to rely on a bit of linear algebra, specifically the concept of *determinants*. A determinant is a scalar value that is a function of the contents of a square matrix. The determinant $|M|$ of a 2×2 matrix M is equal to:

$$|M| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3×3 matrix M, the determinant $|M|$ is equal to:

$$|M| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

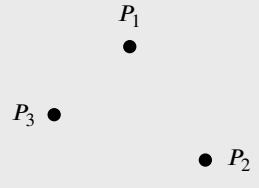
So, why are we talking about determinants? It turns out there is a formula involving determinants that can be used to compute the area of a triangle given three points:

$$\text{Area of Triangle} = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = \frac{1}{2}\left(x_1y_2 - y_1x_2 + y_1x_3 - x_1y_3 + x_2y_3 - x_3y_2\right) = \frac{1}{2}\left((x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)\right)$$

However, this is not all: the sign of the result also determines that orientation of the points in the triangle! If the result of the computation is *negative*, then the points $P_1 \to P_2 \to P_3$ are in a clockwise orientation; if the result is *positive*, then the points $P_1 \to P_2 \to P_3$ are in a counterclockwise orientation. If the result is *zero*, then the three points are collinear. An implementation of this equation is shown in the function below.[4]

```
1  // returns a pair containing the area of triangle formed by the three points,
2  // as well as a bool that indicates whether the points are in a clockwise
3  // or counterclockwise orientation (true = counterclockwise, false = clockwise)
4  std::pair<double, bool> area_of_triangle(Point a, Point b, Point c) {
5    double area = 0.5 * (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
6    if (area > 0) {
7      return {area, true};
8    } // if
9    else if (area < 0) {
10     return {std::fabs(area), false};
11   } // else if
12   else {
13     throw std::invalid_argument("The given points are collinear.");
14   } // else
15 } // area_of_triangle()
```

Example 26.7 You are given three points: $P_1 = (0, 1.5)$, $P_2 = (1, 0)$, and $P_3 = (-1, 0.6)$. Are these three points $P_1 \to P_2 \to P_3$ oriented in a clockwise or counterclockwise order? Also, what is the area of the triangle formed by these three points?



To solve this problem, we can use the formula for the area of a triangle defined above:

$$\begin{aligned}
\text{Area} &= \frac{1}{2}\left((x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)\right) \\
&= \frac{1}{2}\left((1-0)(0.6-1.5) - (-1-0)(0-1.5)\right) \\
&= \frac{1}{2}\left((1-0)(0.6-1.5) - (-1-0)(0-1.5)\right) \\
&= \frac{1}{2}\left((1 \times -0.9) - (-1 \times -1.5)\right) \\
&= \frac{1}{2}(-0.9 - 1.5) \\
&= \frac{1}{2}(-2.4) \\
&= -1.2
\end{aligned}$$

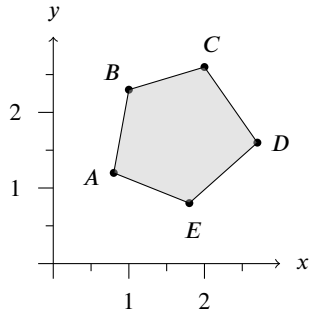
The area of the triangle formed by the given three points is therefore 1.2. Since the result of the calculation was negative, the points must have a clockwise orientation.

[4] If you need to implement this formula in a program, solving for $(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$ is preferred over solving for $x_1y_2 - y_1x_2 + y_1x_3 - x_1y_3 + x_2y_3 - x_3y_2$ since it involves fewer multiplications and additions.

26.5 Area of a Polygon

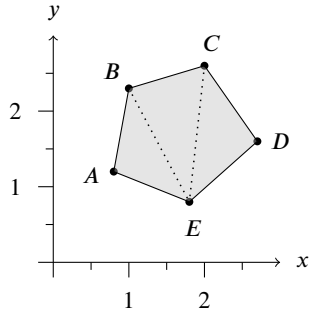
※ 26.5.1 *The Shoelace Formula*

The formula detailed in the previous section gives us the area of a triangle formed by three non-collinear points. However, what if we wanted to find the area of any simple polygon?[5] If the shape we are given is simple enough, such as a square, rectangle, trapezoid, etc., then we have formulas that can be used to solve for the area. In certain cases though, we could be given a polygon that is irregular and does not have a specific formula designed to find its area. For example, consider the following polygon:



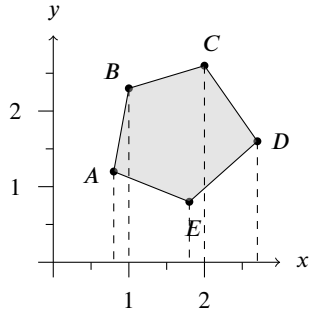
Point	Coordinate
A	$(0.8, 1.2)$
B	$(1, 2.3)$
C	$(2, 2.6)$
D	$(2.7, 1.6)$
E	$(1.8, 0.8)$

How can we find the area of this polygon? One intuitive strategy is to break up the polygon into separate triangles, apply the formula in the previous section to find each of their areas, and then combine the areas to get the area of the overall polygon.



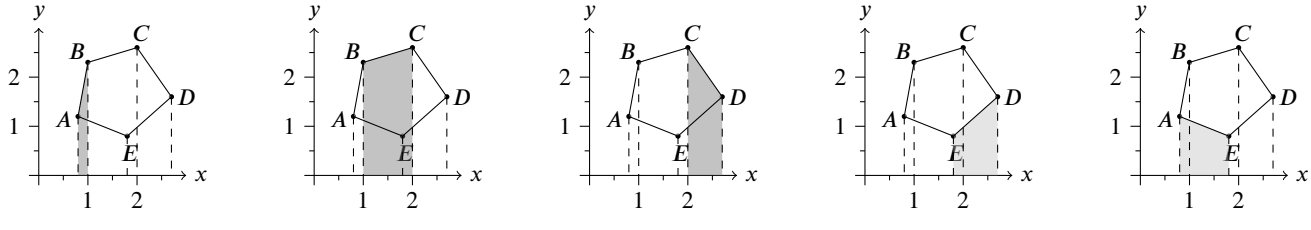
Point	Coordinate
A	$(0.8, 1.2)$
B	$(1, 2.3)$
C	$(2, 2.6)$
D	$(2.7, 1.6)$
E	$(1.8, 0.8)$

However, finding a way to break a polygon into triangles is not always an easy task, especially as the given polygon becomes more complex. Instead, an easier method for finding polygon area relies on *trapezoids* instead of triangles. To see why, take a look at what happens if we take each point of the polygon and draw a vertical line down to the x-axis.



Point	Coordinate
A	$(0.8, 1.2)$
B	$(1, 2.3)$
C	$(2, 2.6)$
D	$(2.7, 1.6)$
E	$(1.8, 0.8)$

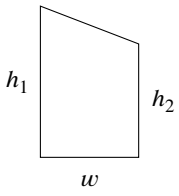
Notice that this deconstructs our polygon into a set of trapezoids whose slanted edges are formed from the polygon's edges.



The area of our polygon would be the total area of the darker trapezoids, minus the total area of the lighter trapezoids.

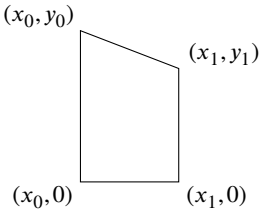
[5] A *simple polygon* is a polygon that does not intersect itself or have any holes. In this section, we will only be dealing with simple polygons, and we will assume that any points we are given trace the edges of the polygon in a valid order.

Much like triangles, trapezoids are easy to work with, as we can use a simple formula to find the area of any trapezoid:



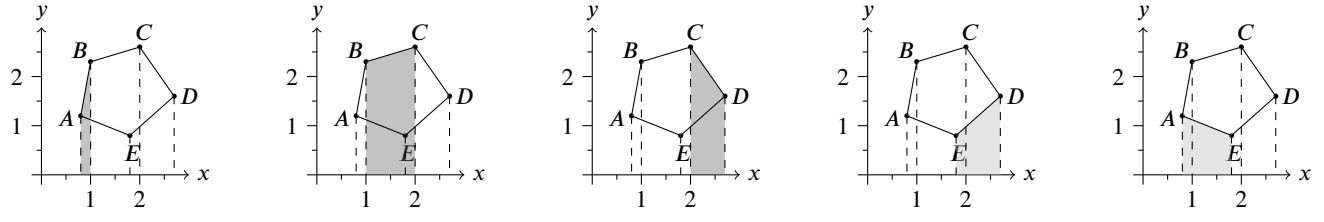
$$\text{Area} = \frac{w(h_1+h_2)}{2}$$

In our case, we are given the coordinate points of the trapezoid instead of the edge lengths. Here, we can use a redefined version of the formula that uses the four coordinates of the trapezoid to compute its area:

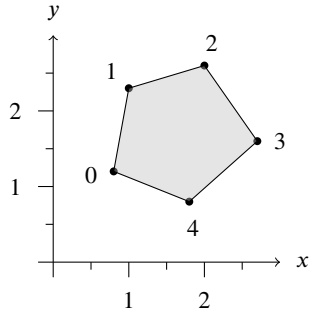


$$\text{Area} = \frac{(x_1-x_0)(y_0+y_1)}{2} = \frac{x_1y_0-x_0y_0+x_1y_1-x_0y_1}{2}$$

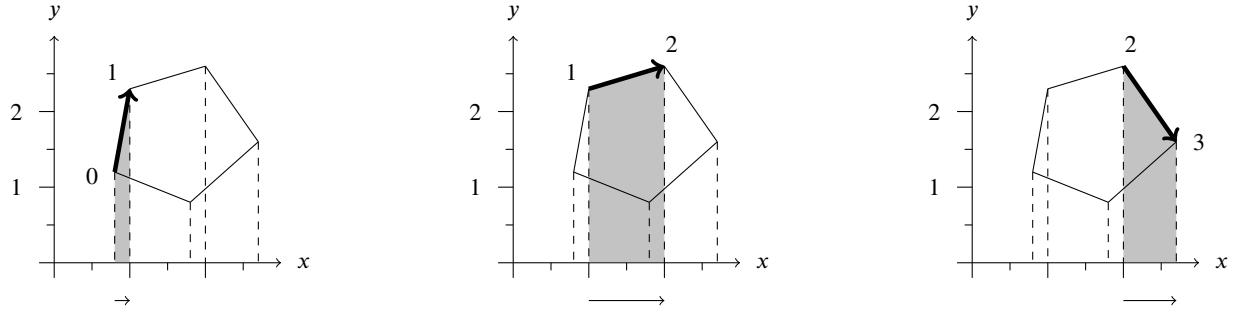
Even though we have a formula that can be used to find the area of a trapezoid, we still need to determine which trapezoids should be added, and which trapezoids should be subtracted. Recall from our example that the first three (darker) trapezoids should have their areas added, while the last two (lighter) trapezoids should have their areas subtracted.



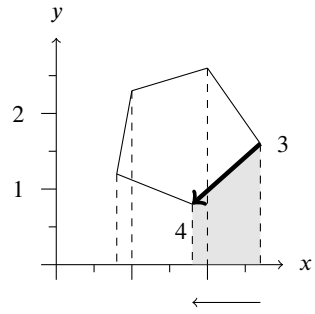
To accomplish this task, let us look at what happens if we number the points of the polygon in a clockwise order. Note that it does not matter which point gets assigned vertex 0, as long as the remaining points are numbered consecutively in a clockwise manner (you can use the clockwise test from the previous section to determine if you are moving in a clockwise direction or not). For our example, we will assign vertex A to 0, vertex B to 1, vertex C to 2, vertex D to 3, and vertex E to 4:

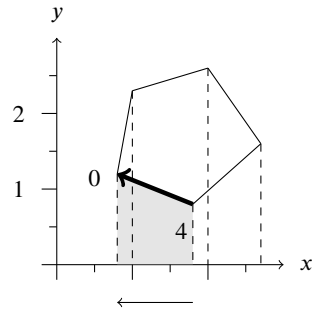


To determine if we need to add or subtract a trapezoid's area, we need to pay attention to how the x-coordinate changes as we move around the shape in a clockwise order, from vertex 0 to vertex 1, from vertex 1 to vertex 2, and so on. Notice that, if the value of the x-coordinate *increases* as we move in a clockwise manner, then the trapezoids formed along the corresponding edges should have their areas added:



Contrarily, if the value of the x-coordinate *decreases* as we move in a clockwise manner, the trapezoid areas should be subtracted:





These two situations do not need to be handled separately. Given n points that form a simple polygon, the sign of $(x_{(i+1) \bmod n} - x_i)$ will always indicate whether the corresponding trapezoid area should be added or subtracted, as long as you are visiting the vertices of the polygon in clockwise order. Therefore, we can define the *signed* trapezoid area using the following formula:[6]

$$\text{Signed Area of Trapezoid (Clockwise)} = \frac{(x_{i+1}-x_i)(y_i+y_{i+1})}{2} = \frac{x_{i+1}y_i - x_i y_i + x_{i+1}y_{i+1} - x_i y_{i+1}}{2}$$

The area of the polygon would therefore be the sum of the signed areas of all the trapezoids:

$$\text{Area of Polygon (Clockwise)} = \sum_{i=0}^{n-1} \frac{(x_{i+1}-x_i)(y_i+y_{i+1})}{2} = \sum_{i=0}^{n-1} \frac{x_{i+1}y_i - x_i y_i + x_{i+1}y_{i+1} - x_i y_{i+1}}{2}$$

Adding these trapezoids together gives us an interesting result. For instance, consider a polygon with three vertices, numbered 0, 1, and 2. Summing up the trapezoid signed areas gives us the following:

$$\text{Area of Polygon (Clockwise)} = \tfrac{1}{2}(x_1y_0 - x_0y_0 + x_1y_1 - x_0y_1 + x_2y_1 - x_1y_1 + x_2y_2 - x_1y_2 + x_0y_2 - x_2y_2 + x_0y_0 - x_2y_0)$$

Notice that several of these terms actually cancel each other out:

$$\text{Area of Polygon (Clockwise)} = \tfrac{1}{2}(x_1y_0 - \cancel{x_0y_0} + \cancel{x_1y_1} - x_0y_1 + x_2y_1 - \cancel{x_1y_1} + \cancel{x_2y_2} - x_1y_2 + x_0y_2 - \cancel{x_2y_2} + \cancel{x_0y_0} - x_2y_0)$$

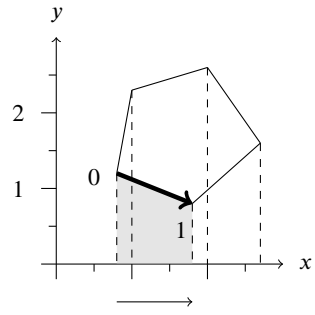
This leaves us with the following:

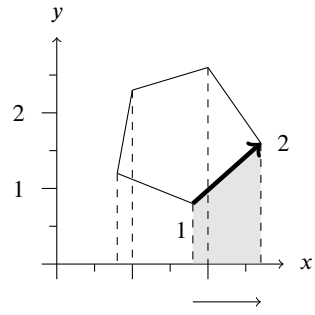
$$\text{Area of Polygon (Clockwise)} = \tfrac{1}{2}(x_1y_0 - x_0y_1 + x_2y_1 - x_1y_2 + x_0y_2 - x_2y_0)$$

In fact, we can generalize this outcome to any number of points. Given a set of n points that form a simple polygon in clockwise order, the total area of the polygon can be expressed using the following abbreviated formula:

$$\text{Area of Polygon (Clockwise)} = \tfrac{1}{2}\left(\sum_{i=0}^{n-1} x_{i+1}y_i - \sum_{i=0}^{n-1} x_i y_{i+1}\right)$$

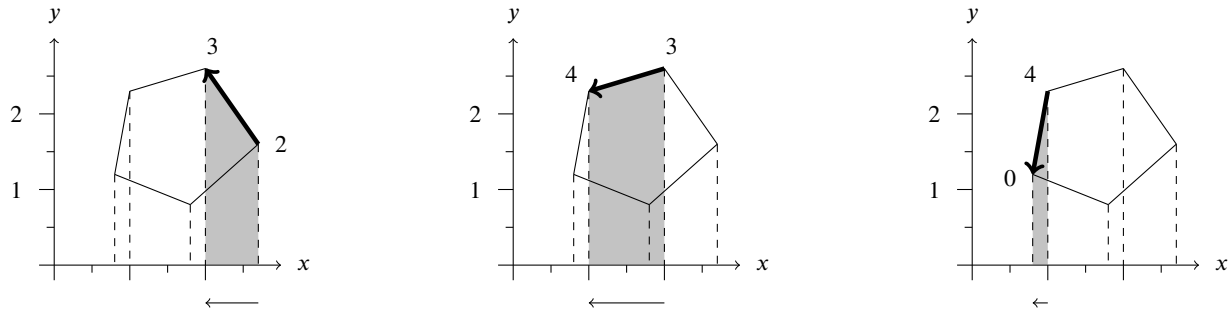
If the points are given in a counterclockwise order instead, we can use a similar approach to get a nearly identical formula (just with the signs flipped). If the x-coordinate value *increases* as we move in a *counterclockwise* manner, then the trapezoids formed along the edges should have their areas subtracted (the opposite of what happened when the points were in clockwise order):





[6]The "mod n" at the end of x_{i+1} is important, since we still want to consider vertex 0 as the vertex that comes after vertex $n-1$. In the example, if $i = 4$, then we want to consider $i+1$ as equal to 0 (or 5 mod 5), since that is the vertex that follows 4. This additional "mod n" has been removed from the formulas in this section just to make the formatting cleaner, but it is still implicitly there even if it is not explicitly written out.

Similarly, if the x-coordinate value *decreases* as we move in a *counterclockwise* manner, the trapezoid areas should be added:



In the counterclockwise case, the sign of $(x_i - x_{(i+1) \bmod n})$ determines if a trapezoid area should be added or subtracted. This allows us to define the signed trapezoid area using the following formula. Notice that this is just the negated version of the clockwise signed area of a trapezoid!

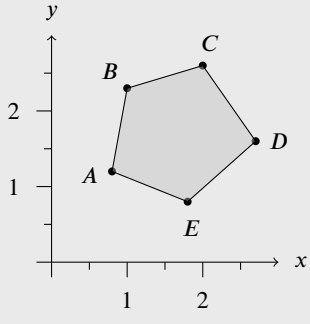
$$\text{Signed Area of Trapezoid (Counterclockwise)} = \frac{(x_i - x_{i+1})(y_{i+1} + y_i)}{2} = \frac{x_i y_{i+1} - x_{i+1} y_{i+1} + x_i y_i - x_{i+1} y_i}{2}$$

If we sum up all the trapezoids using this formula and remove all terms that cancel each other out, we would get this formula for the area of a polygon whose points are listed in a counterclockwise order:

$$\text{Area of Polygon (Counterclockwise)} = \frac{1}{2}\left(\sum_{i=0}^{n-1} x_i y_{i+1} - \sum_{i=0}^{n-1} x_{i+1} y_i\right)$$

Formally, this equation is known as the **Shoelace formula**, and it can be used to find the area of any simple polygon whose vertices are listed (in clockwise or counterclockwise order) as coordinates on a two-dimensional Cartesian plane. To invoke the Shoelace formula on a polygon, we have to solve either $(x_{i+1} y_i - x_i y_{i+1})$ or $(x_i y_{i+1} - x_{i+1} y_i)$ for all n vertices of the polygon. Since this arithmetic can be done in constant time, and it needs to be performed for each vertex, the total time complexity of the Shoelace formula is linear on the number of points that a polygon has, or $\Theta(n)$. While the equation for the Shoelace formula may seem complicated, you luckily do not have to memorize it since there is an intuitive way to remember how it works. This process will be covered using the following example.

Example 26.8 Apply the Shoelace formula to find the area of the example polygon used in this section (reproduced below):



Point	Coordinate
A	$(0.8, 1.2)$
B	$(1, 2.3)$
C	$(2, 2.6)$
D	$(2.7, 1.6)$
E	$(1.8, 0.8)$

The points in the polygon are provided in a clockwise order, so we can use the following formula to find its area:

$$\text{Area of Polygon (Clockwise)} = \frac{1}{2}\left(\sum_{i=0}^{n-1} x_{i+1} y_i - \sum_{i=0}^{n-1} x_i y_{i+1}\right)$$

Plugging in the coordinates gives us the following:

$$\begin{aligned}
\text{Area} &= \frac{1}{2}((x_B y_A + x_C y_B + x_D y_C + x_E y_D + x_A y_E) - (x_A y_B + x_B y_C + x_C y_D + x_D y_E + x_E y_A)) \\
&= \frac{1}{2}((1 \times 1.2 + 2 \times 2.3 + 2.7 \times 2.6 + 1.8 \times 1.6 + 0.8 \times 0.8) - (0.8 \times 2.3 + 1 \times 2.6 + 2 \times 1.6 + 2.7 \times 0.8 + 1.8 \times 1.2)) \\
&= \frac{1}{2}((1.2 + 4.6 + 7.02 + 2.88 + 0.64) - (1.84 + 2.6 + 3.2 + 2.16 + 2.16)) \\
&= \frac{1}{2}(16.34 - 11.96) \\
&= \frac{1}{2}(4.38) \\
&= 2.19
\end{aligned}$$

The area of the polygon is therefore 2.19.

※ 26.5.2 *Visualizing the Shoelace Formula*

From the example, we can actually see how the Shoelace formula got its name. Instead of writing out a giant formula, let's list out each of the vertices in a single column, repeating the first point at the end.

$$\begin{array}{ll} A & (0.8 \quad 1.2) \\ B & (1 \quad 2.3) \\ C & (2 \quad 2.6) \\ D & (2.7 \quad 1.6) \\ E & (1.8 \quad 0.8) \\ A & (0.8 \quad 1.2) \end{array}$$

If we draw a diagonal from each x-coordinate to the y-coordinate of the vertex directly below it, multiply the numbers that are connected, and sum all of these products, we would get the value of $\sum_{i=0}^{n-1} x_i y_{i+1}$:

$$\begin{array}{ll} A & (0.8 \quad 1.2) \\ B & (1 \quad 2.3) \\ C & (2 \quad 2.6) \\ D & (2.7 \quad 1.6) \\ E & (1.8 \quad 0.8) \\ A & (0.8 \quad 1.2) \end{array}$$

$$\sum_{i=0}^{n-1} x_i y_{i+1} = 0.8 \times 2.3 + 1 \times 2.6 + 2 \times 1.6 + 2.7 \times 0.8 + 1.8 \times 1.2 = 11.96$$

If we draw a diagonal from each y-coordinate to the x-coordinate of the vertex directly below it, multiply the numbers that are connected, and sum all of these products, we would get the value of $\sum_{i=0}^{n-1} x_{i+1} y_i$.

$$\begin{array}{ll} A & (0.8 \quad 1.2) \\ B & (1 \quad 2.3) \\ C & (2 \quad 2.6) \\ D & (2.7 \quad 1.6) \\ E & (1.8 \quad 0.8) \\ A & (0.8 \quad 1.2) \end{array}$$

$$\sum_{i=0}^{n-1} x_{i+1} y_i = 1 \times 1.2 + 2 \times 2.3 + 2.7 \times 2.6 + 1.8 \times 1.6 + 0.8 \times 0.8 = 16.34$$

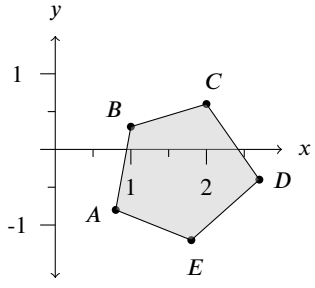
If we halve the difference between these two values and take the absolute value, we would get the area of the polygon. This is why this procedure is known as the Shoelace formula: the connections between the coordinates resemble shoelaces of a shoe. Visualizing the shoelace formula using this process also makes it much easier to remember!

$$\begin{array}{ll} A & (0.8 \quad 1.2) \\ B & (1 \quad 2.3) \\ C & (2 \quad 2.6) \\ D & (2.7 \quad 1.6) \\ E & (1.8 \quad 0.8) \\ A & (0.8 \quad 1.2) \end{array}$$

$$\text{Area} = \tfrac{1}{2}|11.96 - 16.34| = 2.19$$

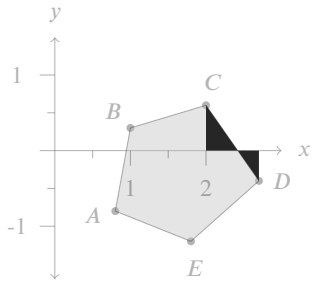
※ 26.5.3 *Shoelace Formula Edge Cases: Axis Intersection and Concavity*

The shoelace formula works for any simple polygon, but there are two edge cases that are worth discussing. First, what happens if a polygon intersects with the x-axis? Would the trapezoid strategy we used earlier still be valid?



Point	Coordinate
A	$(0.8, -0.8)$
B	$(1, 0.3)$
C	$(2, 0.6)$
D	$(2.7, -0.4)$
E	$(1.8, -1.2)$

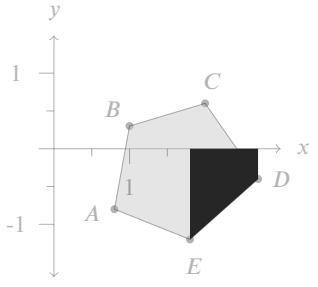
The answer is yes — even though we do not end up with a nicely formed trapezoid when considering two points on opposite ends of the x-axis, the total area obtained by the Shoelace formula is still correct because all excess areas eventually cancel out. For example, consider the edge from C to D. It we apply the signed area of a trapezoid formula to points C and D, we end up getting the signed area of the region below:



$$\text{Signed Area} = \frac{(x_D - x_C)(y_C + y_D)}{2} = \frac{(2.7-2)(0.6-0.4)}{2} = \frac{(0.7)(0.2)}{2} = 0.07$$

If we manually calculated the areas of these two triangles, we would see that the left triangle (within the polygon) has an area of 0.126 while the right triangle (outside the polygon) has an area of 0.056. So why does the signed area come out to 0.07? This is because the left and right triangles are on opposite sides of the x-axis, so the area of the right triangle "cancels out" a portion of the left triangle. However, we know that the area of 0.126 from the left triangle is all part of our total polygon area, so how can we make up this area of 0.056 that was cancelled out?

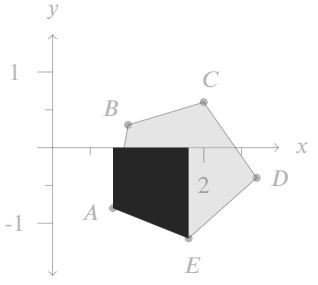
This missing area is actually regained when we look at points D and E. If we apply the signed area of a trapezoid formula to points D and E, we end up getting the signed area of the following trapezoid:



$$\text{Signed Area} = \frac{(x_E - x_D)(y_D + y_E)}{2} = \frac{(1.8-2.7)(-0.4-1.2)}{2} = \frac{(-0.9)(-1.6)}{2} = 0.72$$

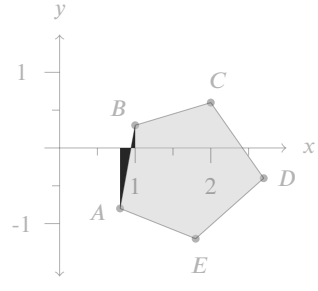
The signed area of the above trapezoid is 0.72, including the excess area near vertex D. As a result, this trapezoid *overcounts* the area of our polygon by the area of the excess region, which is 0.056. However, since the region we previously considered with points C and D *undercounted* the polygon area by 0.056, we end up back at the correct area in the end.

This phenomenon happens again when we find the area of the trapezoid formed by points E and A. Because edge $\overline{AB}$ crosses the x-axis, the trapezoid formed by E and A under the x-axis overcounts the area of the polygon.



$$\text{Signed Area} = \frac{(x_A - x_E)(y_E + y_A)}{2} = \frac{(0.8-1.8)(-1.2-0.8)}{2} = \frac{(-1)(-2)}{2} = 1$$

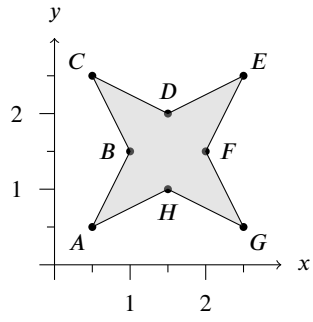
However, the signed area of the region formed from points A and B undercounts the polygon area by the same amount, as the excess area under the x-axis offsets the area above the x-axis, as shown:



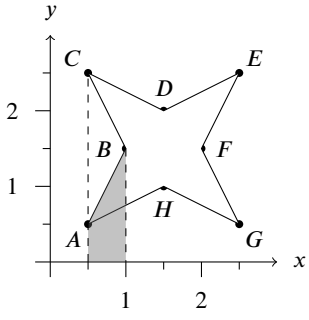
$$\text{Signed Area} = \frac{(x_B - x_A)(y_A + y_B)}{2} = \frac{(1-0.8)(-0.8+0.3)}{2} = \frac{(0.2)(-0.5)}{2} = -0.05$$

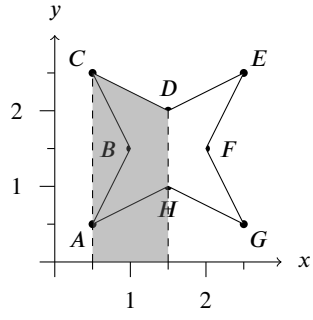
In general, the Shoelace formula still works even if the polygon intersects the x-axis. This is because any area that is overcounted by a trapezoid extending outside the polygon will always be offset by the signed area of another region that undercounts the polygon area by the same amount.

Another edge case that may initially appear problematic is the case of a concave polygon (i.e., a polygon whose edges cave inward), such as the one below. Concave polygons are interesting because multiple trapezoid areas may overlap.

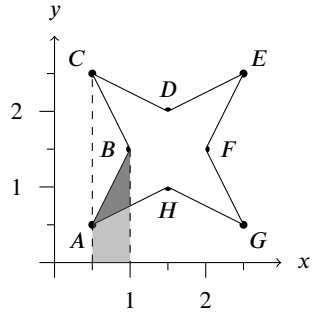


For instance, when considering A-B and C-D, we would add the areas of the following trapezoids:





However, there is a region that is counted by both of these trapezoids when we sum up their areas. Does this mean that this region is overcounted in our final polygon area?



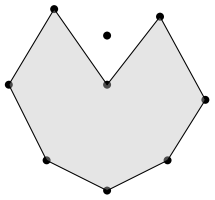
Not quite. Even though the region above is added twice when summing the trapezoid areas, it is also subtracted once to ensure that the total area remains correct. In our example, the darker shaded region of the above polygon is added twice (when considering A-B and C-D) and subtracted once (when considering B-C). The net effect is that this region is counted only once, which is the result we want. This outcome happens with any type of concave polygon — even if trapezoids overlap and a region's area is counted more than once, these excess areas will eventually be subtracted when considering other edges, and the polygon area will still be correct at the very end.

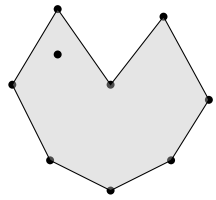
You can play around with many different types of simple polygons positioned anywhere on a two-dimensional coordinate plane, and the Shoelace formula will always find the correct area. This makes the Shoelace formula quite versatile in finding the area of many different shapes, as it not only hones a time complexity linear on the number of polygon points, but it also uses a constant amount of auxiliary space. The shoelace visualization covered earlier also makes this formula quite easy to remember!

Remark: The Shoelace formula can find the area of any simple polygon. However, if you want to find the areas of more advanced non-polygonal shapes, you may need to use a different algorithm. For shapes like circles and ellipses (ovals), there are existing formulas that can be used to compute their areas. Unfortunately, for more irregular shapes that do not support the Shoelace formula and also do not have a separate formula to calculate their areas, things can get a bit tricky. We will not be discussing this situation any further in this chapter, but there are several ways to compute the area of a shape that does not adhere to a formula or algorithm that is already known. One common technique for dealing with these irregular shapes is to first break them down into smaller shapes that you are able to find the areas of (such as triangles, circles, or even fractions of these shapes), and then combine these individual areas to estimate the area of the original shape.

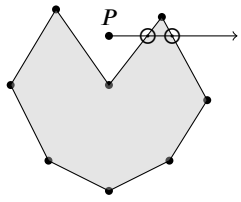
26.6 Point Inside Polygon

In this section, we will look at a problem that has useful applications in a variety of fields such as computer graphics, geographical systems, and game design: determining if a point lies inside a given polygon. Suppose you are given a polygon and coordinate point on a two-dimensional plane — how can you efficiently identify whether this point lines inside or outside the polygon?

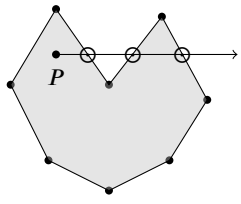




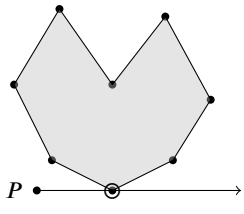
A solution to this problem is actually quite straightforward. To determine if a point P lies inside a polygon, consider a horizontal ray that emanates rightward from P (the direction of the ray does not matter, but we will consider a rightward ray to make things simple). We then count the number of times the ray intersects the polygon. If this number is even, then the point is outside the polygon. If this number is odd, then the point is inside the polygon. *(Note: this is actually an oversimplication that does not handle all edge cases, but we will address these later in this section, so keep reading.)* For example, the following ray intersects the polygon in two places. Two is even, so point P lies outside the polygon.



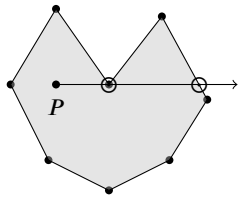
In the figure below, the ray intersects the polygon in three places. Three is odd, so point P lies inside the polygon.



However, there is a catch! For example, the following ray only intersects the polygon in one location. One is odd, so we would presume that its corresponding point is inside the polygon… but it isn't!

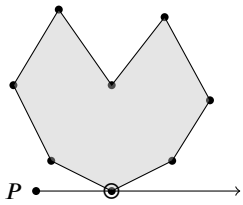


Similarly, the following ray intersects the polygon in two locations. Two is even, so we would presume that its corresponding point is outside the polygon… but once again, it isn't!

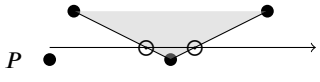


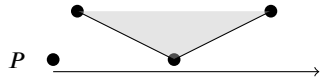
Notice that this issue only comes up if the ray directly intersects with a vertex of the polygon, or if the ray covers the entirety of an edge. This scenario a special edge case that we need to handle. If the ray crosses over a vertex of the polygon, we cannot count it as a single intersection if we want to arrive at the correct answer. Instead, we need to adjust the ray so that it avoids the vertex completely. This is done by wiggling the ray up and down by some small distance epsilon and then recounting the number of times the ray intersects the polygon. Eventually, you will end up with an even or odd number without this edge case, which you can then safely apply to determine if the point lies inside or outside the polygon using the odd-even rule discussed above.

For instance, consider the following point P whose ray intersects a vertex of the polygon.

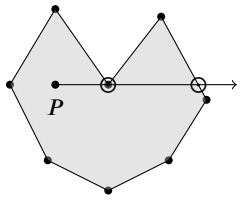


If we move the ray a tiny distance upward, we would see that it intersects the polygon twice without directly crossing over a polygon vertex. Likewise, if we move the ray a tiny distance downward, it would intersect the polygon zero times. Both of these values are even, so we can conclude that P resides outside the polygon.

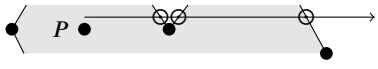


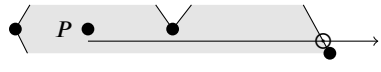


Similarly, let us consider the following point P, whose ray also intersects a vertex of the polygon.



If we move the ray a tiny distance upward, we would see that it intersects the polygon three times without directly crossing over a polygon vertex. Likewise, if we move the ray a tiny distance downward, it would intersect the polygon one time. Both of these values are odd, so we can conclude that P resides inside the polygon.



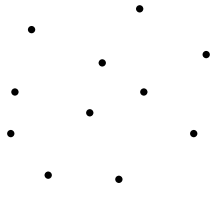


This algorithm is often known as the *ray-casting algorithm*. There are definitely other ways to determine if a point lies within a polygon, but this algorithm is definitely one of the simplest, especially in two dimensions. The worst-case time complexity of this algorithm is $\Theta(n)$, where n is the number of polygon points you are given. This is because the algorithm may need to perform an $\Theta(1)$ intersection analysis for each edge of the polygon, and the total number of edges in the polygon is $\Theta(n)$.

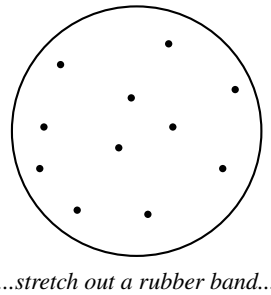
Remark: Another edge case that we have not discussed yet occurs when the given point lies on a vertex or edge of the polygon itself (which we will consider as inside the polygon). You can easily check if a given point lies on a polygon vertex by comparing the point with the coordinates of the vertices that make up the polygon. However, determining if a point lines on an edge is slightly more complicated. One approach is to do some preprocessing and compute the line segments that make up each of the polygon's edges, and then checking to see if the point lies on any of these segments. This does not change the overall time complexity of the algorithm, since this preprocessing step also takes $\Theta(n)$ time.

26.7 Convex Hull Algorithms (⁎)

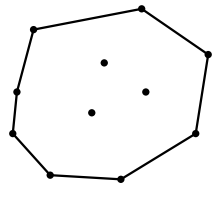
Given a set of points, a **convex hull** is the smallest convex polygon that fully encloses the given pointset. Intuitively, you can think of a convex hull as the polygon that is formed if you surround a set of points with a rubber band and then let go, allowing the rubber band to wrap tightly around the outermost points. An example is shown below:



Given a set of points...



...stretch out a rubber band...



...and let go to form a convex hull.

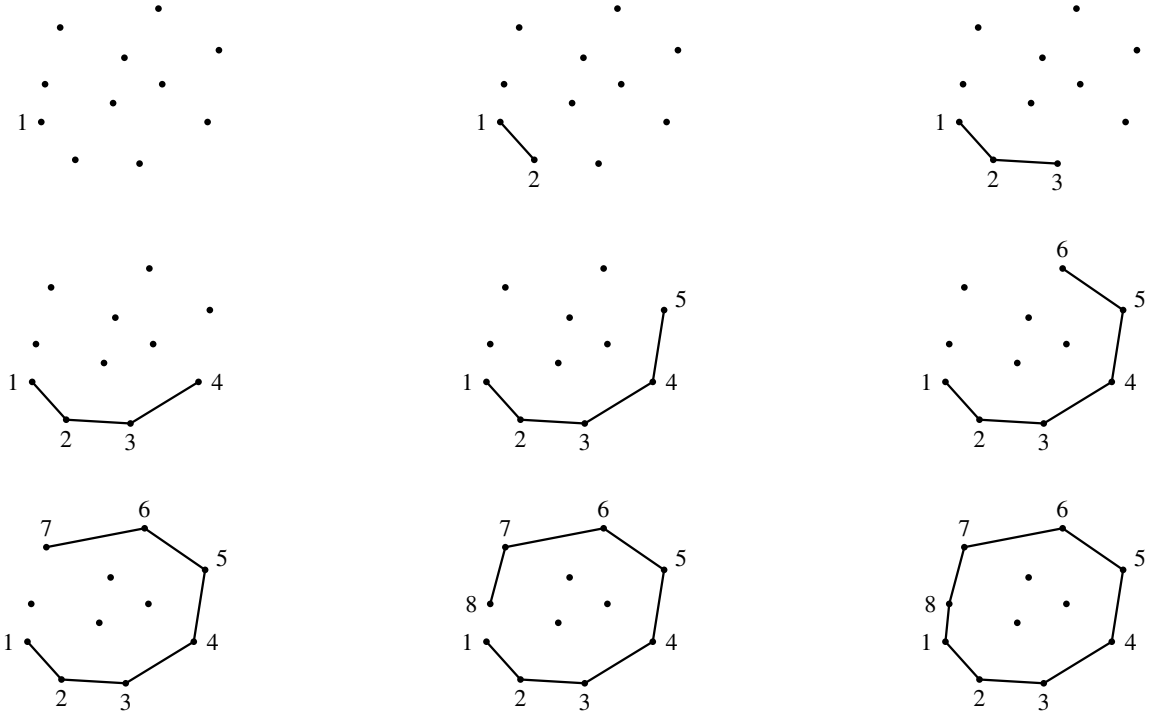
The convex hull is one of the most important structures in the field of computational geometry, as it is one of the simplest ways to approximate a shape for a given set of points. In this section, we will explore two common algorithms that can be used to find the convex hull of a 2-D pointset: *Jarvis's march* and *Graham's scan*.

※ 26.7.1 *Jarvis's March* (✻)

One elementary algorithm for computing a convex hull is **Jarvis's march**, also known as the *gift-wrapping algorithm*. Jarvis's march is implemented as follows:

1. Pick the leftmost point among the given set of points and set it as the current vertex (as this point is guaranteed to be on the convex hull).
2. Find the point that makes the smallest counterclockwise (leftward) turn relative to the current vertex, and connect the two vertices together. Then, set the new vertex as the current vertex.
3. Repeat until you return back to the starting vertex (this ends up "gift-wrapping" the points in a counterclockwise order).

Using the previous example, Jarvis's march would connect the vertices as follows:



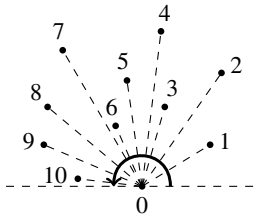
What is the time complexity of Jarvis's march? Let us define n as the total number of points we are given, and h as the total number of points that make up the convex hull. Notice that, for each of the h points on the convex hull, we have to compare it with $\Theta(n)$ other vertices to find the one that yields the smallest leftward turn. This requires us to find the relative angle between two points, which can be done in $\Theta(1)$ time. Therefore, the overall time complexity of Jarvis's march is $\Theta(nh)$.

※ 26.7.2 *Graham's Scan* (✻)

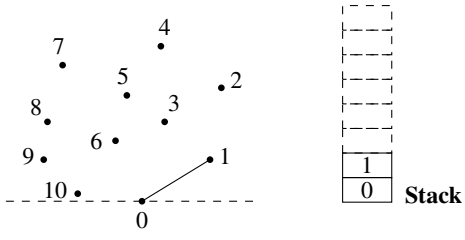
Another algorithm that can be used to compute a convex hull is **Graham's scan**. Graham's scan is implemented as follows:

1. Find the lowest point in the pointset (i.e., the one with the smallest y-coordinate) and use it as the starting point. If there are multiple lowest points, choose the one with the smallest x-coordinate.
2. Sort the remaining points by the angle each point forms with the lowest point, relative to the horizontal axis. If multiple points have the same angle, only the point farthest from the bottom-most point needs to be considered (as all inner points cannot be on the convex hull).
3. Initialize an empty stack. Push the bottom-most vertex and the point with the smallest angle into the stack.
4. Iterate over the remaining points in sorted order and perform the following:
   - If the point under consideration makes a *counterclockwise* (leftward) turn relative to the previous two points on the stack, push the point into the stack.
   - If the point under consideration makes a *clockwise* (rightward) turn or no turn relative to the previous two points on the stack, pop off the point at the top of the stack, and continue popping points from the stack until you end up with a counterclockwise turn.
5. Once you return to the starting point, the contents of the stack holds the vertices of the convex hull.

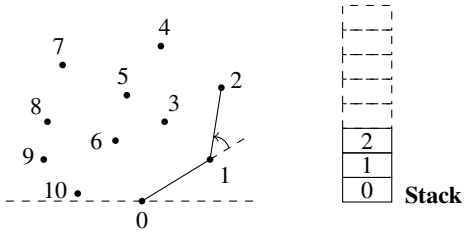
To see this algorithm in action, consider the same set of points as above. Point 0 is the lowest point, so we select it as our starting vertex. The remaining points are sorted in order of the angle it makes with point 0 relative to the x-axis (labeled from 1-10).



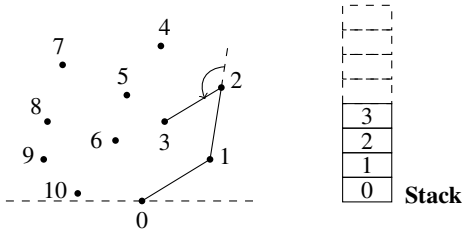
We initialize a stack and push in points 0 and 1. Then, we iterate over the remaining points in sorted order, pushing each point into the stack if it yields a counterclockwise/leftward turn, and popping from the stack if it yields a clockwise/rightward turn.



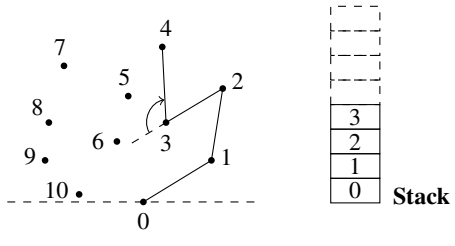
First, we consider point 2 and compare it with the line formed by the two points at the top of the stack (points 0 and 1). To reach point 2 from this line, we need to make a counterclockwise turn, so we push point 2 into the stack.

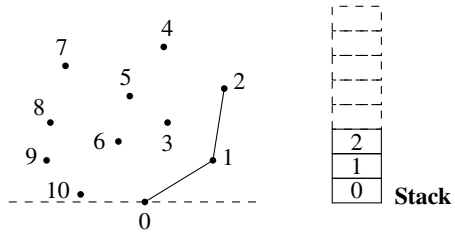


Next, we consider point 3 and compare it with the line formed by the two points at the top of the stack (points 1 and 2). To reach point 3 from the line, we need to make a counterclockwise turn, so we push point 3 into the stack.

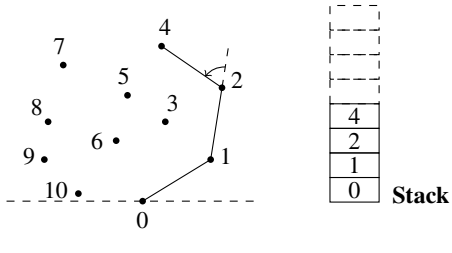


Next, we consider point 4 and compare it with the line formed by the two points at the top of the stack (points 2 and 3). To reach point 4 from the line, we need to make a clockwise turn, so point 3 is popped off the top of the stack.

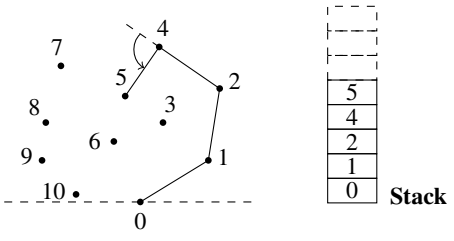




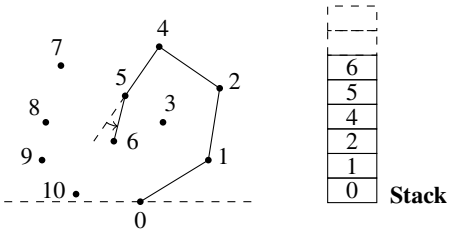
The two points at the top of the stack are now points 1 and 2. If we compare point 4 with the line formed by points 1 and 2, we can see that a counterclockwise turn is now needed to reach point 4. Thus, we can safely push point 4 onto the stack.



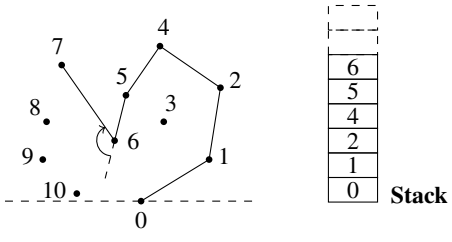
The next point we consider is point 5. To reach point 5 from the line formed by the two points at the top of the stack (points 2 and 4), we need to make a counterclockwise turn, so point 5 is pushed into the stack.

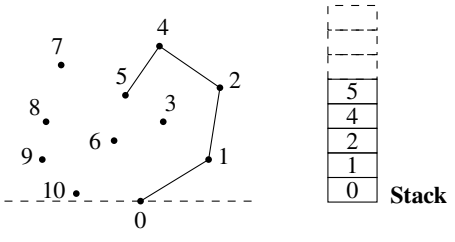


Next up, we consider point 6 and compare it with the line formed by points 4 and 5. To reach point 6 from this line, we need to make a counterclockwise turn, so point 6 is pushed into the stack.

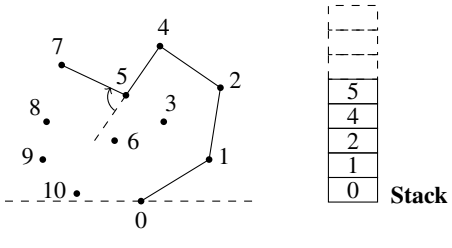


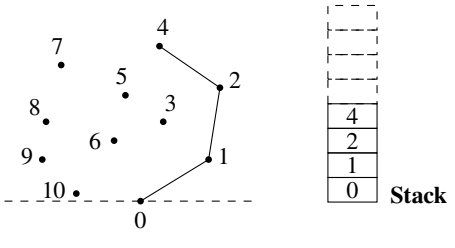
Next, we consider point 7 and compare it with the line formed by points 5 and 6. To reach point 7 from this line, we need to make a clockwise turn. Thus, the point at the top of the stack (point 6) is popped off.



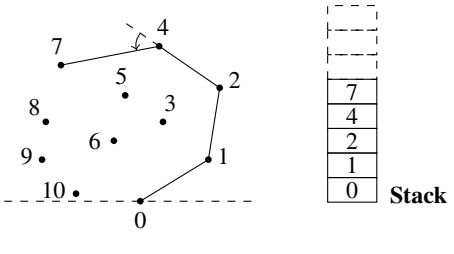


However, reaching point 7 still requires a clockwise turn from the line formed by points 4 and 5. As a result, we have to pop another value (point 5) off the top of the stack. Remember that we need to continuously pop values off the stack until the point under consideration can be reached using a counterclockwise turn!

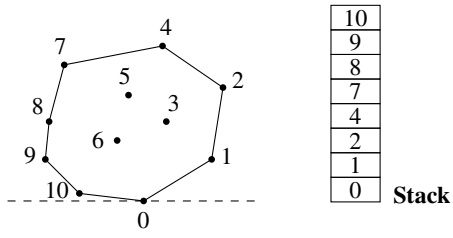




After popping off point 5, we can now reach point 7 using a counterclockwise turn from the line formed by points 2 and 4, which are the two points at the top of our stack. Thus, we can now safely push point 7 onto the stack.



If we continue this approach for the remaining points, we would end up with the following stack when we return to point 0. At this point, the algorithm ends, and the contents of the stack store the vertices of our convex hull.



What is the time complexity of Graham's scan? The implementation of Graham's scan can be split up into three subsequent components: finding the lowest point in the pointset, sorting the points by relative angle, and then processing all the points in sorted order. Given n points, finding the lowest point takes $\Theta(n)$ time (since you need to consider all n points to determine which one has the lowest y-coordinate), and sorting the points by angle takes $\Theta(n \log(n))$ time. However, the time required to process all the points after sorting is a bit more involved. It turns out that this final step actually takes $\Theta(n)$ time.

Why is this the case? Let us denote d_i as the total number of points that are popped from the stack while processing point p_i. In this case, the amount work spent processing p_i is proportional to $d_i + 1$, since we need to perform a $\Theta(1)$ clockwise test for each of the d_i points we pop, as well as the last point that needs to be tested before we can safely push point p_i onto the stack. If we sum up this value for all n, we would get a bound on the total work we need to do to process all n points:

$$\sum_{i=1}^{n} (d_i + 1) \quad = \quad n + \sum_{i=1}^{n} d_i$$

The key observation to notice here is that $\sum_{i=1}^{n} d_i$ cannot be larger than n. This is because each of the n points is pushed onto the stack only once, so a single point cannot be popped more than once. Thus, the total number of points we can pop cannot be greater than the number of points in the entire pointset itself!

$$n + \sum_{i=1}^{n} d_i \quad \leq \quad n + n$$

We can thereby conclude that the time complexity of processing all the points after sorting takes $\Theta(n)$ time. Out of the three steps, sorting has the dominant time complexity of $\Theta(n \log(n))$, so the overall time complexity of Graham's scan is also $\Theta(n \log(n))$.

※ 26.7.3 *Comparing Convex Hull Algorithms* (✻)

Which convex hull algorithm is better, Jarvis's march or Graham's scan? It depends. Given n points, the time complexity of Graham's scan is $\Theta(n \log(n))$. However, the time complexity of Jarvis's march is $\Theta(nh)$, where h is the number of vertices on the convex hull. Therefore, whether one algorithm is better than another depends on the relative values of $\log(n)$ and h. If h is asymptotically smaller than $\log(n)$, then Jarvis's march is better; otherwise, Graham's scan is better.

However, this opens up a brand new can of worms, since we do not know what h is until *after* we compute the convex hull! In fact, we deviated a bit from normal convention when describing the time complexity of Jarvis's march, since previous algorithms were analyzed in terms of input size alone. However, because the performance of Jarvis's march varies greatly depending on the output of the algorithm, its running time is also expressed in terms of its output size to accurately capture its performance. Such algorithms are known as *output sensitive algorithms*, and they are fairly common in the field of computational geometry.

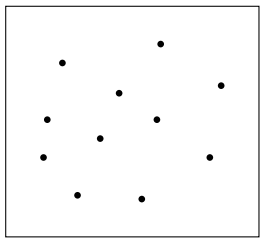
Since neither Jarvis's march nor Graham's scan is asymptotically optimal in all cases, you may be wondering if there exists an algorithm that is asymptotically optimal for all n and h. It turns out that there is: *Chan's algorithm* can be used to compute a convex hull in $\Theta(n \log(h))$ time by combining components from both Jarvis's march and Graham's scan. This algorithm is fairly involved, so we will not be discussing it in detail here. It should be noted that, while Chan's algorithm is asymptotically optimal, the improvement from $\Theta(n \log(n))$ to $\Theta(n \log(h))$ is often very small in practice. As a result, the simpler Graham's scan algorithm should typically suffice for finding a convex hull, even if it is not asymptotically optimal in all cases.

Time Complexity of Convex Hull Algorithms

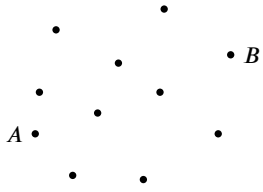
Jarvis's March	Graham's Scan	Chan's Algorithm
$\Theta(nh)$	$\Theta(n \log(n))$	$\Theta(n \log(h))$

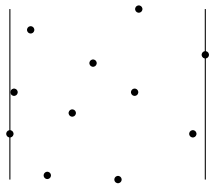
26.8 Bounding Box Algorithms (⁎)

Given a set of points, a **bounding box** is a rectangular border that fully encloses all of the points in the set. For instance, the following is a bounding box for the given set of points on a two-dimensional plane.

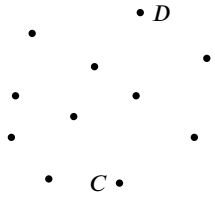


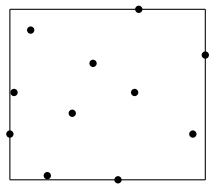
A **minimum bounding box** is the bounding box that encloses all the points with the smallest area (or volume in three dimensions, and so on for larger dimensions). If we confine our bounding box so that its edges are parallel to the x- and y-axes, then finding the minimum bounding box is fairly straightforward: for each axis, we simply have to find the minimum and maximum values among the given points and use these as the boundary of the box. For instance, the following points (labeled A and B) have the smallest and largest x-coordinates respectively, so these coordinates become the vertical boundaries of our bounding box.





Similarly, the following points (labeled C and D) have the smallest and largest y-coordinates respectively, so these coordinates become the horizontal boundaries of our bounding box.





Given n points, the time complexity of this algorithm is $\Theta(n)$, since we can build a bounding box by simply iterating over all n points and keeping track of the smallest and largest values of x and y.

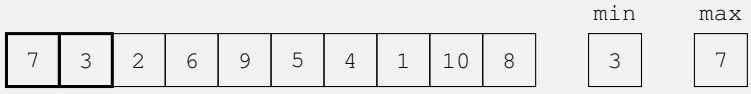
Remark: To solve the above problem, we will need to find the smallest and largest coordinate values of each dimension. This sounds fairly straightforward: to find the minimum of n values, simply keep track of a running minimum and continuously update it while iterating over the values (the same idea applies for finding the maximum).

```
1  int32_t get_min_value(const std::vector<int32_t>& vec) {
2    int32_t running_min = vec[0]; // assuming non-empty
3    for (size_t i = 1; i < vec.size(); ++i) {
4      running_min = std::min(running_min, vec[i]);
5    } // for i
6    return running_min;
7  } // get_min_value()
8
9  int32_t get_max_value(const std::vector<int32_t>& vec) {
10   int32_t running_max = vec[0]; // assuming non-empty
11   for (size_t i = 1; i < vec.size(); ++i) {
12     running_max = std::max(running_max, vec[i]);
13   } // for i
14   return running_max;
15 } // get_max_value()
```

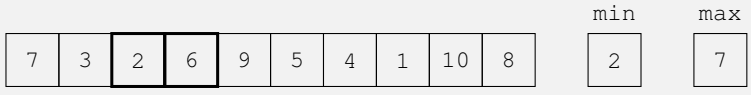
However, if you want to find *both* the minimum and maximum of a set of values, running this algorithm twice (once to find the minimum, once to find the maximum) is not actually the most efficient way to do things. Notice that each of the loops above require $n-1$ comparisons, so running both loops in succession would require a total of $2(n-1)$ comparisons. There is a way to do better *if we compute both the minimum and maximum at once*! The idea behind this optimized approach is to compare values in pairs rather than one by one — by doing so, we can use the relative ordering of the values in the pair to judiciously decide whether we want to compare each value with the running minimum or maximum. To illustrate how this works, consider the following array of values, for which we want to find both the minimum and maximum:

7	3	2	6	9	5	4	1	10	8

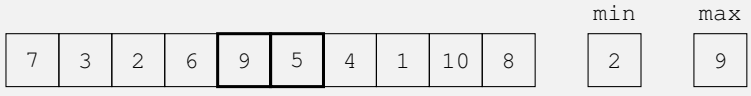
First, we will initialize a running minimum and running maximum, whose values are set to the minimum and maximum of the first two elements. In this case, the running minimum and maximum values are set to 3 and 7, respectively.



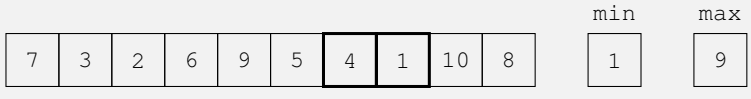
We then iterate over the remaining elements in pairs, comparing the minimum of the pair with the running minimum, and the maximum of the pair with the running maximum. In our example, the next pair we consider includes 2 and 6. 2 is the smaller of the pair, so it is compared with the running minimum; 6 is the larger of the pair, so it is compared with the running maximum (updating any values if necessary).



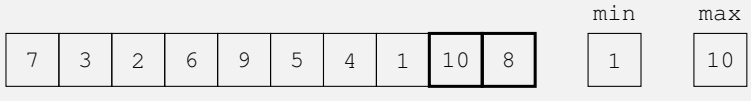
The next pair we consider includes 9 and 5. 5 is the smaller of the two, so it is compared with the running minimum, and 9 is compared with the running maximum (updating any values if necessary).



The next pair includes 4 and 1. 1 is smaller, so it is compared with the minimum, and 4 is compared with the maximum.



The next pair includes 10 and 8. 8 is smaller, so it is compared with the minimum, and 10 is compared with the maximum.



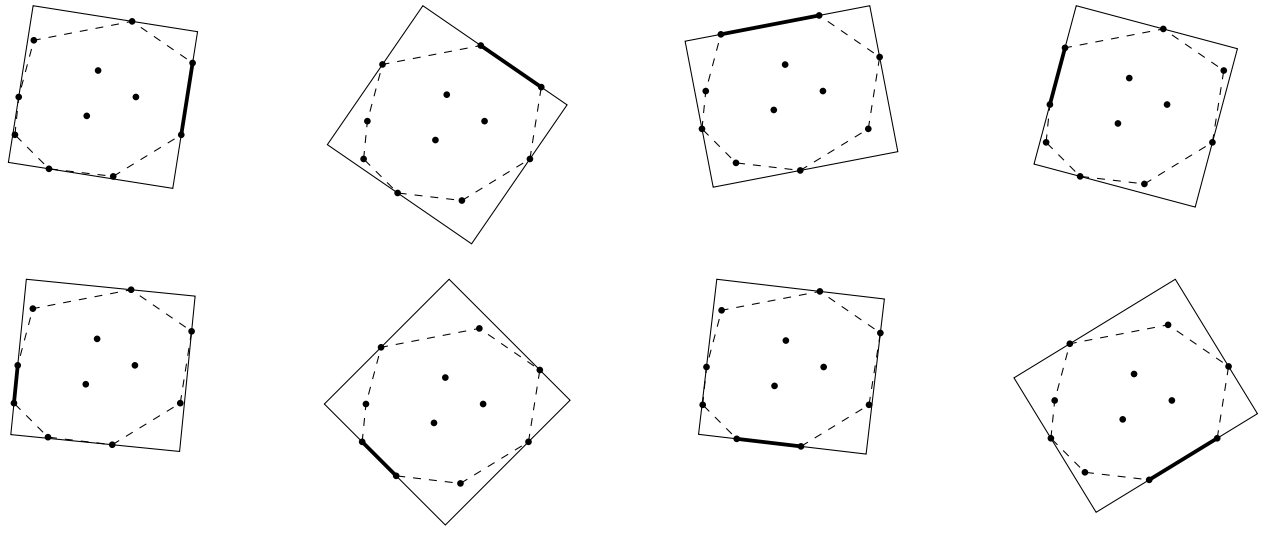
At the end of the algorithm, the running minimum and maximum would store the minimum and maximum of the entire set of values. By looking at values in pairs before comparing with the running minimum and maximum, we avoid having to make fruitless comparisons that will never lead us to a new minimum or maximum. This ends up dropping the total number of comparisons we need to make from $2(n-1)$ to $1+\lceil 3\times\frac{n-2}{2}\rceil$ (this is because we need one comparison between the first two values to initialize the running minimum and maximum, and then three comparisons for each of the remaining $\frac{n-2}{2}$ pairs: (1) comparing the values in the pair to determine which is smaller, (2) comparing the smaller value with the running minimum, and (3) comparing the larger value with the running maximum). Note: although this approach still takes $\Theta(n)$ time, the coefficient of the linear term ends up being smaller since we are completing $\sim 1.5n$ steps rather than $\sim 2n$ steps — as a result, we still have an improved algorithm even though the asymptotic complexity class does not change.

In practice, you will never have to implement a function like this on your own. Recall from chapter 11 that there is already a function in the STL that finds the minimum and maximum values of a range: `std::minmax_element()`.

If we remove the restriction that the bounding box needs to be parallel to the axes, then the problem becomes a bit trickier. The key insight to notice here is that the orientation of the bounding box must be aligned with one of the edges of the pointset's convex hull. This is because a convex hull is, by definition, the smallest convex shape that encloses all of the given points. Therefore, we can find the minimum bounding box by considering the bounding boxes aligned with each edge of the convex hull and taking the one with the minimum area. This procedure is summarized using the following steps:

1. Compute the convex hull of the given set of points.
2. For each edge of the convex hull, "rotate" the points on the plane so that this edge is parallel to the axes. Then, using the process discussed earlier, compute the minimum bounding box that is parallel with the x- and y-axes.
    - The word "rotate" is included in quotes here because we do not need to physically rotate the points in our algorithm. Instead, if we want to rotate our pointset to align with a certain edge of our convex hull, we can find the unit vector $\vec{u}$ corresponding to that edge of the hull and the unit vector $\vec{v}$ that is orthogonal (i.e., perpendicular) to $\vec{u}$. We can then take the dot products of $\vec{u}$ and $\vec{v}$ with the original vertices of the convex hull to get the new rotated coordinates. (Do not worry if you have no idea what this all means; the takeaway here is that we can use math to compute the rotated coordinates of our convex hull without having to physically rotate the points in our plane.)
3. Store the orientation with the minimum area encountered so far, updating it whenever a better solution is found. After all edges of the convex hull are considered, return the best solution that is stored, which corresponds to the minimum bounding box of the entire pointset.

The minimum bounding box for the example pointset, when allowing orientation, is the smallest of the eight boxes below:

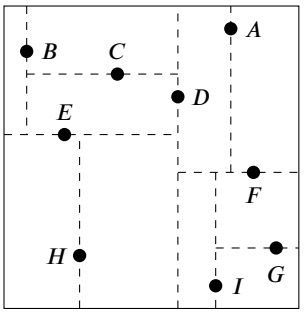


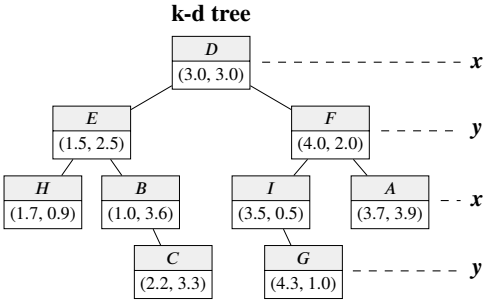
26.9 k-d Trees (⁎)

※ 26.9.1 *Partitioning Data in a k-d Tree* (⁎)

A **k-dimensional tree**, also known as a **k-d tree** for short, is a binary search tree that can be used to efficiently organize points in a k-dimensional space. In a k-d tree, each node represents a k-dimensional point, and each level of the tree recursively partitions a dimension of the point space into two halves. An example of a k-d tree of dimension $k = 2$ is shown below:

Point	Coordinate
A	(3.7, 3.9)
B	(1.0, 3.6)
C	(2.2, 3.3)
D	(3.0, 3.0)
E	(1.5, 2.5)
F	(4.0, 2.0)
G	(4.3, 1.0)
H	(1.7, 0.9)
I	(3.5, 0.5)

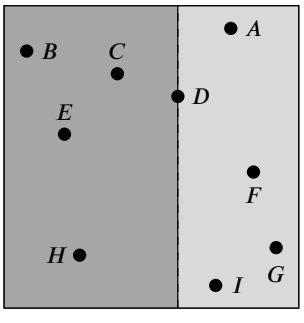


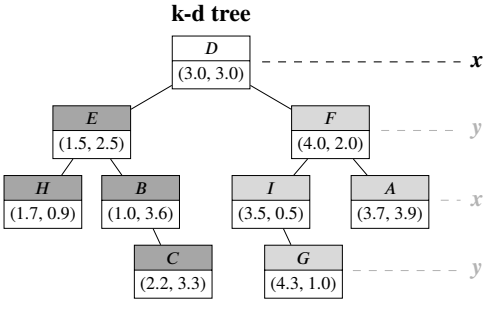


In a standard binary search tree, items that are smaller than a given node end up in its left subtree, and items that are larger end up in its right subtree. A k-d tree works in a very similar way, just generalized to multiple dimensions. You can think of each node of a k-d tree as a partition point that splits a dimension into two halves: points in the smaller half are sent to its left subtree, and points in the larger half are sent to its right subtree. The dimension to partition on alternates at each level of the tree, so a given node that partitions along the x-dimension would have children that partition along the y-dimension, and so on (circling back to the x-dimension after all dimensions are partitioned).[7] The dimension to partition on is known as the **discriminator**.

For example, consider the tree above. Point D serves as the root of the tree, which discriminates on the x-dimension. Visually, this partitions the pointset along $x = 3$ (the x-coordinate of D), as shown. Points that have an x-coordinate < 3 are in the left subtree of D, and points that have an x-coordinate ≥ 3 are in the right subtree (recall from chapter 18 that duplicates can be handled arbitrarily as long as you are consistent, but we will default to the right subtree in these notes).

Point	Coordinate
A	(3.7, 3.9)
B	(1.0, 3.6)
C	(2.2, 3.3)
D	(3.0, 3.0)
E	(1.5, 2.5)
F	(4.0, 2.0)
G	(4.3, 1.0)
H	(1.7, 0.9)
I	(3.5, 0.5)

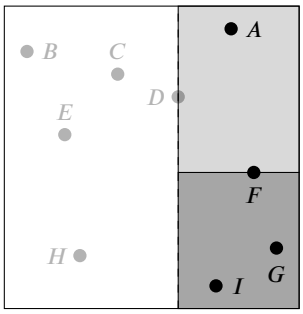


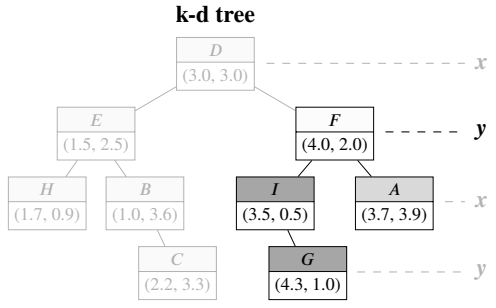


[7] Alternating the partition dimension (discriminator) in a round-robin fashion is not the only way we can build a k-d tree, and it may actually be sub-optimal depending on the data distribution. Another method is to select the discriminator as the dimension on which the points have the greatest *spread*, or difference between the largest and smallest coordinate values. However, to keep things simple, we will only look at the round-robin version in this chapter.

Furthermore, let's look at the right subtree of D, which represents the region to the right of point D. Point F serves as the root of this subtree, and it is on a level of the tree that discriminates on the y-dimension. Visually, this partitions the region to the right of D along $y = 2$ (the y-coordinate of F), as shown. Points to the right of D that have a y-coordinate < 2 are in the left subtree of F, and points to the right of D that have a y-coordinate ≥ 2 are in the right subtree of F. (The same analysis applies to the left subtree rooted at E, which partitions the region to the left of D along $y = 2.5$).

Point	Coordinate
A	(3.7, 3.9)
B	(1.0, 3.6)
C	(2.2, 3.3)
D	(3.0, 3.0)
E	(1.5, 2.5)
F	(4.0, 2.0)
G	(4.3, 1.0)
H	(1.7, 0.9)
I	(3.5, 0.5)

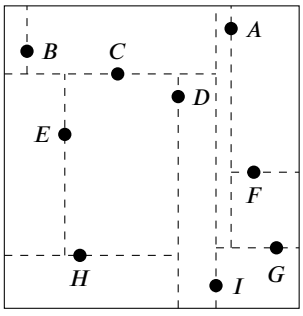


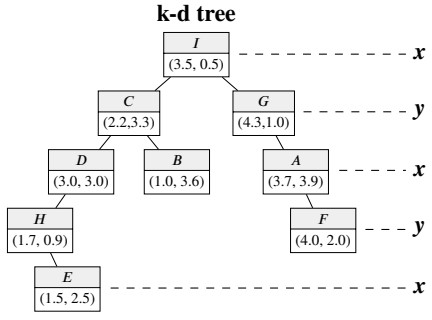


We can follow the same process to see how the remaining points in the tree are partitioned. For example, point I is the root of the left subtree of F, and it is on a level of the tree that discriminates on the x-dimension. Therefore, point I partitions the region of the point space to the right of D and below F (i.e., the darker shaded region above) along its x-coordinate value of 3.5. Any point in this darker shaded region to the left of $x = 3.5$ is sent to the left subtree of I, and any point in this region to the right of $x = 3.5$ is sent to the right subtree of I.

From this example, we can see that multiple distinct k-d trees can be built for the same set of points, based on the manner in which we complete our partitions. Below is an example of another k-d tree that partitions the points in a different order.

Point	Coordinate
A	(3.7, 3.9)
B	(1.0, 3.6)
C	(2.2, 3.3)
D	(3.0, 3.0)
E	(1.5, 2.5)
F	(4.0, 2.0)
G	(4.3, 1.0)
H	(1.7, 0.9)
I	(3.5, 0.5)

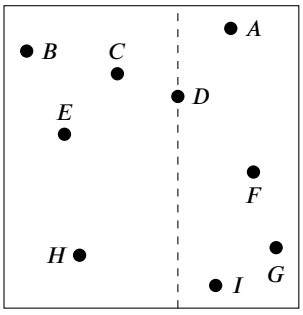




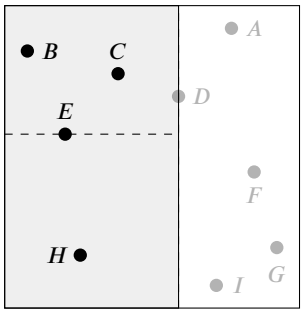
One thing you may have noticed here is that the way in which we partition the points may affect the efficiency of searching through the tree. Recall that the best tree structure we can have is a balanced tree, which ensures worst-case $\Theta(\log(n))$ search, insertion, and removal. However, the balancing techniques that we introduced back in chapter 18 do not work on k-d trees because each level of the tree strictly specifies the dimension that a given point discriminates on (and thus you cannot trivially change the level a node is situated on while rebalancing). Because of this, the time complexities of search, insertion, and removal from the standard k-d tree discussed above are each worst-case $\Theta(n)$.

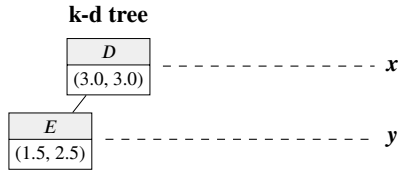
※ 26.9.2 *Inserting into a k-d Tree* (✻)

However, if you know all the points that you need to insert into a k-d tree beforehand, there is a way to build the tree so that it always ends up being balanced (and thus supports $\Theta(\log(n))$ tree operations). The idea is to always insert the *middle* point (or either of the two middle points) for the discriminator being considered, which allows the remaining points to be evenly split to the left and right of the newly inserted node. In our example, if we want to build a k-d tree with an initial discriminator on the x-dimension, we would insert point D first because D has the median x-coordinate of the available points.

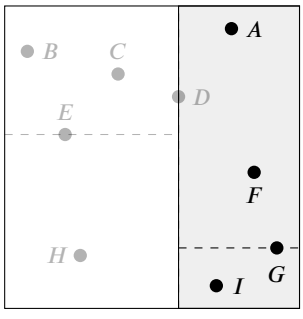


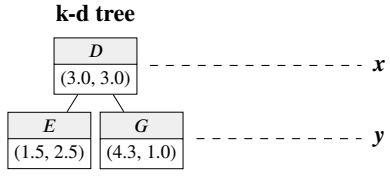
The next level of the k-d tree partitions the points based on y-coordinate. For the left subtree of D, we want to insert the point to the left of D with the middle y-coordinate. There are four points to the left of D: B, C, E, and H. The two middle points are C and E, so we can insert either as the root of the left subtree (in this case, we will choose the smaller one, which happens to be E).



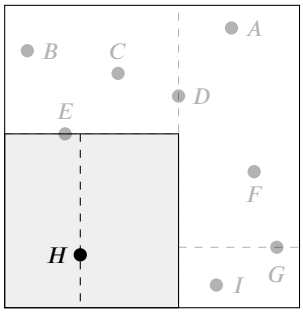


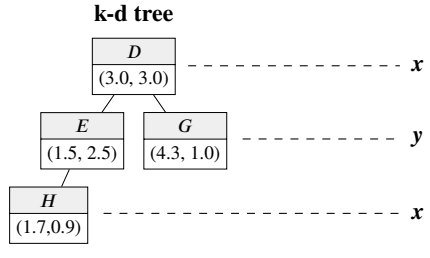
We can apply a similar idea for the points right of D. Of the four points to the right of D, the points with the middle y-coordinates are F and G, so we can insert either of these points as the root of the right subtree of D. We will select the point with the smaller coordinate value, or G.

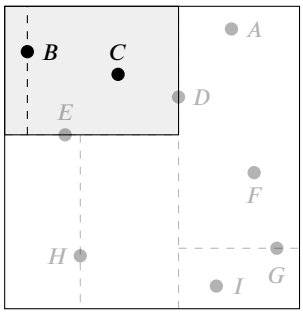


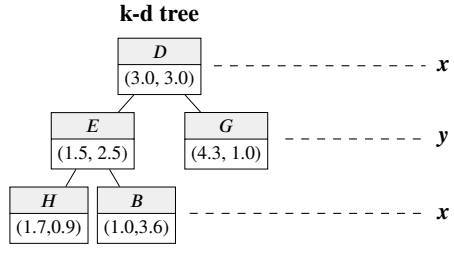


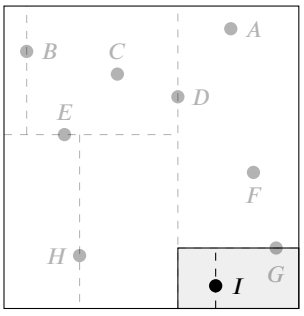
Continuing this process, we would get the following. Each point that we insert into the k-d tree subdivides its region such that the number of points on its left and right subtrees differs by no more than one, ensuring that our final k-d tree is balanced.

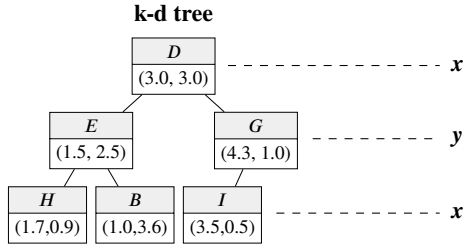


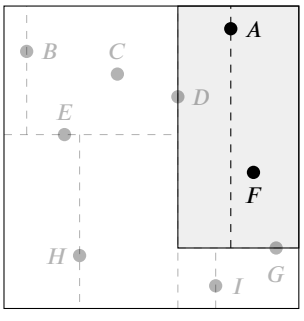


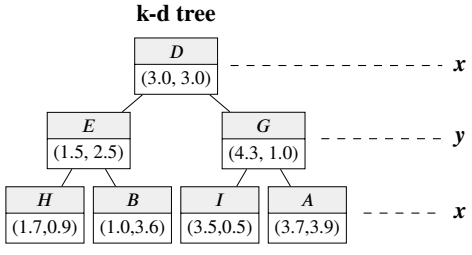


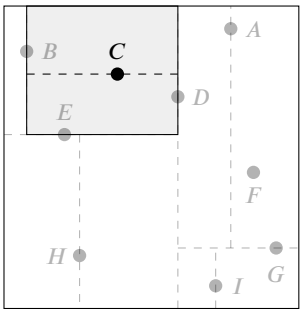


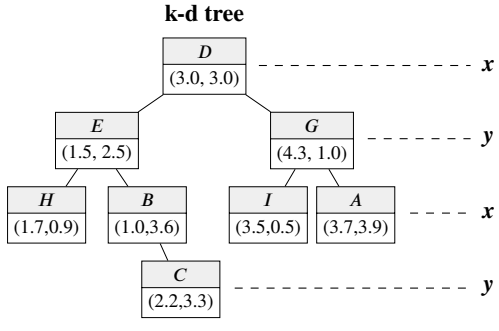


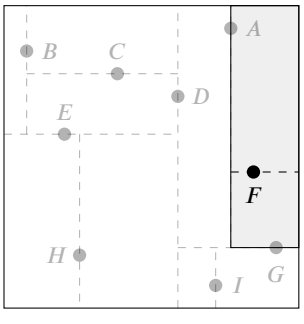


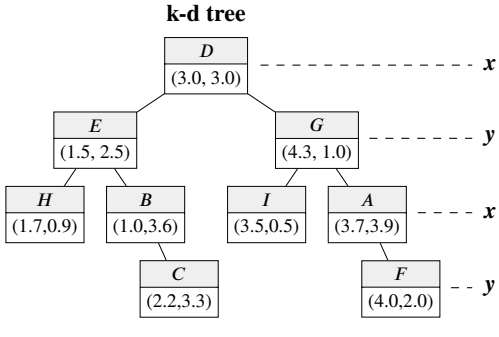






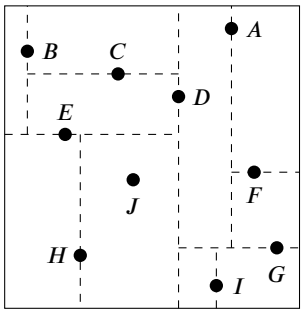


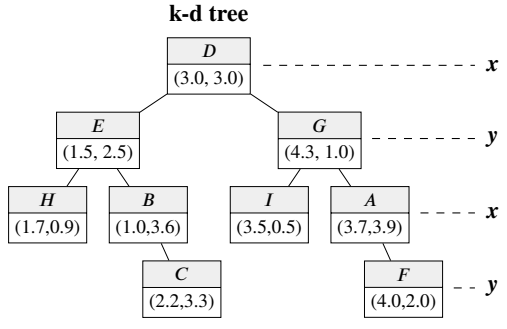




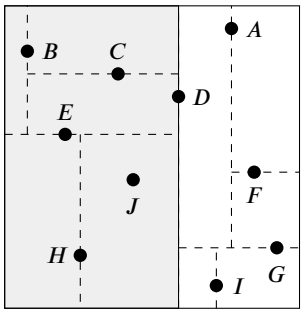
By inserting the points in this manner, we get a balanced k-d tree that supports $\Theta(\log(n))$ search.

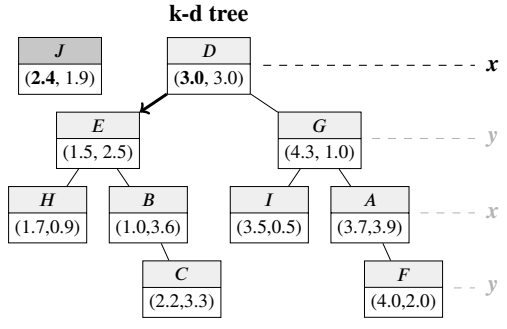
Much like inserting into a standard binary search tree, inserting a point into a k-d tree is relatively straightforward. If the coordinate value of the new point (as indicated by the discriminator) is smaller than the corresponding coordinate value of a node in the tree, make a recursive call on its left subtree; otherwise, make a recursive call on its right subtree. In either case, the discriminator is alternated among all the dimensions as you traverse down the tree. For example, suppose we wanted to insert point J below, located at the coordinate (2.4, 1.9).



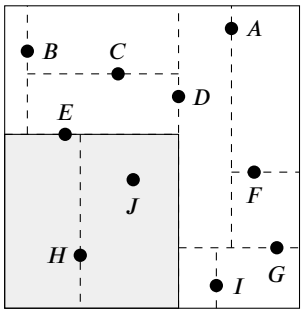


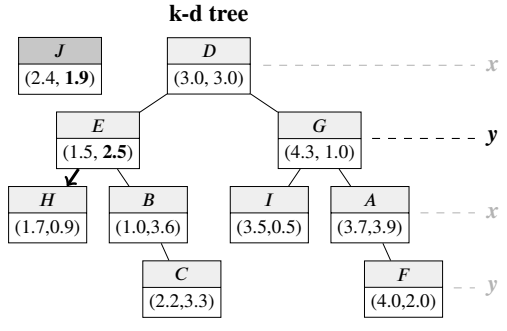
We first compare J with the root node of D. Since the root level of the k-d tree has a discriminator of x, we will compare the x-coordinate of J with the x-coordinate of D to determine which side of the tree J should go. In this case, $2.4 < 3.0$, so J belongs in the left subtree of D.



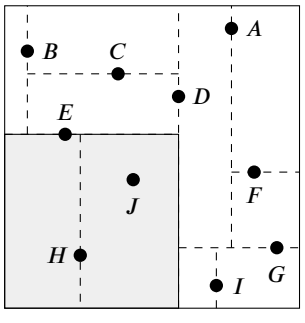


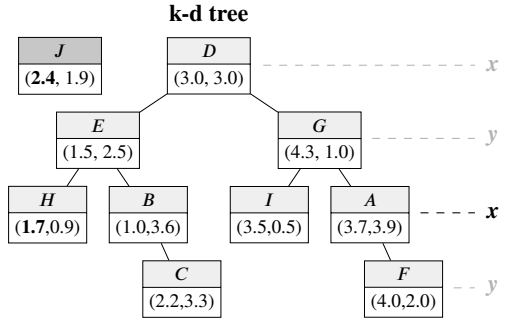
Next, we compare J with E. Since this level of the tree has a discriminator of y, we will compare the y-coordinate of J with the y-coordinate of E to determine which side of the tree J should go. In this case, $1.9 < 2.5$, so J belongs in the left subtree of E.



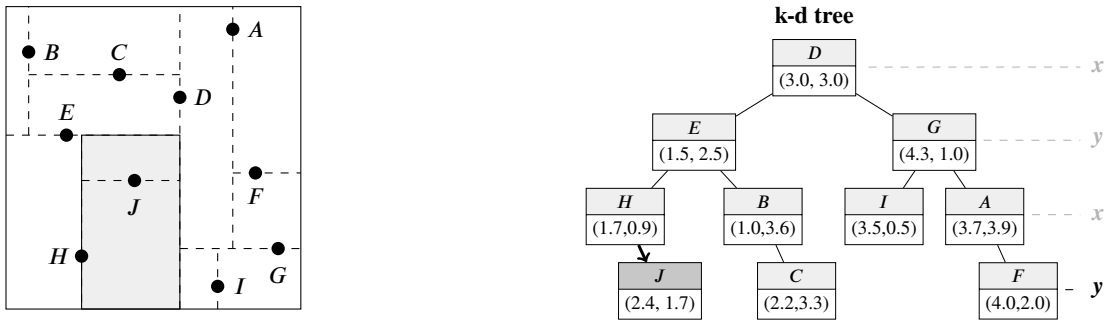


Next, we compare J with H. Since this level of the tree has a discriminator of x, we will compare the x-coordinate of J with the x-coordinate of H to determine which side of the tree H should go. In this case, $2.4 > 1.7$, so J belongs in the right subtree of H.





Since H does not have anything in its right subtree, we can insert J in this position.



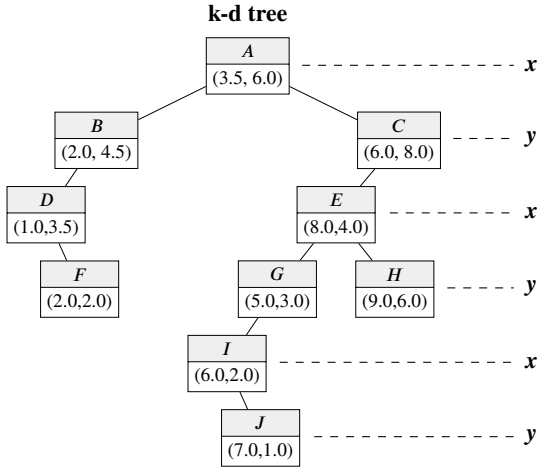
As you can see, the time complexity of insertion depends on the number of elements you have to traverse before you find an open position for the point you want to insert. In the worst-case, your k-d tree could be in the form of a stick, which could require you to traverse every node of the tree before you find an open position; hence, the worst-case time complexity of insertion is $\Theta(n)$ if given n points. On average, however, you can expect the tree to be fairly balanced, and the average-case time complexity of insertion turns out to be $\Theta(\log(n))$.

※ 26.9.3 *Deleting from a k-d Tree* (✻)

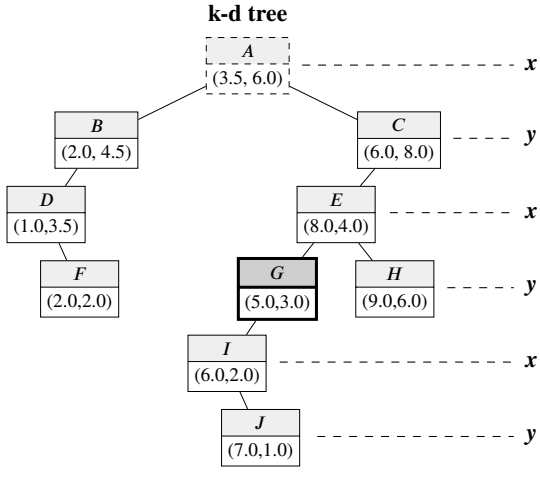
Deleting from a k-d tree is slightly trickier because each level of the tree may have a different discriminator. To delete a point from a k-d tree with a discriminator dimension d, you can follow these steps:

1. If the node to delete is a leaf node, delete the node and return.
2. Otherwise, if the node is not a leaf node:
   - If the node has a right subtree, find the node in the right subtree with the smallest d value and replace the node you want to delete with this d-minimum node. Then, recursively delete the original d-minimum node.
   - Otherwise, if the node has no right subtree, move the left subtree so that it becomes the new right subtree. Then, find the node with the smallest d value in this new right subtree and replace the node you want to delete with this d-minimum node. Then, recursively delete the original d-maximum node.

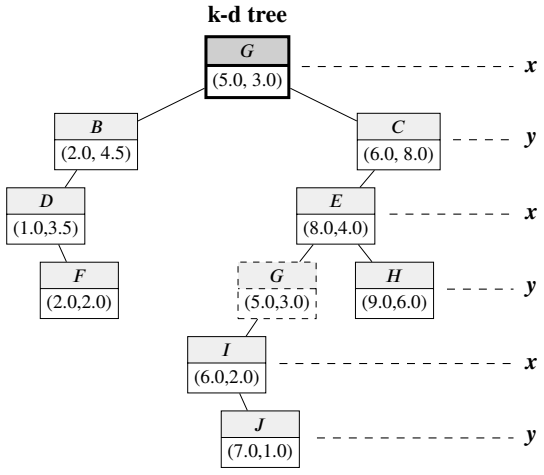
For instance, consider the following k-d tree, from which we want to delete point A:



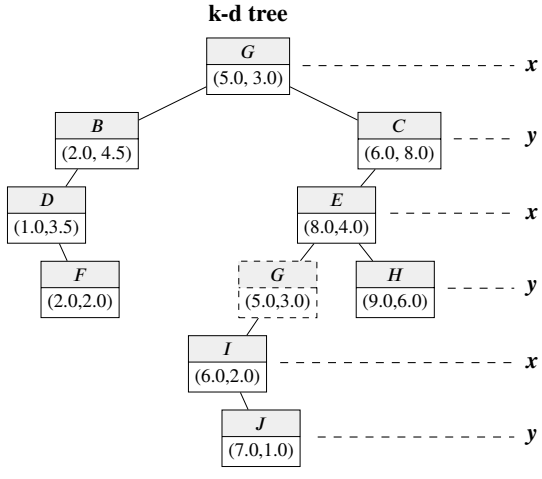
The discriminator of A is x. Since A is not a leaf node, we will find the node in the right subtree of A with the smallest x value, which happens to be point G:

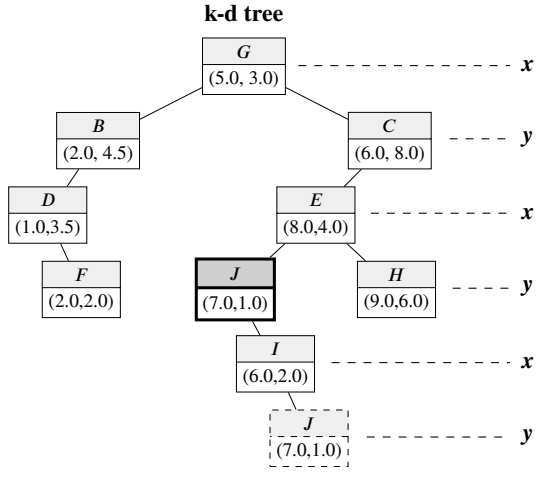


We then replace point A with point G, and then we recursively delete the original node for point G:

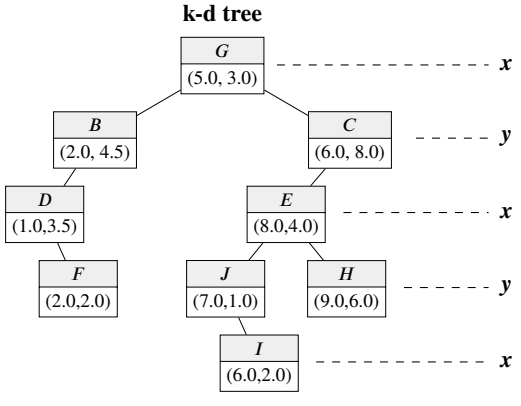


The discriminator of the original node G is y. Since the original G node is not a leaf node and does not have a right subtree, we will swap the left subtree of G so that it becomes the new right subtree. Then, we replace node G with the node in this new subtree with the smallest y value, which happens to be point J:



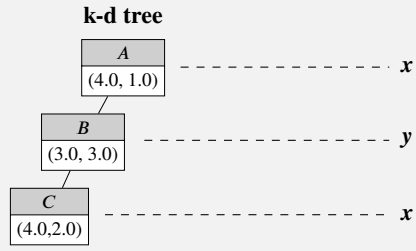


The node for point J is a leaf node, so we can safely delete it without processing any other nodes in the k-d tree. This gives us our final k-d tree after point A is deleted.

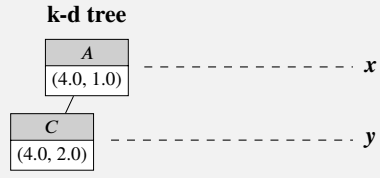


Remark: Why do we need to swap in the left subtree if a node to delete does not have a right subtree? Wouldn't it be easier to replace the node to delete with the descendant in the *left* subtree with the *largest* discriminator value?

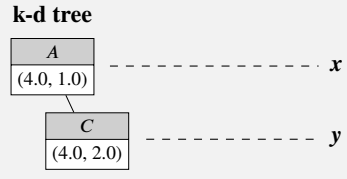
The reason we perform the swap is to handle the case of equal discriminator values. Recall from our previous definition that duplicates will always be sent to the right subtree. Although it does not matter which side duplicates go on, we prefer it to be consistent so that our k-d tree operations can be fully efficient. However, replacing the deleted node with the largest discriminator value in the left subtree does not guarantee this invariant. For instance, consider the following k-d tree:



If we attempt to delete B by replacing the deleted node with the descendant in the left subtree with the largest discriminator, we would get the following. However, this tree is invalid per the rules we initially defined for our k-d tree, since A and C share the same x-coordinate but C is to the left of A!



By swapping the left subtree so that it becomes the new right subtree, we ensure that duplicate discriminator values that were previously in the left subtree correctly end up in the right subtree after we perform the delete.



Similar to insert, the amount of work we need to complete during a deletion depends on the height of the tree. In the worst-case, you could have a stick, which could require you to traverse every node of the tree before you eventually delete a leaf node — this would yield a worst-case time complexity of $\Theta(n)$ for a single deletion (where n is the number of points in the tree). However, much like insertion, you can expect the tree to be fairly balanced in the average case, and the average-case time complexity of deletion again turns out to be $\Theta(\log(n))$. A summary of k-d tree operations is provided in the table below.

	Not Balanced		**Balanced**	
	Average Case	**Worst Case**	**Average Case**	**Worst Case**
Time Complexity of Search	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Time Complexity of Insert	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Time Complexity of Delete	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Auxiliary Space	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

The efficiency of k-d trees in representing a multidimensional space makes it a versatile tool for solving geometrical problems. One such problem is the *nearest neighbor search* problem, which seeks to find the point in a pointset that is nearest to a given query point. k-d trees are also useful for performing *range searches* over multiple parameters. For example, if we are given a list of EECS 281 students with their exam scores and the number of credits they are taking, we could perform a k-d tree range search to efficiently find the set of all students that are taking over 15 credits and have a score of 93 or higher on their EECS 281 exam.

26.10 Segment Trees (*)

※ 26.10.1 *Segment Trees in One Dimension (*)*

Another tree data structure that is useful for working with ranges is the **segment tree**, which allows you to efficiently query and update values within a specific interval of data. Segment trees are useful for solving problems that require you to query some information within a given range of values (such as the sum, or the minimum and maximum value) that may be mutable in between different queries. To understand the motivation behind why a segment tree may be useful, consider the following example:

Example 26.9 You are given an array of integers `nums`. Implement a class that can handle the following two updates to this array:

- `update_index(size_t idx, int32_t new_value)`, which updates the value at `nums[idx]` to `new_value`.
- `get_min_value_in_range(size_t left, size_t right)`, which returns the minimum value within the index range from `left` to `right`, inclusive.

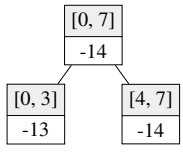
Example: Suppose you are given the array `[4, 7, -13, 19, -12, -14, 1, 10]`. Calling `get_min_value_in_range(2, 4)` would return `-13`, which is the smallest value within the range from index 2 to 4 (or `[-13, 19, -12]`). However, if you were to then call `update_index(3, -15)`, the value at index 3 would change from `19` to `-15`, which would mean that the minimum value in the range [2, 4] would now become `-15` instead of `-13`.

A simple approach toward solving this problem would be to store the values in an array and update the values directly in $\Theta(1)$ time whenever `update_index()` is called. However, this would mean that queries to find the minimum value in a range would each need to take $\Theta(n)$ time, since you would have to perform a linear pass to find the minimum value with every query (recomputation is necessary because values may be changed between queries). A similar problem also applies if you try to cheapen the cost of queries: you could try to precompute the minimum values of all ranges so that queries can take $\Theta(1)$ time, but this would cause updates to take $\Theta(n)$ time since you would also need to perform a linear pass to update your precomputed values. This is where the segment tree comes into play: we can use this data structure to ensure that updates and queries both take $\Theta(\log(n))$ time. To construct a segment tree, we will need to identify the following two things:

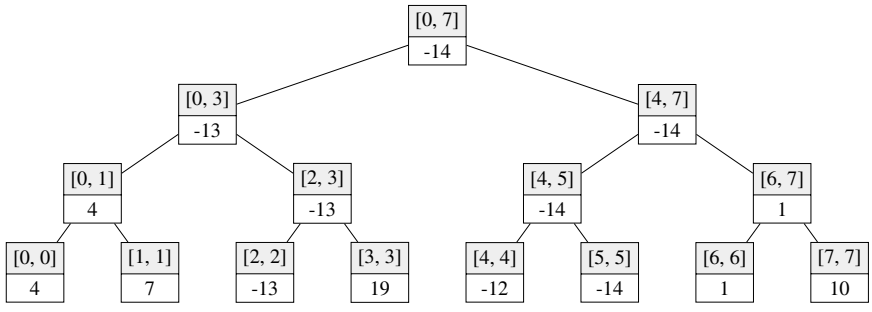
1. The *value* that should be stored at each node of the segment tree. This is typically the value that we want to query for.
2. The *merge operation* that will be used to merge two siblings in the segment tree together. This is used if a query specifies a range that spans multiple nodes of the segment tree. We will discuss this process in more detail later.

In this example, the value we want to query for is the minimum value of a range. To construct our segment tree, we will start by computing the minimum value of the entire array, from index 0 to $n-1$. This is inserted as the root of the segment tree (note that only -14 is actually inserted into the tree; [0, 7] just indicates that this node represents the minimum value between indices 0 and 7 to make understanding the tree easier).

Next, we will split the array into two halves consisting of the index ranges of $[0, n/2-1]$ and $[n/2, n-1]$. The minimum value is computed for each of these ranges and inserted as the left and right subtrees of the root.



This process of halving the index ranges continues until all the ranges only contain a single value, each of which comprises a leaf of the segment tree. The final segment tree created from the previous example is shown below:



Notice that the first level has a single node, the second level contains two nodes, the third level contains four nodes, and so on, until the number of nodes in a level reaches n, the original size of the array. Therefore, we can conclude that the size of a segment tree for n values is bounded above by $4n$ using the following sum (this inequality was determined using the sum of a finite geometric series):

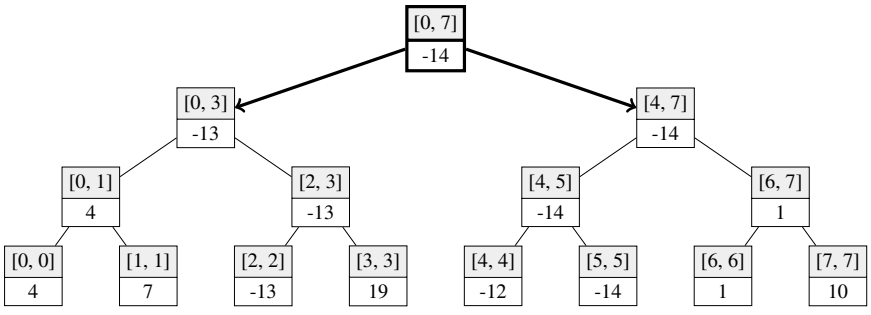
$$1+2+4+\ldots+2^{\lceil \log_2(n) \rceil} < 2^{\lceil \log_2(n) \rceil + 1} < 4n$$

To implement our segment tree, we will also need to devise a method that can be used to merge two nodes in the segment tree together. This is needed if a query requires information from multiple nodes in our segment tree (e.g., a query for the interval [2, 4] would need to combine the information stored in the nodes of [2, 3] and [4, 4]). For our example, our merge operation just needs to take the minimum value of the two nodes we want to combine (e.g., the solution for [2, 4] is the minimum of the solution for [2, 3] and [4, 4]).

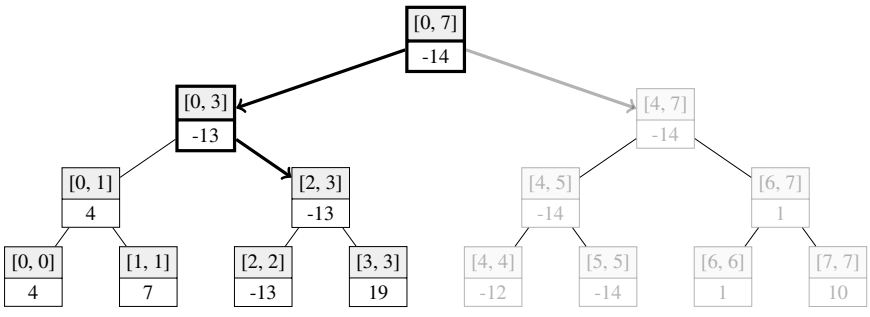
With this information, we are now able to query the minimum value of any range in worst-case $\Theta(\log(n))$ time. If we are given any index range, there are three outcomes that can happen:

1. The range matches the segment of a node in the segment tree. In this case, you can just return the value associated with this node.
2. The range falls entirely within the domain of either the left or right child (i.e., in the left or right half of values). In this case, recurse into the child that covers the range.
3. The range is split across both the left and right children. In this case, you will have to make two recursive calls: one into the left child with its portion of the range, and one into the right child with its portion of the range. Then, combine the two solutions together to get the solution for the original range.

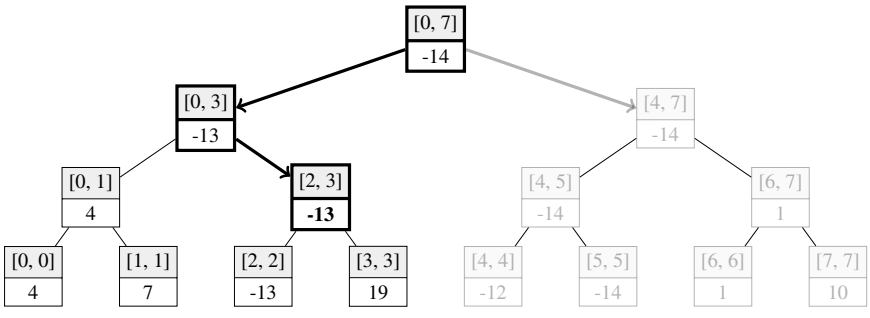
For example, consider what happens when we make a query to find the minimum value within the index range [2, 4]. We start at the root of the tree and see which side the range [2, 4] belongs to. Because the range [2, 4] is split between both the left and right children, we will perform two recursive calls, one on the left child, and one on the right child.



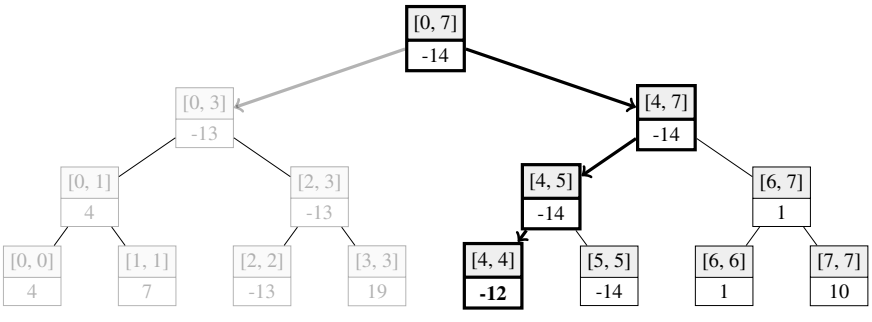
Let's first consider the recursive call on the left child, which stores the range of [0, 3]. The portion of our original range that falls in the left half is [2, 3], which falls entirely in the right half of the range [0, 3]. Thus, we will perform a recursive call on the right child with this range of [2, 3].



We have encountered a node associated with the interval range of [2, 3], which matches the range that we were looking for. Therefore, the value of this node, -13, is the solution of the original recursive call on the root's left child.

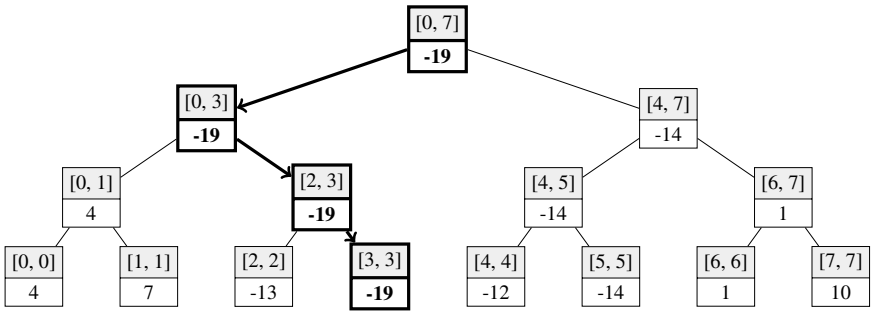


We can repeat this process for the right subtree, as shown. The solution of this recursive call is -12.



The minimum sum in the range [2, 3] is -13, and the minimum sum in the range [4, 4] is -12. We can combine these two solutions to get the minimum sum in the desired range of [2, 4], which is min(-13, -12) = -13.

Updating a value in the array also takes $\Theta(\log(n))$ time in a segment tree. This is because any element in the original data is only associated with a single node at each level of the tree. Since there are only $\Theta(\log(n))$ levels in the tree, each update would only need to change $\Theta(\log(n))$ nodes in the tree, which takes $\Theta(\log(n))$ time if each update takes constant time. An example is shown for the given example: if we change the value at index 3 from `19` to `-19`, only the following nodes of the segment tree need to be updated:



Now that we have the basic structure of how our segment tree can be used to solve the problem, we can begin implementing it in code. To initially build our tree, we can use recursion to construct the values of the segment tree in a postorder fashion. This is done by recursively building the values of a node's two children and then merging the results together to get the value of the current node. An example is shown below (note that we store our segment tree as a binary tree that is represented as an 1-indexed array, which allows us to access the left and right children of the node at index i using index $2i$ and $2i+1$, respectively — see section 18.2.1 for more detail on this procedure).

```
1   class SegmentTree {
2   private:
3     std::vector<int32_t> segment_tree;  // segment tree represented using an array (1-indexed)
4     std::vector<int32_t> values;        // the values we want to query range data from
5
6     // helper function that can be used to construct a segment tree node at curr_idx of the array
7     void build_segment_tree(size_t curr_idx, size_t left, size_t right) {
8       if (left == right) {  // base case of single index in range (leaf node)
9         segment_tree[curr_idx] = values[left];
10      } // if
11      else {
12        size_t middle = left + (right - left) / 2;
13        // recursively build left subtree (left child of idx is located at 2 * idx)
14        build_segment_tree(2 * curr_idx, left, middle);
15        // recursively build right subtree (right child of idx is located at 2 * idx + 1)
16        build_segment_tree(2 * curr_idx + 1, middle + 1, right);
17        // combine the solutions to get the value of the node at idx
18        segment_tree[curr_idx] = std::min(segment_tree[2 * curr_idx], segment_tree[2 * curr_idx + 1]);
19      } // else
20    } // build_segment_tree()
21
22  public:
23    // Segment tree ctor, the segment tree array is initialized to a size of 4n since this is the
24    // maximum potential size needed (the vector is also initialized to the largest possible
25    // integer since we do not want unused positions to interfere in finding the smallest value)
26    SegmentTree(const std::vector<int32_t>& nums)
27      : segment_tree(4 * nums.size(), std::numeric_limits<int32_t>::max()), values(nums) {
28      // 1 is passed in as the initial vertex since that is index of the root, which we start from
29      build_segment_tree(1, 0, nums.size() - 1);
30    } // SegmentTree()
31  };
```

To implement `get_min_value_in_range()`, we will use the input range to decide which side of the tree to recurse into. If the input range matches exactly with a range in our segment tree, we will return its value directly. Otherwise, we will search in the left and/or right subtrees to get the solution(s), combining them if necessary. One possible implementation of this method is shown below:

```
class SegmentTree {
private:
  /* ... other members same as before ... */

  // helper function, seg_left and seg_right represent the boundaries of the current segment
  // tree node, and query_left and query_right represent the boundaries of the current query
  int32_t get_min_value_helper(size_t curr_idx, size_t seg_left, size_t seg_right,
                               size_t query_left, size_t query_right) {
    // query range matches range of segment tree node, so return the node's value
    if (query_left == seg_left && query_right == seg_right) {
      return segment_tree[curr_idx];
    } // if
    size_t seg_middle = seg_left + (seg_right - seg_left) / 2;
    // query entirely in left side, so recurse into left child
    if (query_right <= seg_middle) {
      return get_min_value_helper(2 * curr_idx, seg_left, seg_middle, query_left, query_right);
    } // if
    // query entirely in right side, so recurse into right child
    if (query_left > seg_middle) {
      return get_min_value_helper(2 * curr_idx + 1, seg_middle + 1, seg_right,
                                  query_left, query_right);
    } // if
    // else recurse into both and take the minimum (i.e., merge the two results together)
    return std::min(
      get_min_value_helper(2 * curr_idx, seg_left, seg_middle, query_left, seg_middle),
      get_min_value_helper(2 * curr_idx + 1, seg_middle + 1, seg_right, seg_middle + 1, query_right)
    );
  } // get_min_value_helper()

public:
  /* ... other members same as before ... */
  int32_t get_min_value_in_range(size_t left, size_t right) {
    return get_min_value_helper(1, 0, values.size() - 1, left, right);
  } // get_min_value_in_range()
};
```

To update a value at a specific index, we will recurse down the branch of the segment tree that contains the index we want to update. This can also be done in a postorder fashion, as shown in the implementation below:

```
class SegmentTree {
private:
  /* ... other members same as before ... */
  void update_index_helper(size_t curr_idx, size_t seg_left, size_t seg_right,
                           size_t idx_to_update, int32_t new_value) {
    if (seg_left == seg_right) {  // base case of single index in range (leaf node)
      segment_tree[curr_idx] = new_value;
    } // if
    else {
      size_t seg_middle = seg_left + (seg_right - seg_left) / 2;
      if (idx_to_update <= seg_middle) {
        // update left child
        update_index_helper(2 * curr_idx, seg_left, seg_middle, idx_to_update, new_value);
      } // if
      else {
        // update right child
        update_index_helper(2 * curr_idx + 1, seg_middle + 1, seg_right, idx_to_update, new_value);
      } // else
      // combine solutions
      segment_tree[curr_idx] = std::min(segment_tree[2 * curr_idx], segment_tree[2 * curr_idx + 1]);
    } // else
  } // update_index_helper()

public:
  /* ... other members same as before ... */
  void update_index(size_t idx, int32_t new_value) {
    update_index_helper(1, 0, values.size() - 1, idx, new_value);
  } // update_index()
};
```

This concludes our implementation of the problem. By using a segment tree, we can ensure that values can be updated and that the minimum value can be queried from any range in worst-case $\Theta(\log(n))$ time! This is better than the alternative options of recomputing the minimum value during every query or during every update, both of which involve an operation that takes $\Theta(n)$ time.

Example 26.10 In finance, an *option* is a contract that gives its owner the right to buy or sell a quantity of an underlying asset at a specific price on or before a specified date. This date is known as the option contract's *expiration date*, and the distance to expiration plays a role in determining that option's value and overall risk. For this problem, implement a class that can be used to efficiently retrieve a trader's total position on an asset's options that expire within a given time range (note that there are multiple types of options contracts in real life, but for the sake of simplicity, we will consider all options together in this problem). An outline of this class is shown below:

```
class OptionsExpirationManager {
private:
  int32_t farthest_expiration;
  // TODO: Add any data structures here!
public:
  // Constructor takes in the number of days until the farthest expiration
  OptionsExpirationManager(int32_t farthest_expiration_in)
    : farthest_expiration{farthest_expiration_in} {}

  // This method takes in an asset name, the number of days until that option expires
  // (<= farthest_expiration), and an integer quantity (can be negative if options are
  // sold), and places a trade for 'quantity' options with that asset name and expiration
  void make_trade(const std::string& asset_name, int32_t days_to_exp, int32_t quantity);

  // This method takes in an asset name and two integers, and returns
  // the total position of all assets with expiration in [start, end] days
  int32_t get_position(const std::string& asset_name, int32_t start, int32_t end);
};
```

Example: Consider the following sequence of operations, where the farthest expiration occurs 7 days from now:

- `make_trade("QQQ", 4, 100)`: makes a trade for 100 QQQ options that expire in 4 days.
- `make_trade("QQQ", 2, -200)`: makes a trade for -200 QQQ options that expire in 2 days.
- `make_trade("QQQ", 5, 300)`: makes a trade for 300 QQQ options that expire in 5 days.
- `make_trade("QQQ", 6, -100)`: makes a trade for -100 QQQ options that expire in 6 days.

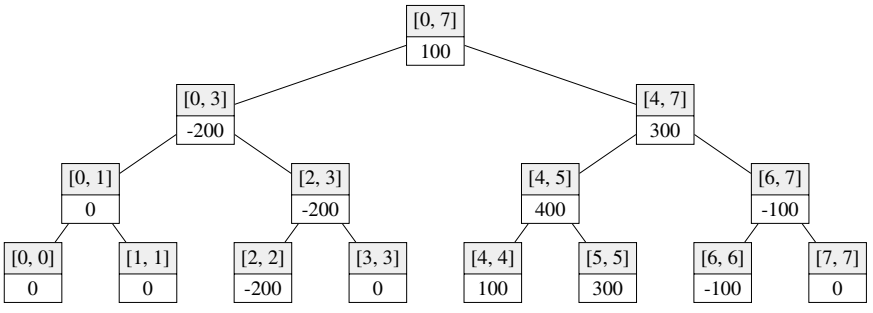
At this point, calling `get_position("QQQ", 3, 6)` would return the total number of options in the position that expire between 3 and 6 days from now, inclusive, which would be 100 + 300 - 100 = 300.

Then, if we make the following trade:

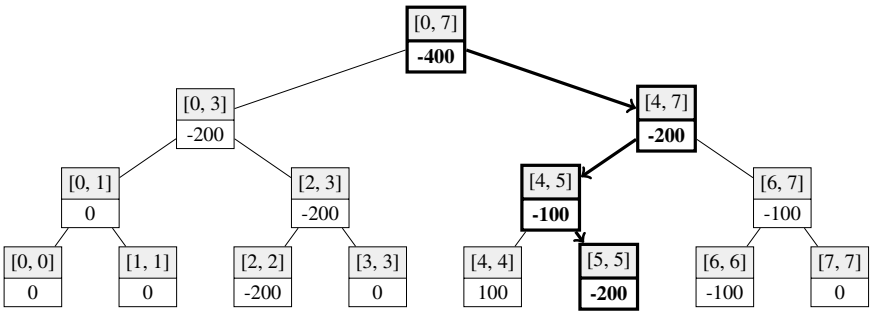
- `make_trade("QQQ", 5, -500)`: makes a trade for -500 QQQ options that expire in 5 days.

our total position in options that expire in 5 days drops from 300 to -200. Thus, any subsequent calls to `get_position("QQQ", 3, 6)` would return a new position of 100 - 200 - 100 = -200.

Similar to before, this problem requires us to efficiently query information about a range of data that may be modified. However, unlike the previous problem, this question involves querying the *sum* of a range of data rather than the minimum value. Therefore, each node of our segment tree will need to store the sum of a given range instead of its minimum value, and the merge operation will have to sum up the values of the two nodes we want to combine. An illustration of the segment tree constructed using the provided example is shown below (first four trades):



We will start our implementation with the `make_trade()` function. When a trade is made on a specific expiration, we recurse down the branch of the segment tree that contains the expiration we want to update and add the new quantity to the existing position. The following illustrates what happens when we make a -500 trade on QQQ options that expire in 5 days:



This behavior is implemented below (note that this is very similar to our implementation in the previous example with the minimum value):

```
class OptionsExpirationManager {
private:
  int32_t farthest_expiration;
  // each asset name needs its own segment tree, so we will use an unordered map
  std::unordered_map<std::string, std::vector<int32_t>> segment_trees_by_asset;

  void trade_helper(const std::string& asset_name, size_t curr_idx, size_t seg_left,
                    size_t seg_right, size_t idx_to_update, int32_t quantity) {
    // initialize segment tree if no trades have been made for the asset yet
    auto segment_tree_it = segment_trees_by_asset.find(asset_name);
    if (segment_tree_it == segment_trees_by_asset.end()) {
      segment_trees_by_asset[asset_name].resize(4 * farthest_expiration);
    } // if
    auto& segment_tree = segment_trees_by_asset[asset_name];
    if (seg_left == seg_right) {
      segment_tree[curr_idx] += quantity;
    } // if
    else {
      size_t seg_middle = seg_left + (seg_right - seg_left) / 2;
      if (idx_to_update <= seg_middle) {
        trade_helper(asset_name, 2 * curr_idx, seg_left, seg_middle, idx_to_update, quantity);
      } // if
      else {
        trade_helper(asset_name, 2 * curr_idx + 1, seg_middle + 1,
                     seg_right, idx_to_update, quantity);
      } // else
      // combine solutions
      segment_tree[curr_idx] = segment_tree[2 * curr_idx] + segment_tree[2 * curr_idx + 1];
    } // else
  } // trade_helper()

public:
  // Constructor takes in the number of days until the farthest expiration
  OptionsExpirationManager(int32_t farthest_expiration_in)
    : farthest_expiration(farthest_expiration_in) {}

  void make_trade(const std::string& asset_name, int32_t days_to_exp, int32_t quantity) {
    trade_helper(asset_name, 1, 0, farthest_expiration - 1, days_to_exp, quantity);
  } // make_trade()
};
```

The `get_position()` function can be implemented similarly to the `get_min_value_in_range()` function in the previous example. We will use the input range to determine which side of the tree to recurse into, and if the input range spans multiple intervals, we will recurse into each of those intervals and combine their solutions. This is implemented below:

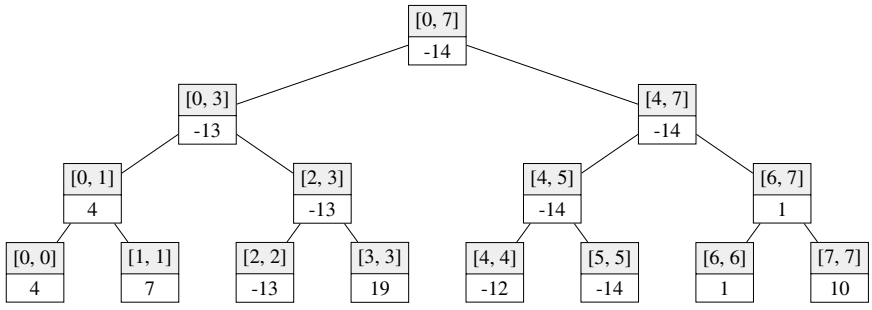
```
class OptionsExpirationManager {
private:
  int32_t farthest_expiration;
  // each asset name needs its own segment tree, so we will use an unordered map
  std::unordered_map<std::string, std::vector<int32_t>> segment_trees_by_asset;

  int32_t position_helper(const std::string& asset_name, size_t curr_idx, size_t seg_left,
                          size_t seg_right, size_t query_left, size_t query_right) {
    // return 0 if no trades have been made for the asset yet
    auto segment_tree_it = segment_trees_by_asset.find(asset_name);
    if (segment_tree_it == segment_trees_by_asset.end()) {
      return 0;
    } // if
    if (query_left == seg_left && query_right == seg_right) {
      return segment_tree_it->second[curr_idx];
    } // if
    size_t seg_middle = seg_left + (seg_right - seg_left) / 2;
    if (query_right <= seg_middle) {
      return position_helper(asset_name, 2 * curr_idx, seg_left,
                             seg_middle, query_left, query_right);
    } // if
    if (query_left > seg_middle) {
      return position_helper(asset_name, 2 * curr_idx + 1, seg_middle + 1,
                             seg_right, query_left, query_right);
    } // if
    return position_helper(asset_name, 2 * curr_idx, seg_left, seg_middle, query_left, seg_middle) +
           position_helper(asset_name, 2 * curr_idx + 1, seg_middle + 1,
                           seg_right, seg_middle + 1, query_right);
  } // position_helper()

  /* ... continued on next page ... */
```

```
32  public:
33    // Constructor takes in the number of days until the farthest expiration
34    OptionsExpirationManager(int32_t farthest_expiration_in)
35      : farthest_expiration(farthest_expiration_in) {}
36
37    int32_t get_position(const std::string& asset_name, int32_t start, int32_t end) {
38      return position_helper(asset_name, 1, 0, farthest_expiration - 1, start, end);
39    } // get_position()
40  };
```

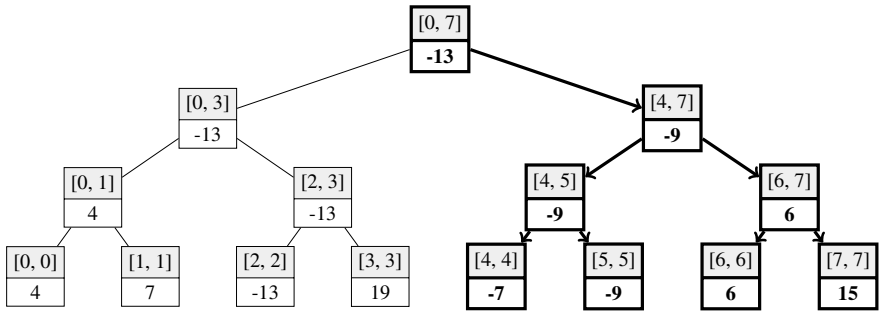
With the help of a segment tree, both `make_trade()` and `get_position()` are able to run in worst-case $\Theta(\log(n))$ time, where n is the number of days until the farthest expiration.

※ 26.10.2 *Lazy Propagation Segment Trees* (✻)

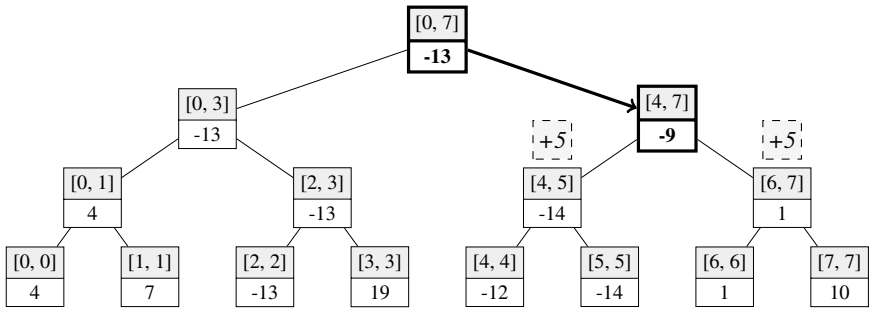
In the previous section, we discussed a segment tree implementation that can support efficient updates to a single value. However, what if we wanted to update an entire range of values (for example, add 10 to every value in the range from index 1 to 5)? Using the previous implementation, we would need to call the update function multiple times, once for every index that needs to be updated. Given n values to update, this would take $\Theta(n\log(n))$ time. It turns out that this is not the most efficient way to do things: we can speed up range updates to $\Theta(\log(n))$ time by using a strategy known as **lazy propagation**, which postpones value updates until they are actually required. To illustrate how this works, consider our first example of `[4, 7, -13, 19, -12, -14, 1, 10]`, where our segment tree stores the minimum in a range.



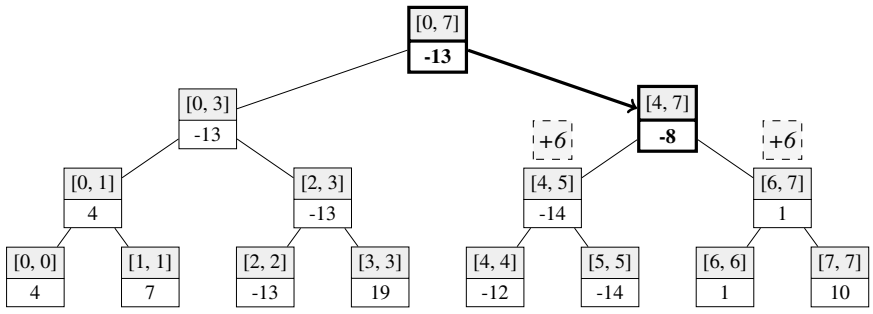
Suppose we wanted to increment all the values in the index range [4, 7] by 5. Without lazy propagation, our segment tree would look this after the update, which would take $\Theta(n\log(n))$ time. A lot of this work, however, can be avoided.



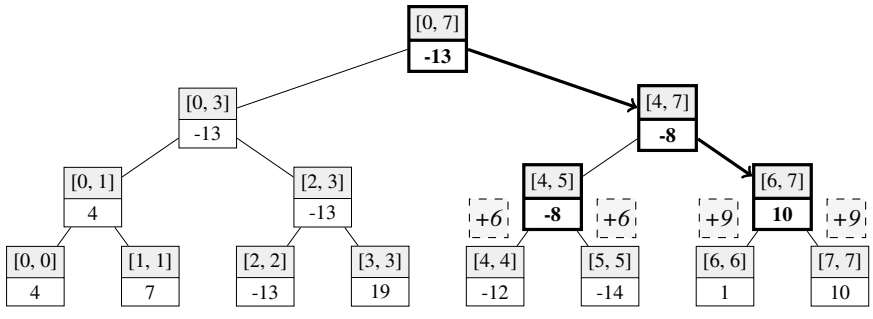
With lazy propagation, we only update a node if we need to use its value during a query. In our example, we only need to reference the values up to the [4, 7] node in the right subtree, so we do not need to recurse any further into its children. Instead, we mark them with a "+5" value to indicate to future queries that an increment of 5 should be applied if the value of those nodes are ever needed. By using this strategy, we are able to drop the time complexity of each range update from $\Theta(n\log(n))$ to $\Theta(\log(n))$, as the number of updates required is now proportional to the height of the segment tree.



Increments that have not yet been applied to the tree are additive. For instance, if the interval range [4, 7] were incremented by 1 after the previous +5 update, the children of [4, 7] would have their values updated from +5 to +6.



Whenever we need to use the value of a node during a query, we must make sure to propagate any changes that may still be pending. For instance, suppose we make a query to increment each value in the range [6, 7] by 3. When recursing into the segment tree, we end up encountering the [6, 7] interval, which still has a pending update of +6. Thus, we will need to apply the +6 in addition to the +3 requested in the current query, for a total of +9. Since we returned the value in the [6, 7] node, we do not need to further recurse into any of its children, so we will assign each child with a "+9" value to reflect changes that will need to be made in the future. (Notice that the node associated with the [4, 5] interval ended up being updated as well, since its value was needed to recompute the [4, 7] node after the [6, 7] node was modified. Its children, however, do not overlap with our initial query of [4, 7], so we do not need to update them immediately.)



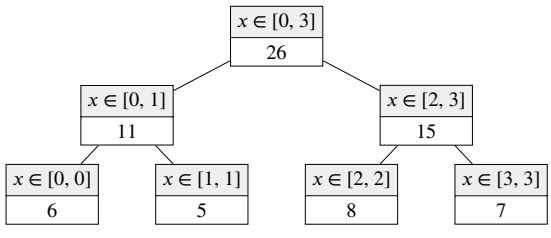
Lazy propagation segment trees are implemented similarly to the standard segment trees we introduced previously. However, in addition to an array of size $4n$ to represent the segment tree, we also keep track of a separate array of size $4n$ to store pending updates (for instance, any pending updates to the node at index 5 of the segment tree array is also stored at index 5 of this additional array). Then, in our query and update functions, we must look into this additional array and apply any pending changes before we use the value of any node in our segment tree.

※ 26.10.3 *Segment Trees in Multiple Dimensions (✻)*

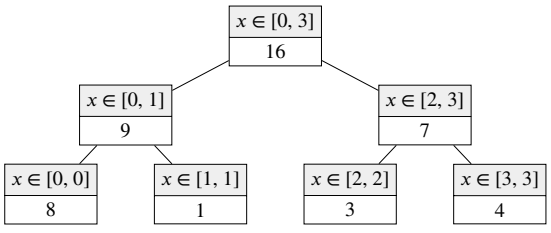
Segment trees can also be used to represent data in multiple dimensions. This can be done using a segment tree of segment trees, one for each dimension of the data. For example, consider the following matrix, for which we want to support efficient sum queries and updates to any given submatrix (such as the one outlined below, for which a query would return a sum of $7 + 2 + 1 + 3 = 13$).

1	3	6	9
4	7	2	5
8	1	3	4
6	5	8	7

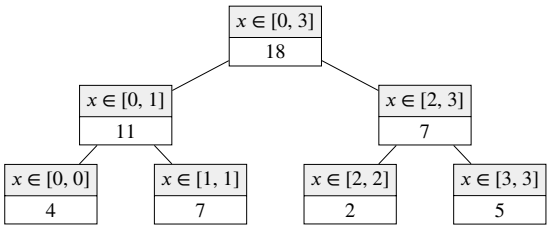
To build a two-dimensional segment tree, we start by constructing one-dimensional segment trees on one of the dimensions while keeping the other dimension fixed. For our example, we will keep y fixed and build a segment tree for each value of x (i.e., for each row of the matrix). The following is the segment tree for the row $y = 0$ (`[6, 5, 8, 7]`), where each node stores the sum of an interval of x:



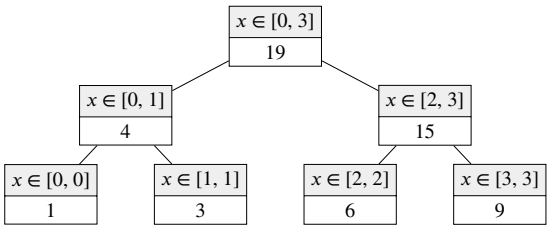
The following is the segment tree for the row $y = 1$ (`[8, 1, 3, 4]`):



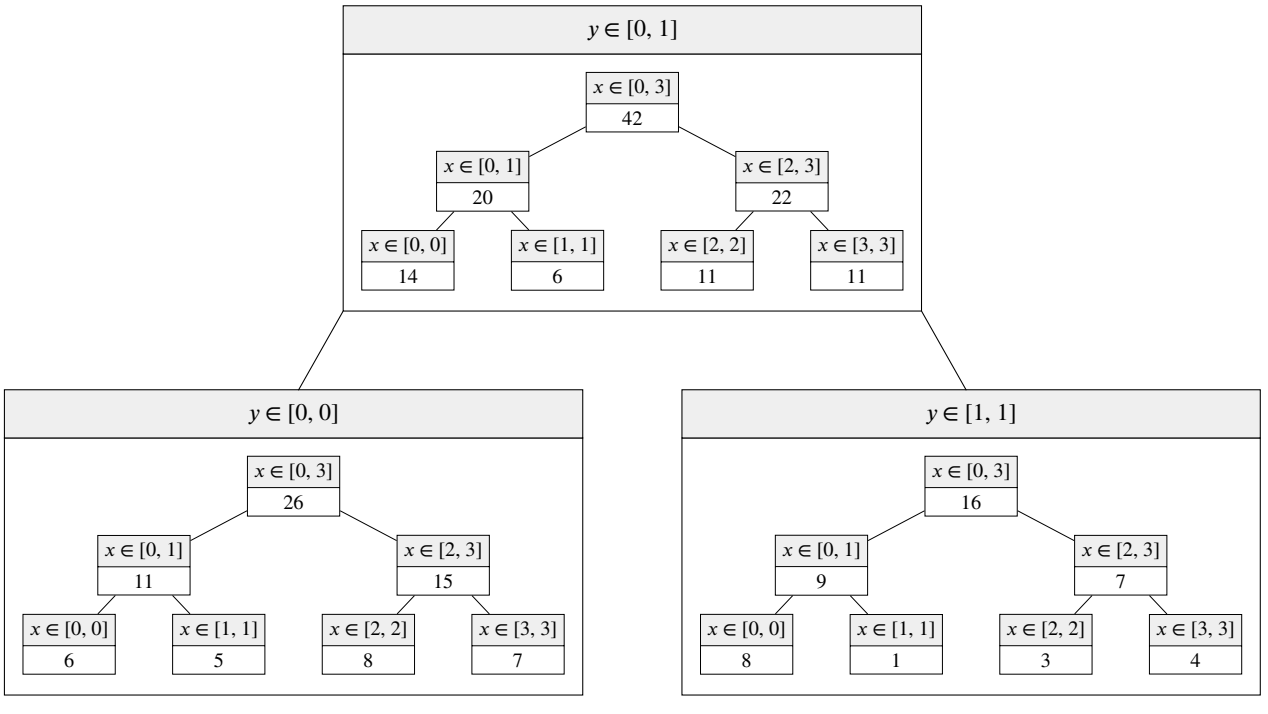
The following is the segment tree for the row $y = 2$ (`[4, 7, 2, 5]`):



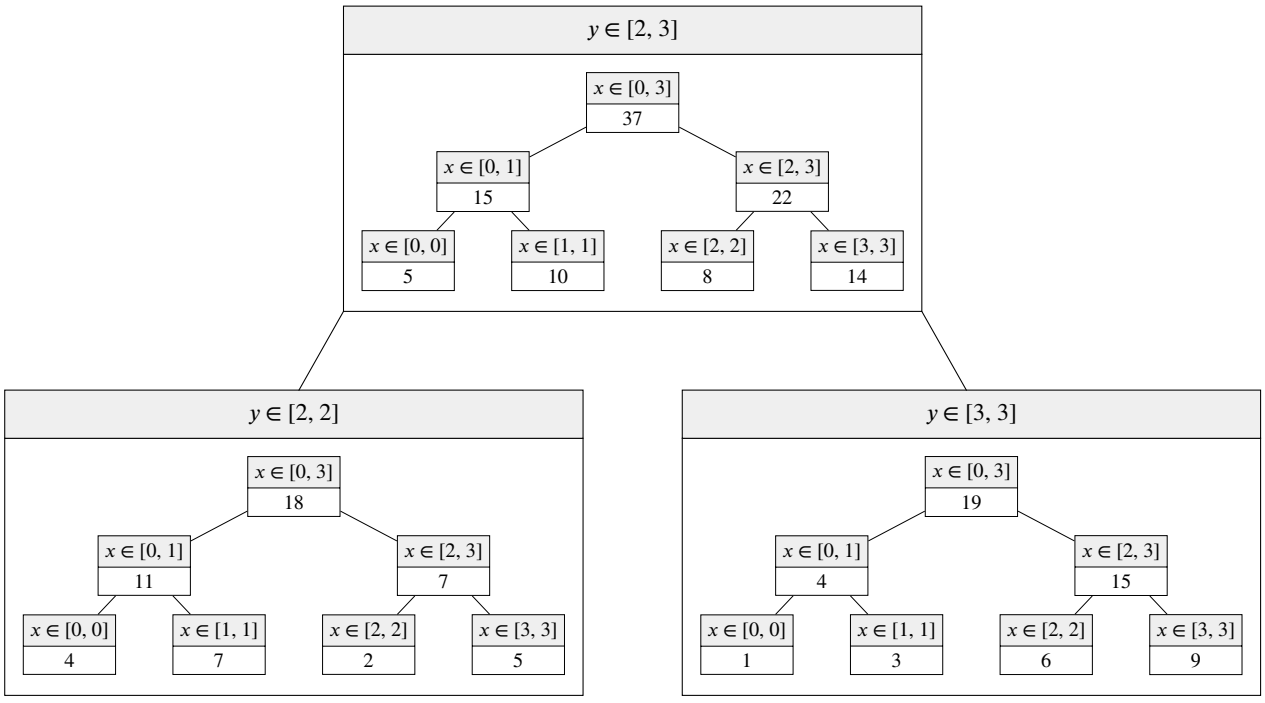
The following is the segment tree for the row $y = 3$ (`[1, 3, 6, 9]`):



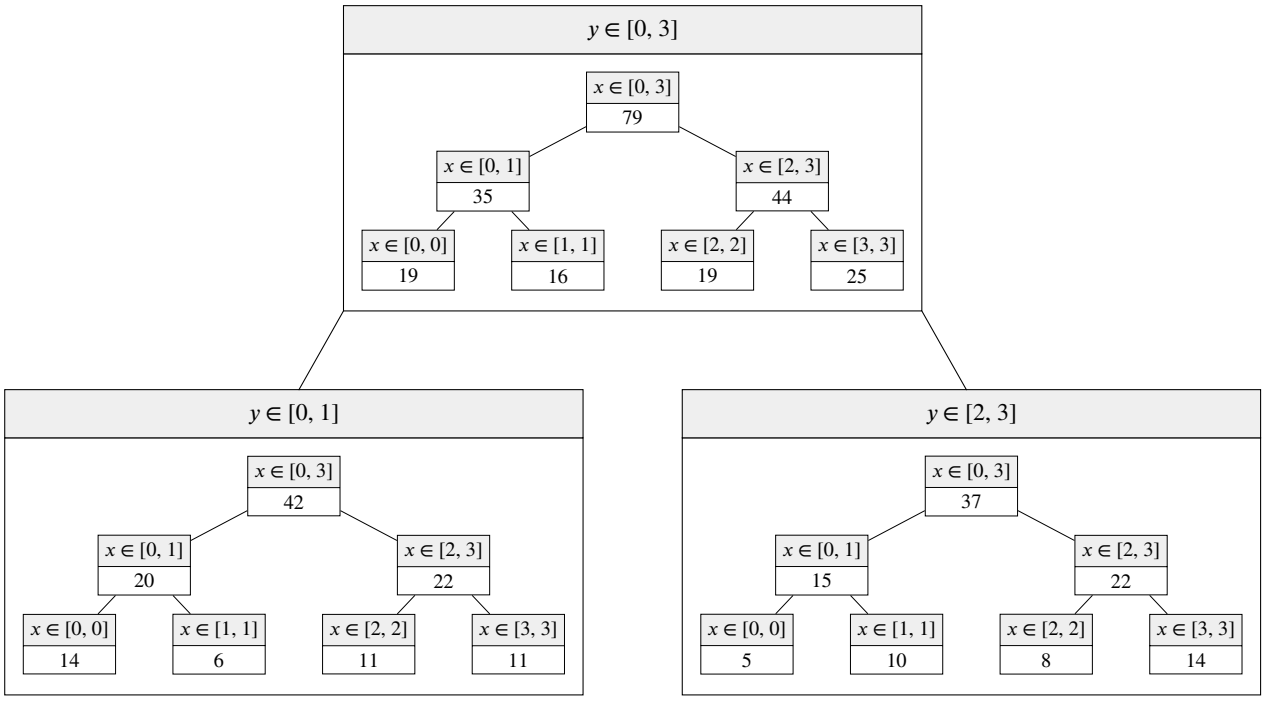
Then, we will merge these segment trees together to get new segment trees that correspond to intervals of the dimension we fixed (in this case, y). For instance, merging together the segment trees for $y = 0$ and $y = 1$ would give us a segment tree for the interval $y \in [0, 1]$.



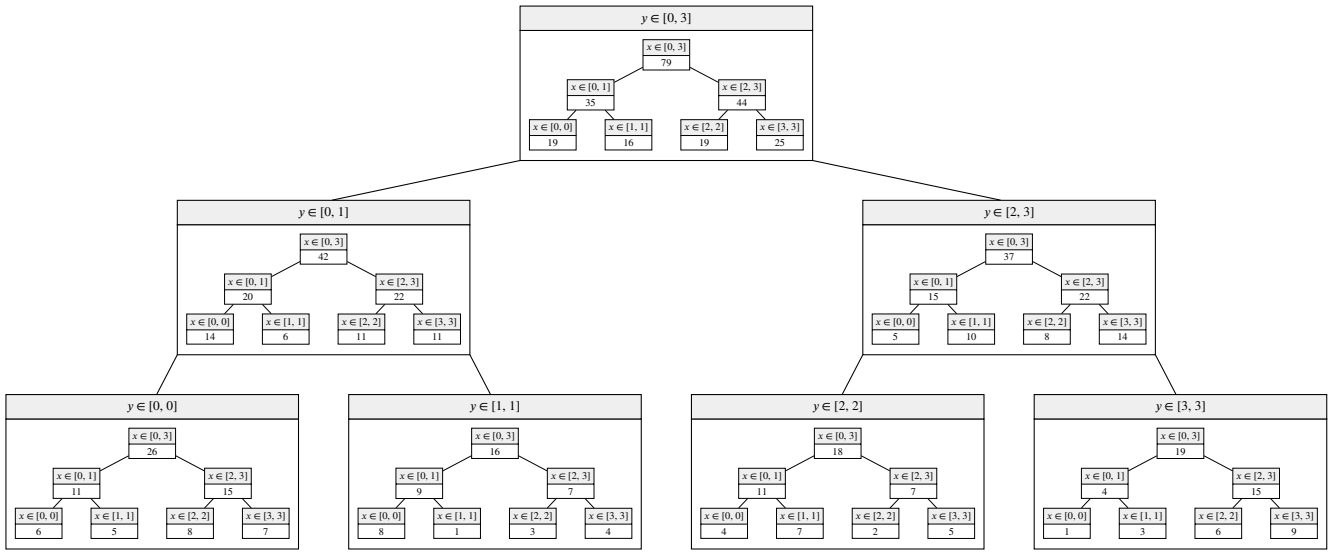
Similarly, merging together the segment trees for $y = 2$ and $y = 3$ would give us a segment tree for the interval $y \in [2, 3]$.



We can then merge together the segment trees for $y \in [0, 1]$ and $y \in [2, 3]$ to get a combined segment tree for $y \in [0, 3]$, as shown.



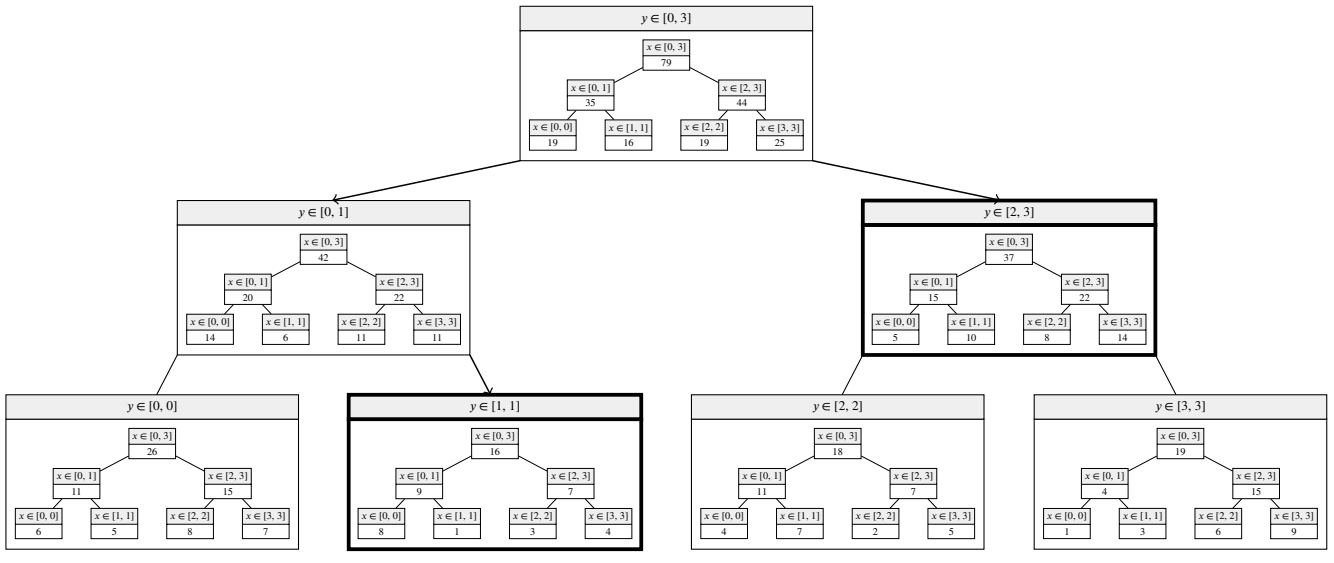
This gives us our final segment tree for the entire two-dimensional matrix, which is shown below:



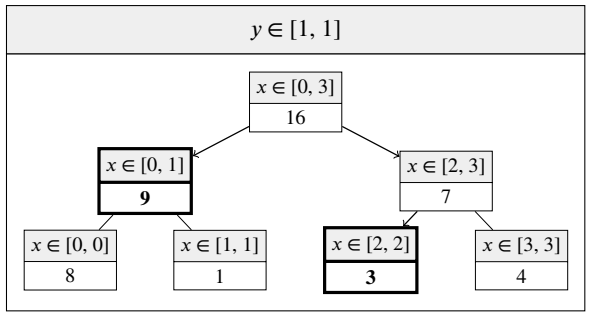
This segment tree can now be used to efficiently query the sum of any region of our original matrix. To illustrate, let us consider an example of summing the region spanning $x \in [0,2]$ and $y \in [1,3]$:

1	3	6	9
4	7	2	5
8	1	3	4
6	5	8	7

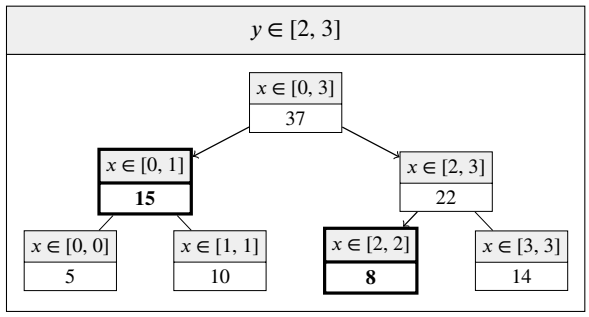
Since the y region we want to query ($y \in [1,3]$) spans the intervals of both the left ($y \in [0,1]$) and right ($y \in [2,3]$) subtrees, we will need to recurse into both sides of the segment tree to compute our solution. In the left subtree, we end up recursing into the node with interval $y \in [1,1]$, and in the right subtree, we end up recursing into the node with interval $y \in [2,3]$.



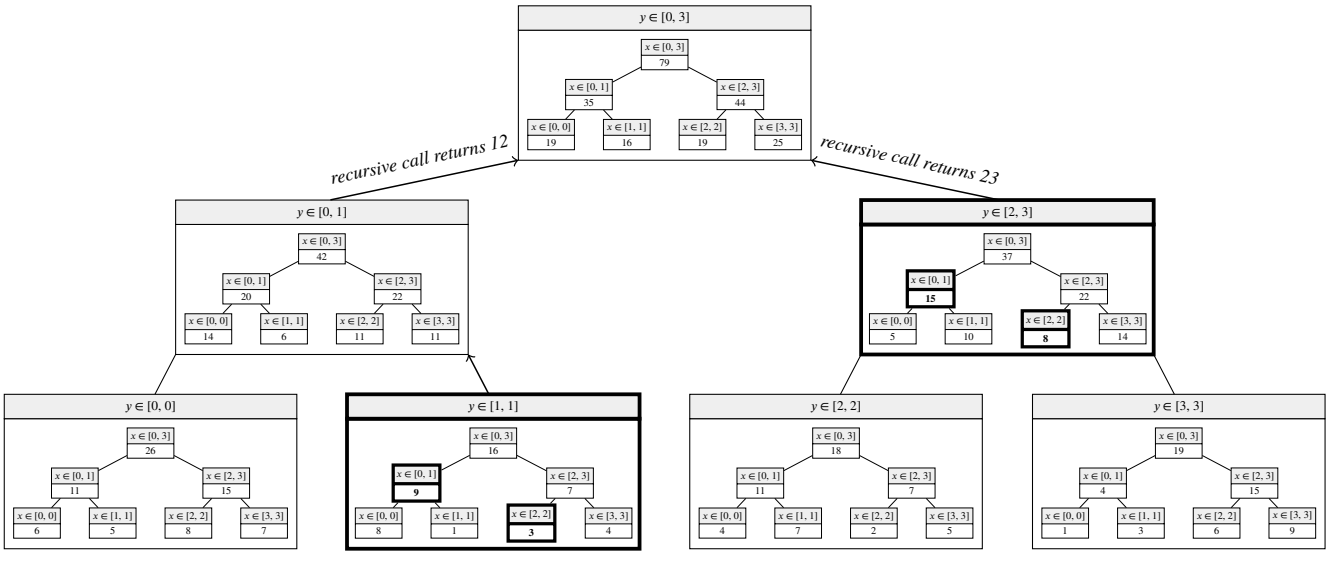
In these two smaller segment trees, we will query the sum using the x dimension of our desired range. Since the x dimension of our initial query spans the the interval $[0, 2]$, we end up recursing into the node associated with the interval $(x \in [0,1])$ in the left subtree, and the node associated with the interval $(x \in [2,2])$ in the right subtree. For the $y \in [1,1]$ segment tree node, this gets us a value of $9 + 3 = 12$.



For the $y \in [2,3]$ segment tree node, this gets us a value of $15 + 8 = 23$.



Combining these two results together, we get a total sum of $12 + 23 = 35$, which is the solution to our initial query.



Queries in this two-dimensional segment tree take $\Theta(\log(m)\log(n))$ time in the worst case, where m and n are the dimensions of the data stored in the segment tree. This is because a query will need to descend down the outer tree for the first coordinate, and then, for each traversed vertex in this outer tree, it may need to descend down its inner tree for the second coordinate. Both of these descents take logarithmic time on the number of nodes in its tree, which is on the order of m for one dimension and on the order of n for the other.

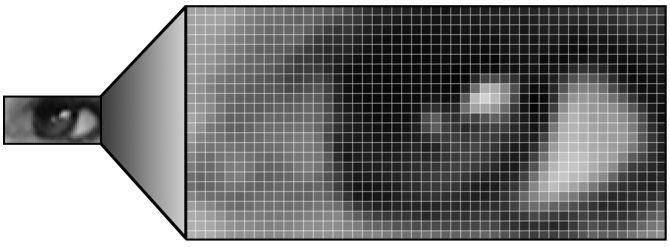
While not discussed in detail here, the time complexity of updating this two-dimensional segment tree is also $\Theta(\log(m)\log(n))$ in the worst case for the same reason. This is because a single update to the tree will only affect nodes whose interval includes the coordinate to be updated, and the process of searching for these nodes follows the same method that we used while querying for a value.

26.11 Images and Graphics

※ 26.11.1 *Raster and Vector Graphics*

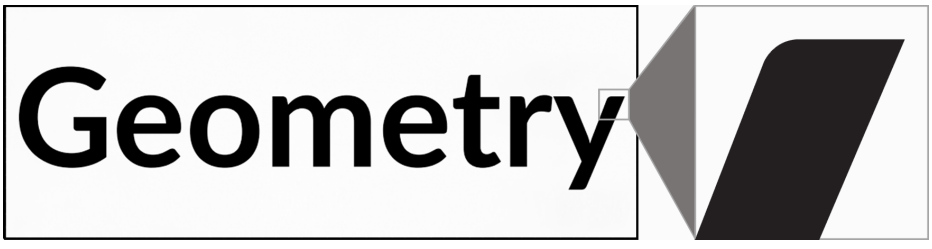
In this final section, we will discuss one of the most prevalent features of our digital landscape: the world of images and graphics. Although this concept may feel a bit disjoint from all of the other topics we have discussed so far, the study of computer graphics is nonetheless one of the core components of computational geometry, and also one of the most relevant in practice. (As a disclaimer, we are not going to delve too deep into the vast scope of computer graphics in this section, as advanced exploration of this field is more fitting for a technical class beyond EECS 281. Rather, the goal is to provide a brief, high-level overview of how visual data can be represented in two dimensions.)

There are two main categories that image files fall into: *raster graphics* and *vector graphics*. The more common form is **raster (bitmap) graphics**, which represent images on a screen using tiny colored pixels that, when combined together, form the basis of more detailed images such as photographs. The greater the number of pixels involved, the higher the quality of the overall image (and vice versa). A vast quantity of graphics files that we deal with on a daily basis (such as JPEG, PNG, and GIF) use raster graphics to represent their underlying visual content.



Raster graphics are created using pixels that are arranged to collectively compose an image.

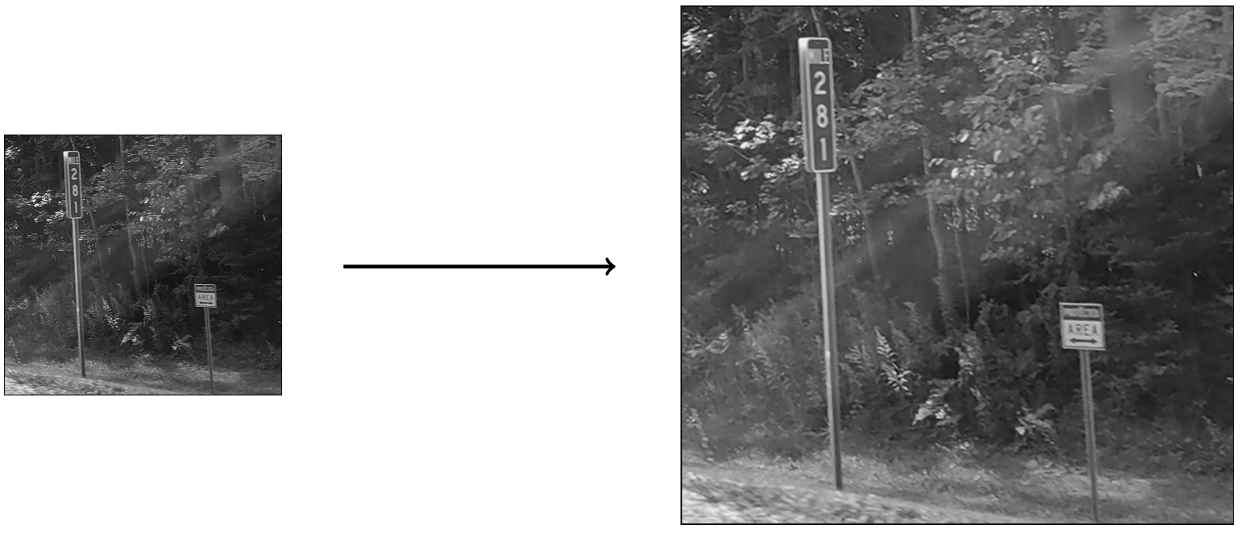
On the other hand, **vector graphics** represent images using a set of fixed points and mathematical equations that define how these points should be connected. The mathematical formulas that comprise a vector file capture most of the details of a vector image, from lines and curves to shapes and fills. Other visual features such as colors and line widths may also be tracked alongside these formulas. Because the details of a vector image can be generated from its underlying points and formulas, no physical pixels are needed to store a vector image. Vector graphics are often used to represent fonts, logos, and an assortment of other digital illustrations that involve geometric shapes and solid, non-blended colors.



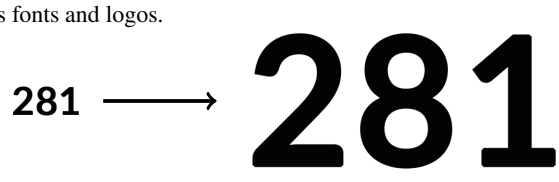
The shape/outline of the letter is generated using mathematical formulas instead of individual pixels.

Vector graphics are created using paths and shapes that are generated using mathematical formulas.

Raster and vector graphics are two distinct formats that can be used to represent an image, and they have several notable differences. One of the biggest differences between the two formats involves scalability. Because raster images are pixel based, they are not easily scalable, and enlarging a raster image may result in substantial quality loss. This is because the pixels in the original image end up being stretched out over a larger area, making the resultant image appear more pixelated and less sharp.



Vector graphics, on the contrary, do not deal with pixels and instead compute an image using mathematical formulas and other non-resolution dependent information. Thus, when you enlarge a vector image, its features are recomputed to adjust for its new size. This allows the resultant image to maintain the same sharpness prior to the size change. Because of this, vector graphics are often preferred for objects that need to be easily scaled without quality loss, such as fonts and logos.



Because vector graphics are represented using mathematical formulas, they are more lightweight compared to raster graphics, which may require a large number of pixels to represent the same image (and thus also more storage). However, raster graphics are more versatile in terms of the spectrum of images they can represent. For instance, blended color schemes and gradients are much easier to depict with raster graphics (which is why photographs are often stored in a raster format). Additionally, raster files are easier to view and edit compared to vector files, which may require specialized software to use.

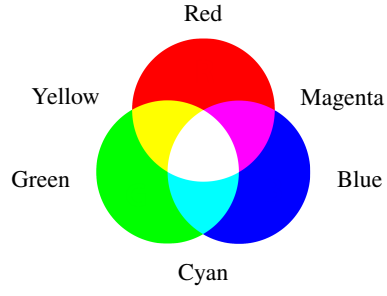
※ 26.11.2 *Color Representation in Images*

There are several ways to represent color in an image, but the most common is the **RGB color model**. In an RGB image, varying amounts of red (R), green (G), and blue (B) are combined to form all the colors in the image. You can essentially think of an RGB image as three versions of the same image — one filtered red, one filtered green, and one filtered blue — that are stacked on top of each other to produce a full color image.

> **Remark:** Why are red, green, and blue selected as the primary colors in the RGB color model? The answer lies in the physiology of how our eyes work. In the retina of our eyes, there are photoreceptor cells (known as cone cells) that allow us to perceive color. There are three primary types of cone cells, each sensitive to a different wavelength of light roughly corresponding to the colors of red, green, and blue. The combination of the light signals received from these cone cells allows our brains to differentiate among a wider variety of colors. Thus, the RGB color model essentially applies the process by which our eyes perceive color to represent color on a digital image!

When representing RGB raster images on a computer, each pixel is typically associated with three unsigned 8-bit integers (R, G, B) that indicate how much red, green, and blue should be added to produce its color (note that an unsigned 8-bit integer supports a range from 0 to 255, so a value of 255 represents the maximum amount of red, green, or blue). For instance, the RGB triplet of (255, 0, 0) represents red, (0, 255, 0) represents green, and (0, 0, 255) represents blue. In addition, the RGB triplet (0, 0, 0) represents black, and (255, 255, 255) represents white. A table of several important colors in the RGB space is provided below:

Color	***(R, G, B)***
Black	$(0,0,0)$
Red	$(255,0,0)$
Green	$(0,255,0)$
Blue	$(0,0,255)$
Yellow	$(255,255,0)$
Cyan	$(0,255,255)$
Magenta	$(255,0,255)$
White	$(255,255,255)$



It is common to see the three RGB values of a color represented in the hexadecimal (base-16) notation #RRGGBB, where RR represents the base-16 numeric value for red, GG represents the base-16 numeric value for green, and BB represents the base-16 numeric value for blue.[8] For example, the color crimson has the hexadecimal notation #DC143C. This corresponds to the RGB values (220, 20, 60), since "DC" is 220 in base-16, "14" is 20 in base-16, and "3C" is 60 in base-16.

RGB is not the only color model that exists. Many other color models can be used, each often serving a different purpose. However, RGB is considered as one of the go-to standards for color representation due to its relative simplicity and ubiquity compared to these other models.

※ 26.11.3 *Image Compression*

Image compression is the process of reducing the size of a digital image, typically for lighter storage or more efficient transmission. There are two primary classes of image compression: lossless and lossy compression. If **lossless compression** is applied to an image, the file size is reduced, but the original image can be perfectly reconstructed from the compressed file. Because of this, lossless compression is more limited with how much it can actually compress, as it must maintain enough information for a compressed image to be restored to its original quality. On the other hand, if **lossy compression** is applied, data may be permanently discarded to produce the compressed image. This allows lossy compression to compress more, but at the expense of losing some of the original image data. Examples of lossless and lossy images are pretty common: for instance, PNG image files utilize lossless compression methods, while JPEG image files utilize lossy compression methods.

Note that both compression categories have their own use cases, so lossy compression is not always inferior depending on what you want to accomplish. If you want a high compression rate and do not need to retain the all of the original image's data, a lossy compression method may be better. This is why JPEG is a popular method for representing digital photographs: for most high-resolution images, a fair amount of image data can be safely compressed away without a perceivable loss in image quality, and the enhanced compression rate of lossy files typically allows JPEG image files to be smaller than their PNG counterparts. On the contrary, images that place a greater importance on smaller details, such as digital art and images created through graphic design, are more often found in a lossless PNG format rather than a lossy JPEG one.

[8] Base-16 notation uses the letters A, B, C, D, E, and F to represent the numbers 10, 11, 12, 13, 14, and 15, respectively.