correctness. For example, we will introduce two search algorithms later in this chapter, and their correctness should be quite straightforward. Other algorithms, however, are not so obvious, and proving their correctness will require more formal analysis. A second feature of algorithms is their efficiency or running time. Of course we always want an algorithm to be fast, but there are theoretical limits on how fast or efficient an algorithm can be, as determined by the problem. We will discuss some of those problems at the end of the book under topics of spatial optimization. Besides correctness and running time, algorithms are often closely related to how the data are organized to enable the processes and how the algorithms are actually implemented.

## 1.1 Computational concerns for algorithms

Let us assume we have a list of $n$ points and the list does not have any order. We want to find a point from the list. How long will we take to find the point? This is a reasonable question. But the actual *time* is highly related to a lot of issues such as the programming language, the skill of the person who codes the program, the platform, the speed and number of the CPUs, and so on. A more useful way to examine the time issue is to know how many *steps* we need to finish the job, and then we analyze the total cost of performing the algorithm in terms of the number of steps used. The cost of each step is of course variable and is dependent on what constitutes a step. Nevertheless, it is still a more reliable way of thinking about computing time because many computational steps, such as simple arithmetic operations, logical expressions, accessing computer memory for information retrieval, and variable value assignment, can be identified and they only cost a constant amount of time. If we can figure out a way to count how many steps are needed to carry out a procedure, we will then have a pretty good idea about how much time the entire procedure will cost, especially when we compare algorithms.

Returning to our list of points, if there is no structure in the list – the points are stored in an arbitrary order – the best we can do to find a point from the list is to test all the points in the list, one by one, until we can conclude it is in or not in the list. Let us assume the name of the list is `points` and we want to find if the list includes point `p0`. We can use a simple algorithm to do the search (Listing 1.1).

Listing 1.1: Linear search to find point `p0` in a list.

```
1 for each point p in points:
2     if p is the same as p0:
3         return p and stop
```

The algorithm in Listing 1.1 is called a linear search; in it we simply go through all the points, if necessary, to search for the information we need. How many steps are necessary in this algorithm? The first line is a loop and, because of the size of the list, it will run as many as $n$ times when the item we are looking for happens to be the last one in the list. The cost of running just once in the loop part in line 1 is a constant because the list is stored in the computer memory, and the main operation steps here are to access the information at a fixed location in the memory and then to move on to the next item in the

memory. Suppose that the cost is $c_1$ and we will run it up to to $n$ times in the loop. The second line is a logic comparison between two points. It will run up to $n$ times as well because it is inside the loop. Suppose that the cost of doing a logic comparison is $c_2$ and it is a constant too. Line 3 simply returns the value of the point found; it has a constant cost of $c$ and it will only run once. For the best case scenario, we will find the target at the first iteration of the loop and therefore the total cost is simply $c_1 + c_2 + c$, which can be generalized as a constant $b + c$. In the worst case scenario, however, we will need to run all the way to the last item in the list and therefore the total cost becomes $c_1n + c_2n + c$, which can be generalized as $bn + c$, where $b$ and $c$ are constants, and $n$ is the size of the list (also the size of the problem). On average, if the list is a random set of points and we are going to search for a random point many times, we should expect a cost of $c_1n/2 + c_2n/2 + c$, which can be generalized as $b'n + c$, and we know $b' < b$, meaning that it will not cost as much as the worse case scenario does.

How much are we interested in the actual values of $b$, $b'$, and $c$ in the above analysis? How will these values impact the total computation cost? As it turns out, not much, because they are constants. But adding them up many times will have a real impact and $n$, the problem size, generally controls how many times these constant costs will be added together. When $n$ reaches a certain level, the impact of the constants will become minimal and it is really the magnitude of $n$ that controls the *growth* of the total computation cost.

Some algorithms will have a cost related to $n^2$, which is significantly different from the cost of $n$. For example, the algorithm in Listing 1.2 is a simple procedure to compute the shortest pairwise distance between two points in the list of $n$ points. Here, the first loop (line 2) will run $n$ times at a cost of $t_1$ each, and the second loop (line 3) will run exactly $n^2$ times at the same cost of $t_1$ each. The logic comparison (line 4) will run $n^2$ times and we assume each time the cost is $t_2$. The calculation of distance (line 5) will definitely be more costly than the other simple operations such as logic comparison, but it is still a constant as the input is fixed (with two points) and only a small finite set of steps will be taken to carry out the calculation. We say the cost of each distance calculation is a constant $t_3$. Since we do not compute the distance between the point and itself, the distance calculation will run $n^2–n$ times, as will the comparison in line 6 (with time $t_4$). The assignment in line 7 will cost a constant time of $t_5$ and may run up to $n^2–n$ times in the worst case scenario where every next distance is shorter than the previous one. The last line will only run once with a time of $c$. Overall, the total time for this algorithm will be $t_1n + t_1n^2 + t_2n^2 + t_3(n^2–n) + t_4(n^2–n) + t_5(n^2 – n) + c$, which can be generalized as $an^2 + bn + c$. Now it should be clear that this algorithm has a running time that is controlled by $n^2$.

Listing 1.2: Linear search to find shortest pairwise distance in a list of points.

```
1  let mindist be a very large number
2  for each point p1 in points:
3      for each point p2 in points:
4          if p1 is not p2:
5              let d be the distance between p1 and p2
6              if d < mindist:
7                  mindist = d
8  return mindist and stop
```

In the two example algorithms we have examined so far, the order of $n$ indicates the total cost and we say that our linear search algorithm has a computation cost in the order of $n$ and the shortest pairwise distance algorithm in the order of $n^2$. For the linear search, we also know that, when $n$ increases, the total cost of search will always have an upper bound of $bn$. But is there a lower bound? We know the best case scenario has a running time of a constant, or in the order of $n^0$, but that does not apply to the general case. When we can definitely find an upper bound but not a lower bound of the running time, we use the $O$-notation to denote the order. In our case, we have $O(n)$ for the average case, and the worst case scenario as well (because again the constants do not control the total cost). In other words, we say that the running time, or time complexity, of the linear search algorithm is $O(n)$. Because the $O$-notation is about the upper bound, which is meant to be the worst case scenario, we also mean the time complexity of the worst case scenario.

There are algorithms for which we do not have the upper bound of their running time. But we know their lower bound and we use the Ω-notation to indicate that. A running time of $\Omega(n)$ would mean we know the algorithm will at least cost an order of $n$ in its running time, though we do not know the upper bound of the running time. For other algorithms, we know both upper and lower bounds of the running time and we use the Θ-notation to indicate that. For example, a running time of $\Theta(n^2)$ indicates that the algorithm will take an order of $n^2$ in running time in all cases, best and worst. This is the case for our shortest distance algorithm because the process will always run $n^2$ times, regardless of the outcome of the comparison in line 6. It is more accurate to say the time complexity is $\Theta(n^2)$ instead of $O(n^2)$ because we know the lower bound of the running time of pairwise shortest distance is always in the order of $n^2$.

Now we reorganize our points in the previous list in a particular tree structure as illustrated in Figure 1.1. This is a binary tree because each node on the tree can have at most two branches, starting from the root. Here the root of the tree stores point (6, 7) and we show it at the top. All the points with $X$ coordinates smaller than or equal to that at the root are stored in the left branches of the root and those with $X$ coordinates greater than that of the root point are stored in the right branches of the root. Going down the tree to the second level, we have two points there, (4, 6) and (9, 4). For each of these points, we make sure that the rest of the points will be stored on the left if they have a smaller or equal $Y$ coordinate value, and on the right if greater. We alternate the use of $X$ and $Y$ coordinates going down the tree, until we find the point we are looking for or reach the end of a branch (a leaf node).

To use the tree structure to search for a point, we start from the root (we always start from the root for a tree structure) and go down the tree by determining which branch to proceed along using the appropriate coordinate at each level of the tree. For example, to

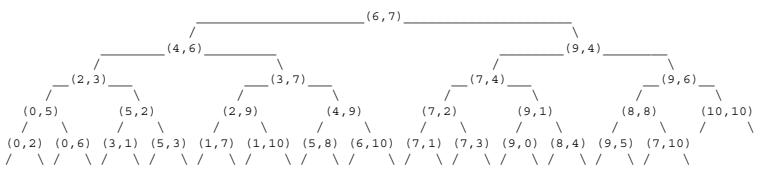


**Figure 1.1** A tree structure that stores 29 random points. Each node of the tree is labeled using a point with $X$ and $Y$ coordinates that range from 0 to 10

search for a target point of (1, 7), we first go to the left branch of the root because the $X$ coordinate of our target point is 1, smaller than that in the root. Then we go the the right branch of the second level node of (4, 6) because the $Y$ coordinate (7) is greater than that in the node. Now we reach the node of (3, 7) at the third level of the tree and we will go to the left branch there because the target $X$ coordinate is smaller than that in the node. Finally, we reach the node (2, 9) and we will move to its left branch because the $Y$ coordinate in the target is smaller than that in the node. In sum, given a tree, we can write the algorithm in Listing 1.3 to fulfill such a search strategy using a tree structure.

Listing 1.3: Binary search to find point `p0` in a tree.

```
1  let t be the root of the tree
2  while t is not empty:
3      let p be the point at node t
4      if p is the same as point p0:
5          return p and stop
6      if t is on an even level of the tree:
7          coordp, coordp0 = X coordinates of p and p0
8      else:
9          coordp, coordp0 = Y coordinates of p and p0
10     if coordp0 <= coordp:
11         t = the left branch of t
12     else:
13         t = the right branch of t
```

This is called a binary search, using the tree structure. Based on our discussion about running time, it is straightforward to see that the running time of this search algorithm is determined by the number of times we have to run the while loop (line 2), which is determined by the height of the tree, as defined by the number of edges from the root to the farthest leaf node. The above tree has a height of 4 and can hold up to 31 points (we only have 29 here). In general, for a binary tree with a height of $H$, we can store up to $2^0 + 2^1 + 2^2 + \ldots + 2^H = 2^{H+1} - 1$ items. In other words, if we have $n$ points in total that fill all the nodes in a perfectly balanced binary tree where all the leaf nodes are at exactly the same level, we have $2^{H+1} - 1 = n$ and hence $H = \log_2(n + 1) - 1$. In this case, when

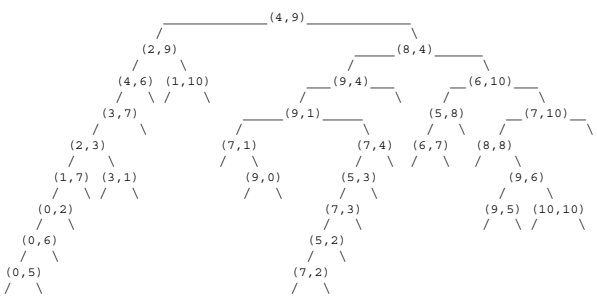


**Figure 1.2** An unbalanced tree structure to store 29 random points

such a tree is given, we have a running time of the order of $\log_2(n+1)$, which is at the same order as $\log_2 n$, and we say the running time is $O(\log_2 n)$. For a balanced but not perfect tree, meaning the difference between the heights of all leaf nodes is at most 1, we can still achieve a running time of $O(\log_2 n)$ since the farthest leaf node has a height of $H$. When a balanced tree cannot be guaranteed, however, things can get worse. An example of an unbalanced tree is given in Figure 1.2 where we have exactly the same points but the tree has a height of 8. Therefore, we know the binary search algorithm is more efficient than linear search because $O(\log_2 n) < O(n)$, but the actual running time is dependent on how the tree is constructed and can be longer than $O(\log_2 n)$ if the tree is unbalanced.

Since using a balanced binary tree to store the points can greatly improve the efficiency of search, will that also help to reduce the running time of calculating the shortest pairwise distance? The brute-force approach we discussed here has a running time of $\Theta(n^2)$, which grows quickly as $n$ increases. It would be reasonable to look for a more efficient way to get the shortest pairwise distance in a list of points. The answer to our question is yes, and the key lies in the use of tree structures. We will continue to explore this topic in the second part of the book where we discuss different tree structures in more depth. Throughout the book, we do not focus much on the theoretical analysis of running time for each algorithm. Instead, we will conduct more empirical analysis by actually running the algorithms on different data sets.

The discussion about search and especially binary search on a tree logically leads to the topic of data structure: how we store and organize data to facilitate the procedures in an algorithm. A tree structure is a good example of how the original data stored in a list can be reorganized to achieve better search performance. Many data structures are problem-specific. Some data structures can be complicated, but the increase in storage is often compensated by a decrease in running time.

## 1.2 Coding

Algorithms can be described in different ways. We use verbal statements in this chapter to describe the linear and binary search algorithms. For theoretical work, a formal description that details the steps but is not necessarily executable will suffice. We call this type of description pseudo-code because it is not real computer code, though very close. In this book, we take a more practical and explicit route by describing algorithms in an actual computer programming language. Specifically, we use Python to describe the algorithms covered in this book.

Writing computer programs (i.e., coding) to describe algorithms has a substantial benefit: all the algorithms will immediately be executable. In this way, we present everything related to how the algorithms work in the plain text of the book. The code becomes part of the text and consequently becomes an open source experiment where each line of the process can be examined, modified, and improved. However, this code as text approach may present too much information, especially when the programming language may need ancillary code to help the main task. For example, many programming languages require end of line symbols and brackets as part of the code to ensure syntax correctness. These symbols, when added as part of the text, may

hamper the reading process and therefore make it difficult for us to concentrate on the main contents of the text. We choose Python in this book largely because of its simple syntax, along with many of its popular, powerful, and well maintained modules. All the programs listed in this book were tested in Python 2.7, which is a stable and widely adopted version at the time of writing. The majority of the programs in this book only use the basic Python features, so these programs in principle are likely to be compatible with newer versions of Python.

Python has become a popular programming language in recent years, including the use of Python for plugins in GIS packages such as QGIS and ArcGIS. It is important to point out that programming in Python is a skill that can definitely be acquired through learning and practice. To help readers make a quick start on the language, a short introduction to Python is included as Appendix A. This is not a comprehensive tutorial as many of the online tutorials have more detailed and in-depth discussion about the language. However, many of Python's useful features, especially those related to the main text, are included in the tutorial.

## 1.3 How to use this book

The main text of this book is divided into three major parts.The general logic here is to start the book with a discussion on the most fundamental aspects of the data – the geometry – before we move on to more advanced topics in spatial indexing and spatial analysis and modeling. At the end of each chapter we review the major literature related to the topics covered in a section called Notes. At the end of the book, we also include three appendices to help readers understand the Python programming language and the structure of the programs included in the book.

In Part I, we focus on locations, or more specifically on coordinates that can be used to help us understand geospatial information. In Chapter 2, we examine a few algorithms to compute different kinds of distance, such as distances between points and distance from a point to a line. We also look at the calculation of polygon centroids and a widely used algorithm called point-in-polygon that efficiently helps us determine whether a point is located within a polygon. The final topic in Chapter 2 is about the transformation between coordinate systems involving map projections. Chapter 3 covers a traditional GIS operation, known as overlay. As "old" as this topic is, the actual computation of overlaying two polygons can be tedious, though not necessarily complicated. Many of the topics in this part of the book are related to the field of computational geometry. But we focus on those that are most relevant to the GIS world.

Part II is centered around the idea of spatial indexing. Spatial information is special. Though the general concept in indexing, divide and conquer, is the same for spatial information, because of the two dimensions (or more in some cases) in spatial information, more dedicated algorithms must be designed. We first introduce the basic concepts of indexing in Chapter 4, where we focus on the development of a tree structure. Chapter 5 is devoted to $k$-D trees that are commonly used to index point data. Chapter 6 covers a popular indexing technique called quadtrees, for both point and raster data. Chapter 7 extends the discussion to indexing lines and polygons in spatial data.

Part III of the book focuses on the heart of GIS applications: spatial analysis and modeling. We first explore the interpolation methods on point data in Chapter 8 where we compare and contrast two commonly used interpolation methods: inverse distance weighting and kriging. We also include a data simulation algorithm called midpoint displacement from the fractal geometry literature. Chapter 9 is devoted to spatial pattern analysis where the calculation of indices such as Moran's $I$ is included. Algorithms for network analysis, especially those calculating the shortest paths, are included in Chapter 10. We devote two chapters to topics in spatial optimization: in Chapter 11 we focus on the exact methods and in Chapter 12 we explore some of the heuristic methods.

In addition to the three parts in the main text, we have also included three appendices to cover some of the technical details about coding. It should be obvious that, while we talk about algorithms most of the time, this book is about coding as well. For this reason, a short introduction to Python is first included. Then we compile a short introduction on the Python binding of a powerful library called GDAL/OGR and a Python library for spatial analysis called PySAL. The purpose here is to help readers quickly get started with these libraries so that they can have a sense how "real-world" data sets can be closely related to the topics (and code of course) presented in this book.

Most of the programs listed in this book are also given a file name that can be used in other programs. In that case the name of the program is listed in the caption of each code listing. We use directories to organize the programs. In the last appendix, we provide an overview of the code by listing all the Python programs and example data sets and discuss how they can be used.

Each of the chapters in this book could easily be extended into another book to cover the depth of each topic. This book, then, is a survey of the general topics in GIS algorithms. The best way to grasp the breadth of the topics presented in this book is coding. A collective Github page[2] is under development where readers can contribute their thoughts on implementing the algorithms included in this book and new algorithms that are beyond the scope of the book. While theoretical derivation is not the focus of this book, empirical analysis definitely is. We have included many experiments in the text and have suggested more in the exercises. This, however, should not limit further experiments, especially those that are innovative and overarch various topics. The book will only be successful if it achieves two goals: first, based on the skills built on understanding the algorithms and code, that readers are able to develop their own tool sets that fit different data sets and application requirements; and second, that coding becomes a habit when it comes to dealing with geospatial data.

[2] https://github.com/gisalgs

# Part I

# Geometric Algorithms

# 2

# Basic Geometric Operations

> Imagine a vast sheet of paper on which straight Lines, Triangles, Squares, Pentagons, Hexagons, and other figures, instead of remaining fixed in their places, move freely about, on or in the surface, but without the power of rising above or sinking below it, very much like shadows – only hard with luminous edges – and you will then have a pretty correct notion of my country and countrymen.
>
> Edwin A. Abbott, *Flatland: A Romance of Many Dimensions*

This chapter focuses on the essential algorithms in GIS that involve geometric operations. We first introduce the calculation of distance between two points and then move on to geometric objects with higher dimensions, including algorithms to determine properties such as point-in-polygon, point distance to lines, polygon centroid, and polygon area. We will also examine two map projections and discuss how to transform geospatial data from one coordinate system to another.

## 2.1 Point

Before we start our discussion, let us define a data structure for a point that will be used throughout this book. This is a Python class called `Point` (Listing 2.1). We are only interested in two-dimensional cases and we just store the $X$ and $Y$ coordinates of a point in the class. We override a few Python built-in methods to provide some convenient features. The `__getitem__` method allows us to iterate through the $X$ and $Y$ coordinates of a point using indices 0 and 1, respectively. The `__len__` method returns the number of dimensions in the point (we only return two here). We should also be able to judge whether two points are identical if they have exactly the same $X$ and $Y$ coordinates (`__eq__`), or different if the coordinates are different (`__ne__`). Additionally, we also need a way to compare points in a coordinate system so that a point on the lower left-hand side of the coordinate system is always "smaller" than those on the upper right-hand side. For two points $p_1$ and $p_2$, we say that $p_1 < p_2$ if $p_1$ has a smaller $X$ coordinate. For two points with the same $X$ coordinate, the lower point with a smaller $Y$ coordinate is considered as smaller. This is captured by overriding the built-in comparison operators of `__lt__` (less than), `__gt__` (greater than), `__le__` (less than or equal to), and `__ge__` (greater than or equal to). We also use the coordinates of the point when it is printed out in text (`__str__` and `__repr__`). While we

do not specifically use this type of ordering and comparison in this chapter, they play a big role in later chapters. A `distance` method is included to compute the Euclidean distance between two points.

Listing 2.1: Data structure for a point class (point.py).

```
from math import sqrt
class Point():
    """A class for points in Cartesian coordinate systems."""
    def __init__(self, x=None, y=None):
        self.x, self.y = x, y
    def __getitem__(self, i):
        if i==0: return self.x
        if i==1: return self.y
        return None
    def __len__(self):
        return 2
    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x==other.x and self.y==other.y
        return NotImplemented
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    def __lt__(self, other):
        if isinstance(other, Point):
            if self.x<other.x:
                return True
            elif self.x==other.x and self.y<other.y:
                return True
            return False
        return NotImplemented
    def __gt__(self, other):
        if isinstance(other, Point):
            if self.x>other.x:
                return True
            elif self.x==other.x and self.y>other.y:
                return True
            return False
        return NotImplemented
    def __ge__(self, other):
        if isinstance(other, Point):
            if self > other or self == other:
                return True
            else:
                return False
            return False
```

```
44          return NotImplemented
45      def __le__(self, other):
46          if isinstance(other, Point):
47              if self < other or self == other:
48                  return True
49              else:
50                  return False
51              return False
52          return NotImplemented
53      def __str__(self):
54          if type(self.x) is int and type(self.y) is int:
55              return "({0},{1})".format(self.x,self.y)
56          else:
57              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
58      def __repr__(self):
59          if type(self.x) is int and type(self.y) is int:
60              return "({0},{1})".format(self.x,self.y)
61          else:
62              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
63      def distance(self, other):
64          return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
```

The `Point` class can be easily extended to become a more flexible representation of points in geospatial data. For example, while we assume *X* and *Y* are on a Cartesian plane (because of the way the `distance` method is defined), we can relax this constraint by adding a new data member of the class called `CS` that can be used to specify the kind of coordinate system for the point and we can calculate the distance accordingly. We can create a subclass that inherits from the `Point` class so that points in higher dimensions can be represented. We can also add time and other attributes to the class.

## 2.2 Distance between two points

The distance between two points can be calculated in various ways, largely depending on the application domain and the coordinate system in which the two points are measured. The most commonly used distance measure is the Euclidean distance on the straight line between the two points on a Cartesian plane where the distance can be simply computed as

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

where $x_1$ and $x_2$ are the horizontal ($X$) coordinates of the two points and $y_1$ and $y_2$ are their vertical ($Y$) coordinates. The `distance` method in the `Point` class returns the Euclidean distance between two points (Listing 2.1).

In some special cases, the Euclidean distance may not be a suitable measure between two points, especially in an urban setting where one must follow the street network to go

from one point to another (Figure 2.1). In this case, we can use the Manhattan distance between these two points:

$$d_1 = |x_1 - x_2| + |y_1 - y_2|.$$

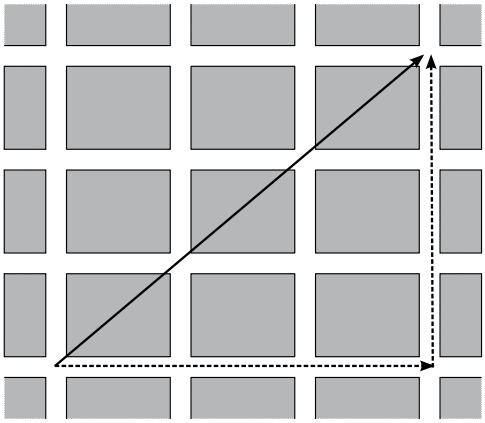

**Figure 2.1** Euclidean (solid line) and Manhattan distance (dashed lines). Shaded areas represent buildings in an urban setting where roads run between buildings

When the coordinates of the points are measured on a spherical surface, the above distance measures will not give the correct distance between the points. In this case, the shortest distance between the two points is measured on the great circle that passes through the two points. More specifically, when each point is measured by its latitude or the angle to the equatorial plane ($\varphi$) and longitude or the angle between the meridian of the point and the prime meridian ($\lambda$), the arc distance $\alpha$ (angle) between two points on the great circle is given by

$$\cos\alpha = \sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos d\lambda,$$

where $d\lambda = |\lambda_1 - \lambda_2|$ is the absolute difference between the longitudes of the two points. In the actual calculation, however, we use the following Haversine formula to avoid the difficulty of handling negative values:

$$a = \sin^2\frac{d\varphi}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2\frac{d\lambda}{2},$$

$$c = 2\arcsin\min(1, \sqrt{a}),$$

where the min function returns the smaller value of the two inputs in order to avoid numerical artifacts that may cause the value of $a$ to be greater than 1; then the distance can be calculated as

$$d = cR,$$

where $R$ is the radius of the spherical earth (3,959 miles or 6,371 kilometers). It is important to note that the latitude and longitude readings must be converted to radians before the above formula can be used to calculate the distance.

We now write a Python program to calculate the spherical distance between two points (Listing 2.2). Here the latitudes and longitudes are provided in degrees and we convert them to radians (lines 13–16). Using this code, we can calculate the distance between Columbus, OH (40° N, 83° W) and Beijing (39.91° N, 116.56° E) as 6,780 miles (10,911 kilometers).

Listing 2.2: Calculating the great circle distance (spherical_distance.py).

```
1  import math
2  def spdist(lat1, lon1, lat2, lon2):
3      """
4      Calculates the great circle distance given
5      the latitudes and longitudes of two points.
6      Input
7        lat1, lon1: lat and long in degrees for the first point
8        lat2, lon2: lat and long in degrees for the second point
9      Output
10       d: great circle distance
11     """
12     D = 3959                          # earth radius in miles
13     phi1 = math.radians(lat1)
14     lambda1 = math.radians(lon1)
15     phi2 = math.radians(lat2)
16     lambda2 = math.radians(lon2)
17     dlambda = lambda2 - lambda1
18     dphi = phi2 - phi1
19     sinlat = math.sin(dphi/2.0)
20     sinlong = math.sin(dlambda/2.0)
21     alpha=(sinlat*sinlat) + math.cos(phi1) * \
22         math.cos(phi2) * (sinlong*sinlong)
23     c=2 * math.asin(min(1, math.sqrt(alpha)))
24     d=D*c
25     return d
26
27 if __name__ == "__main__":
28     lat1, lon1 = 40, -83              # Columbus, OH
29     lat2, lon2 = 39.91, 116.56        # Beijing
30     print spdist(lat1, lon1, lat2, lon2)
```

## 2.3 Distance from a point to a line

Let $ax + by + c = 0$ be a line on the plane, where $a$, $b$, and $c$ are constants. The distance from a point $(x_0, y_0)$ on the plane to the line is computed as

$$\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

We can prove this by calculating the distance from the point to the intersection point between line $ax + by + c = 0$ and its perpendicular line that goes through point $(x_0, y_0)$. Let $(x_1, y_1)$ be the intersection point, and we know the slope of the perpendicular line is $b / a$. Hence, we have

$$\frac{y_1 - y_0}{x_1 - x_0} = \frac{b}{a},$$

which gives

$$a(y_1 - y_0) - b(x_1 - x_0) = 0.$$

We then take the square of both sides and rearrange the result to give

$$a^2(y_1 - y_0)^2 + b^2(x_1 - x_0)^2 = 2ab(x_1 - x_0)(y_1 - y_0).$$

We add $a^2(x_1 - x_0)^2 + b^2(y_1 - y_0)^2$ to both sides. The left-hand side can be rewritten as

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2],$$

and the right-hand side as

$$[a(x_1 - x_0) + b(y_1 - y_0)]^2 = [ax_1 + by_1 - ax_0 - by_0]^2.$$

Because point $(x_1, y_1)$ is also on the original line, we have $ax_1 + by_1 = -c$. Hence

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2] = [ax_0 + by_0 + c]^2.$$

The term $(y_1 - y_0)^2 + (x_1 - x_0)^2$ is the square of the distance between $(x_0, y_0)$ and line $ax + by + c$. We then have the distance as

$$\sqrt{(y_1 - y_0)^2 + (x_1 - x_0)^2} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

In most GIS applications, we can easily get information about a line segment defined by the two endpoints. It is therefore necessary to compute the values of $a$, $b$, and $c$ in the above equations using the two points. Let $(x_1, y_1)$ and $(x_2, y_2)$ be the two endpoints. We define $dx = x_1 - x_2$ and $dy = y_1 - y_2$. We can then write the line equation using the slope:

$$y = \frac{dy}{dx}x + n,$$

where $n$ is a constant that we will compute. We now plug in the first endpoint to get

$$y_1 = \frac{dy}{dx} x_1 + n,$$

which gives

$$n = y_1 - \frac{dy}{dx} x_1.$$

Therefore, we have a general form for the line as

$$y = \frac{dy}{dx} x + y_1 - \frac{dy}{dx} x_1.$$

We multiply both sides of the above equation by $dx$ and rearrange it as

$$xdy - ydx + y_1 dx - x_1 dy = 0.$$

Now we have $a = dy$, $b = -dx$, and $c = y_1 dx - x_1 dy$. With these parameters, we can quickly calculate the distance between a point and the line segment (Listing 2.3).

Listing 2.3: A Python program to calculate point to line distance (point2line.py).

```
1  import math
2  from point import *
3
4  def point2line(p, p1, p2):
5      """
6      Calculate the distance from point to a line.
7      Input
8        p: the point
9        p1 and p2: the two points that define a line
10     Output
11       d: distance from p to line p1p2
12     """
13     x0 = float(p.x)
14     y0 = float(p.y)
15     x1 = float(p1.x)
16     y1 = float(p1.y)
17     x2 = float(p2.x)
18     y2 = float(p2.y)
19     dx = x1-x2
20     dy = y1-y2
21     a = dy
22     b = -dx
23     c = y1*dx - x1*dy
24     if a==0 and b==0:          # p1 and p2 are the same point
25         d = math.sqrt((x1-x0)*(x1-x0) + (y1-y0)*(y1-y0))
```

```
26      else:
27          d = abs(a*x0+b*y0+c)/math.sqrt(a*a+b*b)
28      return d
29
30  if __name__ == "__main__":
31      p, p1, p2 = Point(10,0), Point(0,100), Point(0,1)
32      print point2line(p, p1, p2)
33      p, p1, p2 = Point(0,10), Point(1000,0.001), Point(-100,0)
34      print point2line(p, p1, p2)
35      p, p1, p2 = Point(0,0), Point(0,10), Point(10,0)
36      print point2line(p, p1, p2)
37      p, p1, p2 = Point(0,0), Point(10,10), Point(10,10)
38      print point2line(p, p1, p2)
```

We use a few test cases (lines 31–38) to demonstrate the use of the code. The output is as follows:

```
10.0
9.9999090909
7.07106781187
14.1421356237
```

## 2.4 Polygon centroid and area

We represent a polygon $P$ of $n$ points as $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n), (x_{n+1}, y_{n+1})$, where we add an additional point in the sequence $(x_{n+1}, y_{n+1}) = (x_1, y_1)$ to ensure that the polygon is closed. If the polygon does not have holes and its boundaries do not intersect, the centroid of the polygon is determined by the coordinates

$$x = \frac{1}{6A}\sum_{i=1}^{n}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$y = \frac{1}{6A}\sum_{i=1}^{n}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

where $A$ is the area of the polygon. If the polygon is convex, the centroid is bound to be inside the polygon. However, for concave polygons, the centroid computed as above may be outside the polygon. Though the concept of centroid is related to the center of gravity of the polygon, an outside centroid does not meet the requirement of being the center, which makes it necessary to snap the centroid to a point on the polygon.

To understand how the area formula works, let us use the points of the polygon in Figure 2.2, ($a$, $b$, $c$, $d$, $e$, $f$, $g$, $a$), as an example, where $a$ appears twice to ensure polygon closure. Each line segment of the polygon can be used to form a trapezoid,

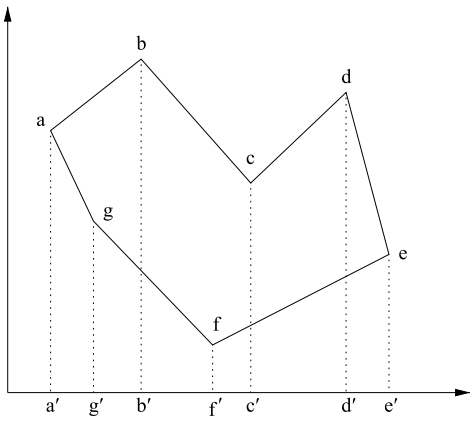


**Figure 2.2** Using trapezoids to calculate polygon area. The dotted lines project the nodes of the polygon onto the horizontal axis

and the area of each trapezoid can be calculated as $(x_2 - x_1)(y_2 + y_1)/2$, where subscripts 1 and 2 are used to denote two consecutive points in the sequence. For the case of trapezoid $abb'a'$, the area is calculated as $(x_b - x_a)(y_b + y_a)/2$. For trapezoid $fgg'f'$, the area is $(x_g - x_f)(y_g + y_f)/2$, which will be negative. Clearly, using this formula to calculate the areas of trapezoids underneath line segments *ab*, *bc*, *cd*, and *de* (note the order of points) will return positive values, while negative values will be obtained for the areas of trapezoids underneath lines *ef*, *fg*, and *ga*. In this way, the formula will finally return the correct area of the polygon by subtracting all the areas enclosed by $efgaa'e'$ from the area enclosed by $abcdee'a'$. The area of the polygon therefore can be computed as

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1} - x_i)(y_{i+1} + y_i).$$

The above formula is based on the decomposition of a polygon into a series of trapezoids. We can also rewrite the formula into a cross-product form:

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1}y_i - x_i y_{i+1}).$$

While this cross-product form works exactly the same as the above, in the past it has been more convenient to use for manual calculation. Depending on the order of the points in the sequence, the polygon area formula may return a negative value. Therefore an absolute operation may be needed to obtain the actual area of a polygon.

The Python program in Listing 2.4 returns both the centroid and the area using the above formulas. We also test the program using sample data where the polygon is simply represented using a list of points.

Listing 2.4: A Python program for calculating the area and centroid of a polygon (centroid.py).

```
from point import *

def centroid(pgon):
    """
    Calculates the centroid and area of a polygon.
    Input
      pgon: a list of Point objects
    Output
      A: the area of the polygon
      C: the centroid of the polygon
    """
    numvert = len(pgon)
    A = 0
    xmean = 0
    ymean = 0
    for i in range(numvert-1):
      ai = pgon[i].x*pgon[i+1].y - pgon[i+1].x*pgon[i].y
      A += ai
      xmean += (pgon[i+1].x+pgon[i].x) * ai
      ymean += (pgon[i+1].y+pgon[i].y) * ai
    A = A/2.0
    C = Point(xmean / (6*A), ymean / (6*A))
    return A, C

# TEST
if __name__ == "__main__":
    points = [ [0,10], [5,0], [10,10], [15,0], [20,10],
               [25,0], [30,20], [40,20], [45,0], [50,50],
               [40,40], [30,50], [25,20], [20,50], [15,10],
               [10,50], [8, 8], [4,50], [0,10] ]
    polygon = [ Point(p[0], p[1]) for p in points ]
    print centroid(polygon)
```

## 2.5 Determining the position of a point with respect to a line

The calculation of the area of a polygon can return a negative or positive value, depending on the order of the points. This is a very interesting feature and we can use the sign to determine whether a point is on a specific side of a line by considering the area of the triangle of the point and two points that form the line. Let points $a$, $b$, and $c$ be the three points on a triangle. Using the cross-product formula, we can obtain the area of this triangle as

$$A(abc) = \frac{1}{2}(x_b y_a - x_a y_b + x_c y_b - x_b y_c + x_a y_c - x_c y_a).$$

Adding $x_b y_b - x_b y_b$ into the parentheses, we can rewrite the above formula in a simpler form:

$$\begin{aligned} \text{sideplr}(abc) &= 2A(abc) \\ &= (x_a - x_b)(y_c - y_b) - (x_c - x_b)(y_a - y_b). \end{aligned}$$

If point *a* is on the left of the vector formed from point *b* to *c*, the above calculation will yield a negative value. For example, in the configurations of points *a*, *b*, and *c* in Figure 2.3A–C, the area of triangle *abc* (note the order of the points) is calculated in such a way that we always use the area of the trapezoids underneath the triangle (e.g., the trapezoid underneath line *cb* in 2.3A) to subtract from the total area (e.g., the sum of areas of trapezoids underneath lines *ba* and *ac* in Figure 2.3A). This results in a negative value. On the other hand, if *a* is on the right side of the the vector *bc* (see Figure 2.3D–F), we will have a positive value. The code for calculating the side is simple, as shown in Listing 2.5.

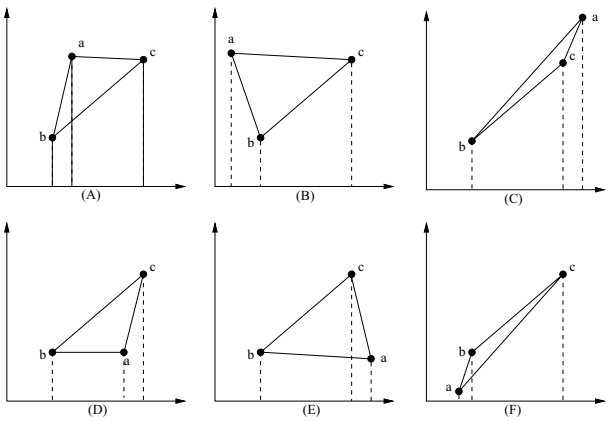


**Figure 2.3** The position of point *a* in relation to line *bc*

Listing 2.5: Determining the side of a point (sideplr.py).

```
1  from point import *
2
3  def sideplr(p, p1, p2):
4      """
5      Calculates the side of point p to the vector p1p2.
6      Input
7        p: the point
8        p1, p2: the start and end points of the line
9      Output
10       -1: p is on the left side of p1p2
11       0: p is on the line of p1p2
```

```
        1: p is on the right side of p1p2
    """
    return int((p.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p.y-p1.y))

if __name__ == "__main__":
    p=Point(1,1)
    p1=Point(0,0)
    p2=Point(1,0)
    print "Point %s to line %s->%s: %d"%(
        p, p1, p2, sideplr(p, p1, p2))
    print "Point %s to line %s->%s: %d"%(
        p, p2, p1, sideplr(p, p2, p1))
    p = Point(0.5, 0)
    print "Point %s to line %s->%s: %d"%(
        p, p1, p2, sideplr(p, p1, p2))
    print "Point %s to line %s->%s: %d"%(
        p, p2, p1, sideplr(p, p2, p1))
```

Running the test cases in the above code should return the following:

```
Point (1,1) to line (0,0)->(1,0): -1
Point (1,1) to line (1,0)->(0,0): 1
Point (0.5, 0.0) to line (0,0)->(1,0): 0
Point (0.5, 0.0) to line (1,0)->(0,0): 0
```

A collinear alignment of the three points will return a zero. By extending this, we can easily come up with a way to test whether a point is on a line, or lies on one side of the line. We will see how this technique is useful in the next chapter when we discuss the test of intersection between line segments.

## 2.6 Intersection of two line segments

Let us first consider the intersection between two lines: $L_1$ passing through points $(x_1, y_1)$ and $(x_2, y_2)$ and $L_2$ through $(x_3, y_3)$ and $(x_4, y_4)$, with slopes

$$\alpha_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{and} \quad \alpha_2 = \frac{y_4 - y_3}{x_4 - x_3},$$

respectively. Calculating their intersection is straightforward. We first write the equations of the lines for these two segments as

$$y = \alpha_1 (x - x_1) + y_1,$$

and

$$y = \alpha_2 (x - x_3) + y_3,$$

which can then be used to compute the $X$ coordinate of the intersection point as

$$x = \frac{\alpha_1 x_1 - \alpha_2 x_3 + y_3 - y_1}{\alpha_1 - \alpha_2},$$

and the $Y$ coordinate as

$$y = \alpha_1 (x - x_1) + y_1.$$

It is obvious that we will need to consider a few special cases. If the two lines have exactly the same slope, then there will be no intersection. But what if either or both of the lines are vertical with infinite slope? In this case, if both $x_2 - x_1$ and $x_4 - x_3$ are zero then we have two parallel lines. Otherwise, if only one of them is zero, the intersection point has the $x$ coordinate of the vertical line. For example, if $x_1 = x_2$, we have the intersection at point $x = x_1$ and $y = \alpha_2 (x_1 - x_3) + y_3$.

When only line segments are considered, however, using the above approach may not be necessary because the two segments may not intersect. We can do a quick check about whether it is possible for the two segments to intersect by looking at their endpoints. If both endpoints of one line segment are on the same side of the other segment, there will be no intersection. We can use the `sideplr` algorithm discussed in the previous section to test the side of a point with respect to a line segment.

Before we formally give the algorithm to compute the intersection between two line segments, let us define a data structure that can be used to effectively store the information about a line segment (Listing 2.6). When we store a segment, we require an edge number (`e`) and the endpoints. We keep a record of the original left point of the line (`lp0` in line 23) because in our later discussion the left point of a line will change due to the calculation of multiple intersection points that move from left to right. Finally, we use the `status` attribute to indicate if the left endpoint of the line segment is the original endpoint (default) or an interior point because of intersection with other segments as a result of polygon overlay, and use attribute `c` to store attributes associated with the line segment. We will use more of these features in the next chapter.

All the endpoints in `Segment` are based on the `Point` class that we have previously developed in Listing 2.1. We also define a suite of logic relations between two segments by overriding built-in Python functions such as `__eq__` (for equality) and `__lt__` (less than). Segment $s_1$ is said to be smaller than segment $s_2$ if $s_1$ is completely below $s_2$. We also include the function `contains` to test if a point is one of the endpoints of a segment.

Listing 2.6: Data structure for line segments (linesegment.py).

```
1 from point import *
2 from sideplr import *
3
4 ## Two statuses of the left endpoint
5 ENDPOINT = 0 ## original left endpoint
6 INTERIOR = 1 ## interior in the segment
7
```

```
8  class Segment:
9     """
10    A class for line segments.
11    """
12    def __init__(self, e, p0, p1, c=None):
13        """
14        Constructor of Segment class.
15        Input
16          e: segment ID, an integer
17          p0, p1: endpoints of segment, Point objects
18        """
19        if p0>=p1:
20           p0,p1 = p1,p0            # p0 is always left
21        self.edge = e               # ID, in all edges
22        self.lp = p0                # left point
23        self.lp0 = p0               # original left point
24        self.rp = p1                # right point
25        self.status = ENDPOINT      # status of segment
26        self.c = c                  # c: feature ID
27    def __eq__(self, other):
28        if isinstance(other, Segment):
29            return (self.lp==other.lp and self.rp==other.rp)\
30                or (self.lp==other.rp and self.rp==other.lp)
31        return NotImplemented
32    def __ne__(self, other):
33        result = self.__eq__(other)
34        if result is NotImplemented:
35            return result
36        return not result
37    def __lt__(self, other):
38        if isinstance(other, Segment):
39            if self.lp and other.lp:
40                lr = sideplr(self.lp, other.lp, other.rp)
41                if lr == 0:
42                    lrr = sideplr(self.rp, other.lp, other.rp)
43                    if other.lp.x < other.rp.x:
44                        return lrr > 0
45                    else:
46                        return lrr < 0
47                else:
48                    if other.lp.x > other.rp.x:
49                        return lr < 0
50                    else:
51                        return lr > 0
52        return NotImplemented
53    def __gt__(self, other):
54        result = self.__lt__(other)
55        if result is NotImplemented:
56            return result
57        return not result
58    def __repr__(self):
59        return "{0}".format(self.edge)
```

```
60      def contains(self, p):
61          """
62          Returns True if segment has p as an endpoint
63          """
64          if self.lp == p:
65              return -1
66          elif self.rp == p:
67              return 1
68          else:
69              return 0
```

The code in Listing 2.7 shows an example of how to calculate the intersection point between two line segments. We first use function `test_intersect` to determine if two given line segments will intersect (line 60), which can be done by testing the sides of the segments using the `sideplr` function (e.g., line 41). If both endpoints of a segment are on the same side of the other segment, no intersection will occur. Otherwise, we use the equations introduced at the beginning of this section in function `getIntersectionPoint` to compute the actual intersection point. This function assumes the two input segments indeed intersect (hence it is necessary to test intersection first). The two segments in the test data (lines 57 and 58) have an intersection point at (1.5, 2.5).

Listing 2.7: Calculating the intersection between two line segments (intersection.py).

```
1   from linesegment import *
2   from sideplr import *
3
4   def getIntersectionPoint(s1, s2):
5       """
6       Calculates the intersection point of two line segments
7       s1 and s2. This function assumes s1 and s2 intersect.
8       Intersection must be tested before calling this function.
9       """
10      x1 = float(s1.lp0.x)
11      y1 = float(s1.lp0.y)
12      x2 = float(s1.rp.x)
13      y2 = float(s1.rp.y)
14      x3 = float(s2.lp0.x)
15      y3 = float(s2.lp0.y)
16      x4 = float(s2.rp.x)
17      y4 = float(s2.rp.y)
18      if s1.lp < s2.lp:
19          x1,x2,y1,y2,x3,x4,y3,y4=x3,x4,y3,y4,x1,x2,y1,y2
20      if x1 != x2:
21          alpha1 = (y2-y1)/(x2-x1)
22      if x3 != x4:
23          alpha2 = (y4-y3)/(x4-x3)
24      if x1 == x2: # s1 is vertical
25          y = alpha2*(x1-x3)+y3
26          return Point([x1, y])
```

```
    if x3==x4: # s2 is vertical
        y = alpha1*(x3-x1)+y1
        return Point([x3, y])
    if alpha1 == alpha2: # parallel lines
        return None
    # need to calculate
    x = (alpha1*x1-alpha2*x3+y3-y1)/(alpha1-alpha2)
    y = alpha1*(x-x1) + y1
    return Point(x, y)

def test_intersect(s1, s2):
    if s1==None or s2==None:
        return False
    # testing: s2 endpoints on the same side of s1
    lsign = sideplr(s2.lp0, s1.lp0, s1.rp)
    rsign = sideplr(s2.rp, s1.lp0, s1.rp)
    if lsign*rsign > 0:
        return False
    # testing: s1 endpoints on the same side of s2
    lsign = sideplr(s1.lp0, s2.lp0, s2.rp)
    rsign = sideplr(s1.rp, s2.lp0, s2.rp)
    if lsign*rsign > 0:
        return False
    return True

if __name__ == "__main__":
    p1 = Point(1, 2)
    p2 = Point(3, 4)
    p3 = Point(2, 1)
    p4 = Point(1, 4)
    s1 = Segment(0, p1, p2)
    s2 = Segment(1, p3, p4)
    s3 = Segment(2, p1, p2)
    if test_intersect(s1, s2):
        print getIntersectionPoint(s1, s2)
        print s1==s2
        print s1==s3
```

## 2.7 Point-in-polygon operation

Identifying whether a point is inside a polygon is one of the most important operations in everyday GIS uses. For example, when we click on a point on a digital world map, we would expect to quickly retrieve the information related to the region that contains the point. For a simple (not self-intersecting) convex polygon, we can determine that a point is inside a polygon if it is on the same side of all the edges of that polygon. This approach seems to be computationally intensive because of the many multiplications, and it does not work for concave polygons (Figure 2.4).

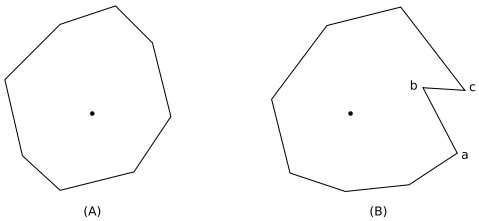


**Figure 2.4** Point-in-polygon determination using the side of the point (dot) in relation to the line segments of a polygon, given a fixed sequence of points (clockwise or counterclockwise). (A) The point is always on the same side of all the segments. (B) The point is on the opposite sides of segments *ab* and *bc*

### 2.7.1 Even–odd algorithm

The even–odd algorithm is a popular method and is also known as the ray-casting or crossing number algorithm. This algorithm runs a crossing test that draws a half-line (or casts a ray) from the point. If the ray crosses the polygon boundary in an odd number of points, then we conclude that the point is inside the polygon. Otherwise, it is outside the polygon. Conveniently, we can draw the half-line horizontally (Figure 2.5). The overall process is straightforward but we hope to avoid the actual calculation of intersections because such calculation can be time-consuming, especially when we have to do it repeatedly.

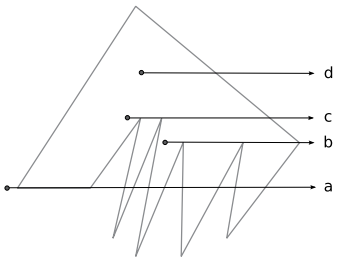


**Figure 2.5** Point-in-polygon algorithm. The start point of half-line *a* is outside the polygon, and the start points of half-lines *b*, *c*, and *d* are inside the polygon

Figure 2.6 illustrates the different cases of whether it is necessary to compute the intersection. In this figure, we have a horizontal half-line that starts at point $A$, and a number of line segments that are labeled as $a$ through $e$ and $b'$. We try to test whether a line segment intersects the half-line without actually trying to compute the intersection point. Segments $a$, $d$, and $e$ will not intersect the half-line because they have both $X$ coordinates on one side of the origin of the half-line ($a$), or both $Y$ coordinates on one side of the half-line ($d$ and $e$). We count segment $c$ as a crossing because we have an intersection, but we do not need to compute the intersection point because we know for sure that segment $c$ crosses the half-line: the $Y$ coordinates of segment $c$ are on different sides of the half-line and both $X$ coordinates are to the right of point $A$. For segments $b$ and $b'$, we cannot conclude whether they intersect the half-line immediately and must