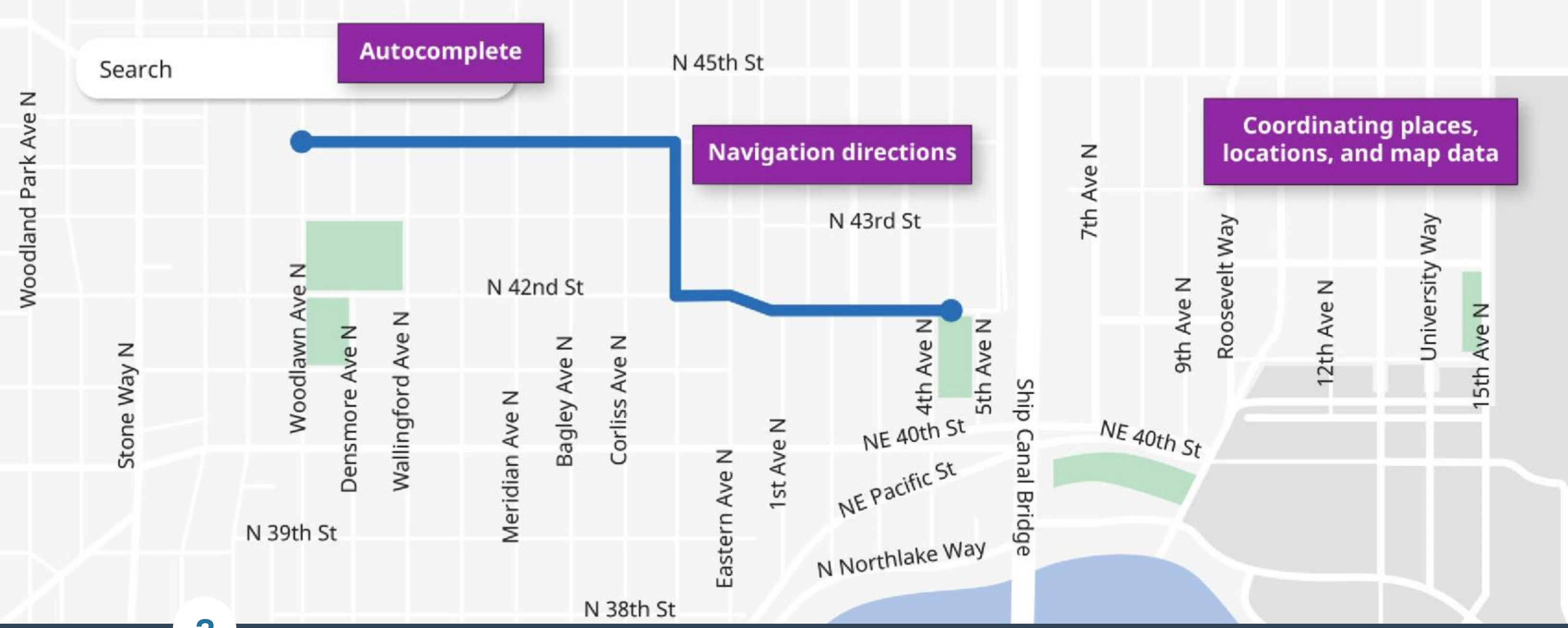


3 Data Structures and Algorithms

Figure 3.1 Online mapping applications represent places, locations, and map data while providing functionality to look around, search for places, and get navigation directions. The right combination of data structures to manage collections of places, locations, and map data along with efficient search and navigation algorithms will help optimize the experience of users trying to find their way through the map and will also make optimal use of computing resources. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license; data source: OpenStreetMap under Open Database License; attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Chapter Outline

3.1 Introduction to Data Structures and Algorithms

3.2 Algorithm Design and Discovery

3.3 Formal Properties of Algorithms

3.4 Algorithmic Paradigms

3.5 Sample Algorithms by Problem

3.6 Computer Science Theory

Introduction

Online maps help people navigate a rapidly changing world. It was not long ago that maps were on paper and that knowledge came from non-digital, trusted sources. In this chapter, we will study how computer scientists design and analyze the foundational structures behind many of today’s technologies. Data structures and algorithms are not only foundational to map apps, but also enable an amazing variety of other technologies too.

From self-driving cars to inventory management to simulating the movement of galaxies to transferring data between computers—all these applications use data structures and algorithms to efficiently organize and process large amounts of information.

3.1 Introduction to Data Structures and Algorithms

Learning Objectives

By the end of this section, you will be able to:

- Understand the difference between algorithms and programs
- Relate data structures and abstract data types
- Select the data structure that is appropriate to solve a given problem practically

Computer science is the study of computers and computational systems that involve data representation and process automation. Owing to their historical roots as calculators, computers can easily represent numerical data. Calculators rely on algorithms to add, subtract, multiply, and divide numbers. But what about more complex data? How do computers represent complex objects like graphs, images, videos, or sentences? What complications arise when we represent or process data in certain ways? These are some of the foundational questions that computer scientists and programmers focus on when designing software and applications that we use to solve problems.

A data type determines how computers process data by defining the possible values for data and the possible functionality or operations on that data. For example, the integer data type is defined as values from a certain range of positive or negative whole numbers with functionality including addition, subtraction, multiplication, and division. The string data type is defined as a sequence of characters where each character can be a letter, digit, punctuation, or space, with functionalities that include adding or deleting a character from a string, concatenating strings, and comparing two strings based, for example, on their alphabetical order.

Data types like strings are an example of abstraction, the process of simplifying a concept in order to represent it in a computer. The string data type takes a complex concept like a sentence and represents it in terms of more basic data that a computer can work with. When a computer compares two strings, it is really comparing the individual numerical character codes (see Chapter 5 Hardware Realizations of Algorithms: Computer Systems Design) corresponding to each pair of characters within the two strings.

In this section, we will learn how to solve problems by choosing abstractions for complex data. We will see that just as our data grows more complex, so do our algorithms.

Introduction to Algorithms

An algorithm is a sequence of precise instructions that operate on data. We can think of recipes, plans, or instructions from our daily lives as examples of algorithms. Computers can only execute a finite pre-defined set of instructions exactly as instructed, which is why programming can feel like such a different way of communicating as compared to our natural human languages. A program is an implementation (realization) of an algorithm written in a formal programming language.

Although each programming language is different from all the others, there are still common ideas across all of them. Knowing just a few of these common ideas enables computer scientists to address a wide variety of problems without having to start from scratch every single time. For example, the abstraction of string data enables programmers to write programs that operate on human-readable letters, digits, punctuation, or spaces without having to determine how to delve into each of these concepts. Programming languages allow us to define abstractions for representing ideas in a computer (see Chapter 4 Linguistic Realization of Algorithms: Low-Level Programming Languages for more).

The study of data structures and algorithms focuses on identifying what is known as a canonical algorithm: a well-known algorithm that showcases design principles helpful across a wide variety of problems. In this chapter, rather than focusing on the programming details, we will instead focus on algorithms and the ideas behind them.

Understanding Data Structures

For many real-world problems, the ability to design an algorithm depends on how the data is represented. A data structure is a complex data type with two equally important parts:

1. a specific representation or way of organizing a collection of more than one element, which is an individual value or data point, and
2. a specific functionality or operations such as adding, retrieving, and removing elements.

In our previous example, a string is a data structure for representing sentences as a sequence of characters. It has specific functionality such as character insertion or deletion, string concatenation, and string comparison.

Although the values for complex data are often diverse, computer scientists have designed data structures so that they can be reused for other problems. For example, rather than designing a specialized data structure for sentences in every human language, we often use a single, universal string data structure to represent sentences, including characters from different languages in the same sentence. (We will later see some drawbacks of generalizing assumptions in the design of data structures and algorithms.) In addition, computers take time to execute algorithms, so computer scientists are concerned about efficiency in terms of how long an algorithm takes to compute a result.

Among the different types of universal data structures, computer scientists have found it helpful to categorize data structures according to their functionality without considering their specific representation. An abstract data type (ADT) consists of all data structures that share common functionality but differ in specific representation.

Common abstract data types for complex data follow, and list and set types are shown in Figure 3.2. We will discuss each abstract data type in more detail together with their data structure implementations.

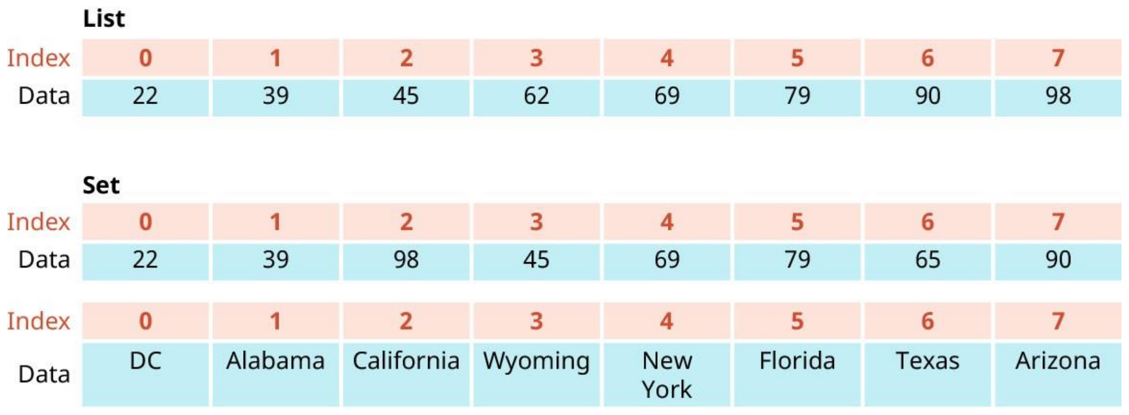
- A list represents an ordered sequence of elements and allows adding, retrieving, and removing elements from any position in the list. Lists are indexed because they allow access to elements by referring to the element's index, which is the position or address for an element in the list. For example, a list can be used to represent a to-do list, where each item in the list is the next task to be completed in chronological order.

- A set represents an unordered collection of unique elements and allows adding, retrieving, and removing elements from the set. Sets typically offer less functionality than lists, but this reduction in functionality allows for more efficient data structure representations. For example, a set can be used to represent the names of all the places that you want to visit in the future.

- A map represents unordered associations between key-value pairs of elements, where each key can only appear once in the map. A map is also known as a dictionary since each term (key) has an associated definition (value). Maps are often used in combination with other data structures. For example, a map can be used to represent a travel wish list: each place that you want to visit in the future can be associated with the list of things that you want to do when you arrive at a given place.

- A priority queue represents a collection of elements where each element has an associated priority value. In addition to adding elements, priority queues focus on retrieving and removing the element with the highest priority. For example, a priority queue can be used to represent inpatient processing at a hospital emergency room: the patients with more urgent need for care may be prioritized and dealt with first.

- A graph represents binary relations among a collection of entities. More specifically, the entities are represented as vertices in the graph, and a directed or undirected edge is added between two vertices to represent the presence or absence of a certain relation. For example, a friendship graph can be used to represent the friendship relations between people, in which case an undirected edge is added between two persons if they are friends. Graphs allow operations such as adding vertices and edges, removing vertices and edges, and retrieving all edges adjacent to a given vertex.



List

Index	0	1	2	3	4	5	6	7
Data	22	39	45	62	69	79	90	98

Set

Index	0	1	2	3	4	5	6	7
Data	22	39	98	45	69	79	65	90

Index	0	1	2	3	4	5	6	7
Data	DC	Alabama	California	Wyoming	New York	Florida	Texas	Arizona

Figure 3.2 Lists and sets are common abstract data types used to represent complex data and can be in the form of integers or string data. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Selecting a Data Structure

Since data representation is a fundamental task in designing algorithms that solve problems, how do we select data structures for a particular problem? Computer scientists apply a top-down approach.

1. Select an appropriate abstract data type by analyzing the problem to determine the necessary functionality and operations.
2. Select an appropriate data structure by quantifying the resource constraints (usually program running time) for each operation.

The primary concern is the data and the operations to be performed on them. Thinking back to simple data types like numbers, we focused on addition, subtraction, multiplication, and division as the basic operations. Likewise, to represent complex data types, we also focus on the operations that will most directly support our algorithms. After deciding on an abstract data type, we then choose a particular data structure that implements the abstract data type. Linear Data Structures

If a problem can be solved with an ordered sequence of elements (e.g., numbers, payroll records, or text messages), the simplest approach might be to store them in a list. Some problems require that actions be performed in a strict chronological order, such as processing items in the order that they arrive or in the reverse order. In these situations, a linear data structure, which is a category of data structures where elements are ordered in a line, is appropriate. There are two possible implementations for the list abstract data type. The first, an array list (Figure 3.3), is a data structure that stores list elements next to each other in memory. The other is a linked list (Figure 3.4), which is a list data structure that does not necessarily store list elements next to each other, but instead works by maintaining, for each element, a link to the next element in the list. Both array lists and linked lists are linear data structures because their elements are organized in a line, one after the other. An advantage of array lists is that they allow (random) access to every element in the list in a single step. This is in sharp contrast with linked lists, which only supports "sequential access." On the other hand, linked lists support fast insertion and deletion operations, which array lists do not.

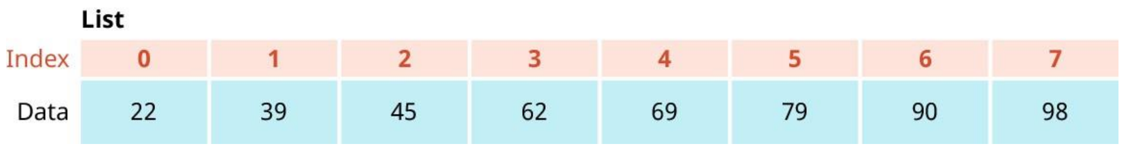


Figure 3.3 An array list stores elements next to each other in memory in the exact list order. (attribution: Copyright Rice University,

OpenStax, under CC BY 4.0 license)

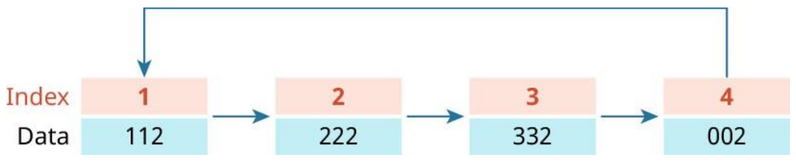


Figure 3.4 A linked list maintains a link for each element to the next element in the list. (attribution: Copyright Rice University,

OpenStax, under CC BY 4.0 license)

Earlier, we introduced sets as offering less functionality than lists. Both array lists and linked lists can also implement the set abstract data type. Sets differ from lists in two ways: sets are unordered—so elements are not assigned specific positions—and sets only consist of unique elements. In addition to implementing sets, linear data structures can also implement the map, priority queue, and graph abstract data types. If linear data structures can cover such a wide range of abstract data types, why learn other data structures? In theory, any complex data can be represented with an array list or a linked list, although it may not be optimal, as we will explain further.

One drawback of relying only on linear data structures is related to the concept of efficiency. Even if linear data structures can solve any problem, we might prefer more specialized data structures that can solve fewer problems more efficiently, and help represent real world data arrangements more closely. This is particularly useful when we have large amounts of data like places or roads in an online map of the entire world. Linear data structures ultimately organize elements in a line, which is necessary for implementing lists but not necessary for other abstract data types. Other data structures specialize in implementing sets, maps, and priority queues by organizing elements in a hierarchy rather than in a line.

Tree Data Structures

A tree is a hierarchical data structure. While there are many kinds of tree data structures, all of them share the same basic organizing structure: a node represents an element in a tree or graph. A node may or may not have a descendant. A child node is a descendant of another node. Often, the primary node is referred to as the “parent node.” Trees maintain a hierarchy through parent-child relationships, which repeat from the root node at the top of the tree down to each leaf node, which is at the bottom of the tree and has no children. The height of a tree corresponds to the depth of the hierarchy of descendants. Figure 3.5 illustrates the structure and elements of a tree.

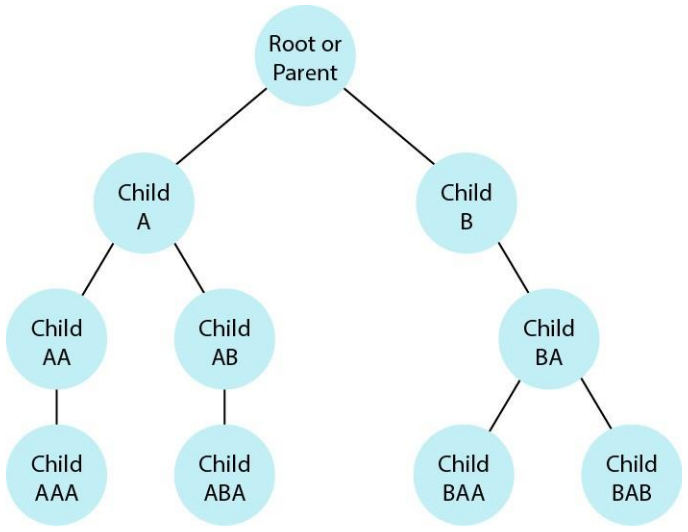


Figure 3.5 A tree is a hierarchical data structure with nodes where each node can have zero or more descendant child nodes. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Binary Search Trees

A binary search tree is a kind of tree data structure often used to implement sets and maps with the binary tree property, which requires that each node can have either zero, one, or two children, and the search tree property, which requires that elements in the tree are organized least-to-greatest from left-to-right. In other words, the values of all the elements of the left subtree of a node have a lesser value than that of the node. Similarly, the values of all the elements of the right subtree of a node have a greater value than that of the node. The search tree property suggests that when elements are read left-to-right in a search tree, we will get the elements in sorted order. For numbers, we can compare and sort numbers by their numeric values. For more complex data like words or sentences, we can compare and sort them in dictionary order. Binary search trees use these intrinsic properties of data to organize elements in a searchable hierarchy (Figure 3.6).

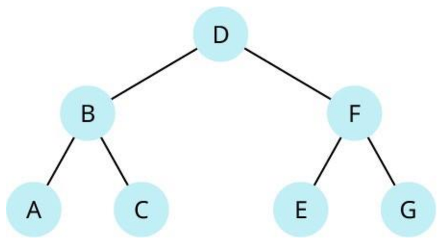


Figure 3.6 A binary search tree organizes elements least to greatest from left to right. (attribution: Copyright Rice University,

OpenStax, under CC BY 4.0 license)

The tree illustrated satisfies the binary tree property based on the natural alphabetical order between letters since the elements in the tree are organized least to greatest from left to right. In other words, for a given list of letters (A, B, C, D, E, F, G), start at the middle of the list of letters with D (the root node) then pick B as the left sub-node of D which is at the middle of the list of letters (A, B, C) that is on the right of D and pick F as the right sub-node of D, which is at the middle of the list letters (E, F, G) that is on the left of D. Finally, organize the remaining letters under sub-nodes B and F to ensure that they are least-to-greatest from the left to right.

The search tree property is responsible for efficiency improvements over linear data structures. By storing elements in a sorted order in the search tree rather than in an indexed order in a list, binary search trees can more efficiently find a given element. Consider how we might look up words in a dictionary. A binary search tree dictionary storing all the terms and their associated definitions can enable efficient search by starting at the middle of the dictionary (the root node) before determining whether to go left or right based on whether we expect our word to appear earlier or later in the dictionary order. If we repeat this process, we can repeatedly rule out half of the remaining elements each time. Searching for a term in a list-based dictionary that is not sorted, on the other hand, would require us to start from the beginning of the list and consider every word until the end of the list since there is no underlying ordering structure to the elements.

Balanced Binary Search Trees

Binary search trees are not as effective as we have described. The dictionary example represents a best-case scenario for binary search trees. We can only rule out half of the remaining elements each time if the binary search tree is perfectly balanced, which means that for every node in the binary search tree, its left and right subtrees contain the same number of elements. This is a strong requirement, since the order in which elements are added to a binary search tree determines the shape of the tree. In other words, binary search trees can easily become unbalanced. It is possible for a binary search tree to look exactly like a linked list, in which each node contains either zero children or one child, which is no more efficient than a linear data structure.

An AVL tree (named after its inventors, Adelson-Velsky and Landis) is a balanced binary search tree data structure often used to implement sets or maps with one additional tree property: the AVL tree property, which requires the left and right subtrees to be balanced at every node of the tree. AVL trees are just one among many "self-balancing" binary search trees. A balanced binary search tree introduces additional properties that ensure that the tree reorganizes elements to maintain balance (Figure 3.7).

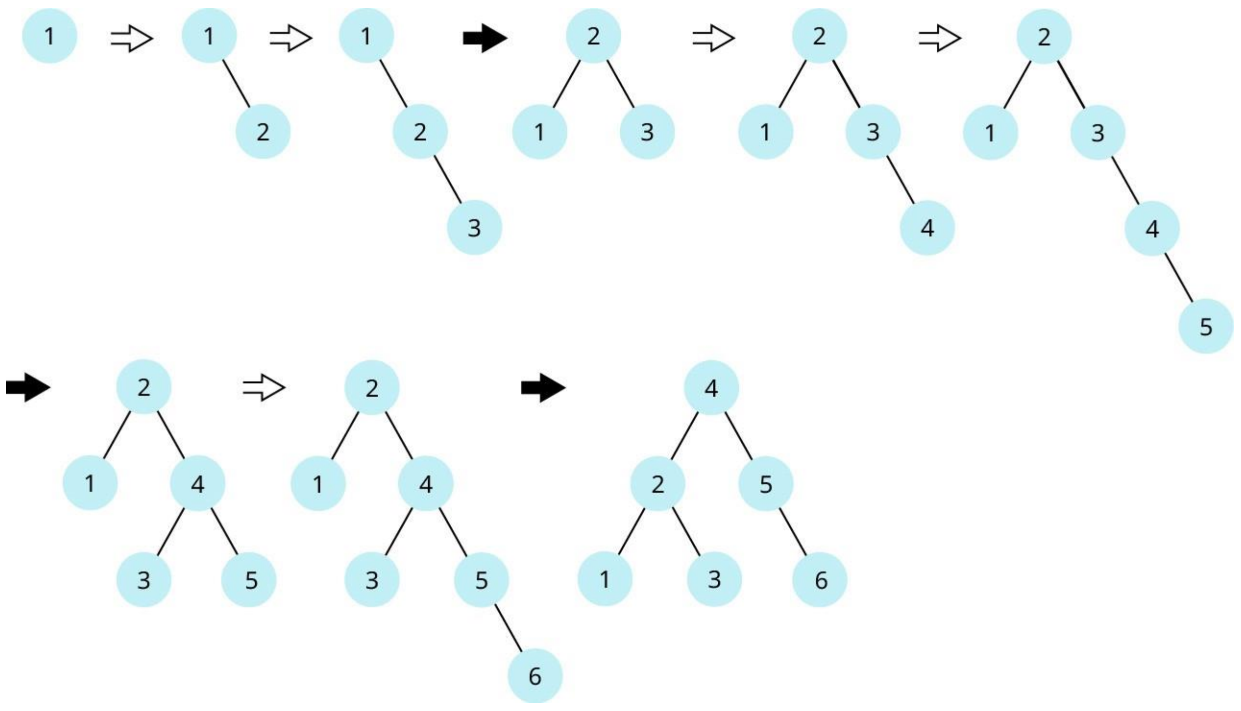


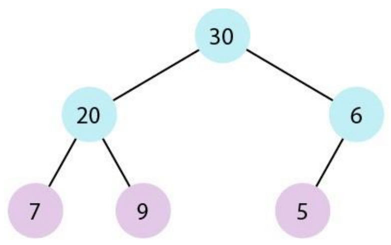
Figure 3.7 An AVL tree rotates nodes in a binary search tree to maintain balance. This sequence of steps illustrates the insertion of numbers 1, 2, 3, 4, 5, 6 into an initially empty AVL tree. (The steps in which rotation occurs are represented by the solid black arrows.) (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Balanced binary search trees such as AVL trees represent just one approach for ensuring that the tree never enters a worst-case situation. There are many other balanced binary search tree data structures in addition to AVL trees. Balanced binary search trees can also be used to implement the priority queue abstract data type if the elements are ordered according to their priority value. But balanced search trees are not the only way to implement priority queues.

Binary Heaps

Priority queues focus on retrieving and removing the highest-priority elements first, adding an element to a priority queue also involves specifying an associated priority value that is used to determine which elements are served next. For example, patients in an emergency room might be served according to the severity of their health concerns rather than according to arrival time. A binary heap is a type of binary tree data structure that is also the most common implementation for the priority queue abstract data type (Figure 3.8).

A binary heap is not a search tree, but rather a hybrid data structure between a binary tree and an array list. Data is stored as an array list in memory, but the binary heap helps visualize data in the same way that a binary tree does, which makes it easier to understand how data are stored and manipulated. Binary heaps organize elements according to the heap property, which requires that the priority value of each node in the heap is greater than or equal to the priority values of its children. The heap property suggests that the highest-priority element will always be the root node where it is efficient to access.



Index	0	1	2	3	4	5
Data	30	20	6	7	9	5

Figure 3.8 A binary heap is the most common implementation of the priority queue abstract data type. The priority value of each node in the binary heap is greater than or equal to the priority values of the children. Note that the value stored in the root node of the right subtree can be smaller than the value stored in any node in the left subtree, while not violating the heap property. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

CONCEPTS IN PRACTICE

Tracking Earthquakes

Earthquakes, hurricanes, tsunamis, and other natural disasters occur regularly, and often demand an international response. How do we track natural disasters and identify the most affected areas in order to coordinate relief and support efforts? In the United States, the U.S. Geological Survey (USGS) is responsible for reporting earthquakes using thousands of earthquake sensors. However, that still leaves many places without earthquake sensors. Outside the United States, sensor technology may be less robust or inaccessible.

Social network data can be used to enhance this information and more quickly alert governments about natural disasters in real-time. By monitoring public social network platforms for occurrences of short posts such as “earthquake?,” we can quickly localize earthquakes based on the user’s geolocation data. However, aggregating and understanding this data—often thousands of data points arriving in minutes—requires efficient data structures and algorithms. We can use a binary heap that implements the priority queue abstract data type for an earthquake-tracking program. For each “earthquake?” post received for a given geolocation, we can increase the priority of the earthquake locations, which helps identify the likelyearthquake location that is closest to the user’s real location. At any time, we can efficiently retrieve the highest-priority element from the priority queue. By choosing to use a binary heap rather than a linear data structure for implementing the priority queue, we can ensure that the earthquake-tracking program is able to keep up with the thousands of posts made every minute during an earthquake.

Graph Data Structures

Both binary search trees and binary heap data structures represent more efficient ways to implement sets, maps, and priority queues by organizing data according to their intrinsic properties. In both cases, the properties of data enable efficient addition, retrieval, and removal of elements.

Graphs are a different kind of abstract data type. Rather than focusing on addition, retrieval, and removal, graphs focus on explicitly modeling the relationships between elements. Graphs afford access not only to elements, but also to relationships between elements.

- A vertex represents an element in a graph or a special type of it, such as a tree.
- An edge is the relationship between vertices or nodes. Optionally, edges can have associated weights. In a graph abstract data type, the relationships between two vertices connected by an edge are considered adjacent.

LINK TO LEARNING

Visualgo is a website that provides animations to help users learn more about algorithms and data structures. They have exercises to help understand several concepts presented in this chapter. You can access animations on various data structures and algorithms (https://openstax.org/r/76Visualgo) such as a linked list, a binary search tree, and graph structures.

In computer networks such as the Internet, graphs can represent individual network routers as nodes with data packets flowing between directly connected routers along edges. Even though not every router is directly connected to every other router, the router at the current node can analyze an incoming data packet to determine which edge it should travel through next. By repeating this process, a data packet can travel from a router on the Internet to another router on the Internet even though the two routers are not directly adjacent to each other.

Graphs are unique in that they can directly represent a wide variety of real-world problems, such as the following:

- A social network, where each vertex is a person, and each edge is a friendship.
- The Web, where each vertex is a webpage, and each edge is a link.
- A campus map, where each vertex is a building, and each edge is a footpath.
- A course prerequisite diagram, where each vertex is a course, and each edge is a prerequisite.

Unlike the list, set, map, and priority queue abstract data types, which have relatively standardized functionality focusing on the addition, retrieval, and removal of elements, the graph abstract data type is much less standardized. Typically, graph algorithm designers will create their own graph data type to represent a problem. The corresponding graph problem can then be represented using or adapting a standard graph algorithm. Unlike programming with other abstract data types, much of the hard work of solving a problem with a graph occurs when programmers decide what the vertices and edges represent, and which graph algorithm would be appropriate to solve the problem. They also must consider the consequences of how they represent the problem.

GLOBAL ISSUES IN TECHNOLOGY

Contact Tracing

Epidemiology is the study of how infectious diseases spread across the world. Within epidemiology, contact tracing attempts to identify confirmed cases of disease and limit its spread by tracing contacted people and isolating them from further spreading the disease.

Graph data structures can help epidemiologists manage the data and people involved in contact tracing. Imagine a graph where each vertex in a tracing graph represents a person, and each edge between two people represents a possible contact. When a person receives a positive test result for contracting the disease, healthcare professionals can identify all the people that they’ve been in contact with by tracing through the graph.

In addition to improving public health through contact tracing, our imaginary graph can also represent a history of the spread of the disease for epidemiologists to understand how the disease moves through communities. For example, if each vertex includes identity characteristic data such as age, race, or gender, epidemiologists can study which groups of people are most affected by the disease. This can then inform the distribution of vaccines to assist the most impacted groups first.

Complex Data

Now that we have seen several data structure implementations for abstract data, let us consider how these data structures are used in practice. Recall that we compared calculators whose algorithms operated on numbers with the idea of a computer whose algorithms operated on complex data. Data structures can be used to represent complex data by modeling hierarchy and relationships.

We might represent an online retail store as a map associating each item with details such as an image, a brief description, price, and the number of items in stock. This map makes some online storefront features easier to implement than others. Given an item, this map makes it easy to retrieve the details associated with that item. On the other hand, it is not so easy to sort items by price, sort items by popularity, or search for an item by keywords in its description. All these features could be implemented with additional data structures. We can combine multiple data structures together to implement these features. In computer science, we use database systems (see Chapter 8 Data Management) that work behind the scenes in many applications to manage these data structures and facilitate long-term storage and access to large amounts of data.

Just as calculators have algorithms for calculating numbers, computers have algorithms for computing complex data. Data structures represent these complex data, and algorithms act on these data structures.

3.2 Algorithm Design and Discovery

Learning Objectives

By the end of this section, you will be able to:

- Understand the approach to solving algorithmic problems
- Explain how algorithm design patterns are used to solve new problems
- Describe how algorithms are analyzed

Our introduction to data structures focused primarily on representing complex data. But computer scientists are also interested in designing algorithms for solving a wider variety of problems beyond storing and retrieving data. For example, they may want to plan a route between a start location and an end location on a map. Although every real-world problem is unique, computer scientists can use a general set of principles to design solutions without needing to develop new algorithms from scratch. Just like how many data structures can represent the same abstract data type, many different solutions exist to solve the same problem.

Algorithmic Problem Solving

An algorithm is a sequence of precise instructions that takes any input and computes the corresponding output, while algorithmic problem-solving refers to a particular set of approaches and methods for designing algorithms that draws on computing's historical connections to the study of mathematical problem solving. Early computer scientists were influenced by mathematical formalism and mathematical problem solving. George Pólya's 1945 book, How to Solve It, outlines a process for solving problems that begins with a formal understanding of the problem and ends with a solution to the problem. As an algorithm's input size is always finite, finding a solution to an algorithmic problem can always be accomplished by exhaustive search. Therefore, the goal of algorithmic problem-solving, as opposed to mathematical problem solving, is to find an

"efficient" solution, either in terms of execution time (e.g., number of computer instructions) or space used (e.g., computer memory size). Consequently, the study of algorithmic problem-solving emphasizes the formal problem or task, with specific input data and output data corresponding to each input. There are many other ways to solve problems with computers, but this mathematical approach remains the dominant approach in the field. Here are a few well-known problems in computer science that we will explore later in this chapter.

A data structure problem is a computational problem involving the storage and retrieval of elements for implementing abstract data types such as lists, sets, maps, and priority queues. These include:

- searching, or the problem of retrieving a target element from a collection of elements

- sorting, or the problem of rearranging elements into a logical order
- hashing, or the problem of assigning a meaningful integer index for each object

A graph problem is a computational problem involving graphs that represent relationships between data. These include:

- traversal, or the problem of exploring all the vertices in a graph
- minimum spanning tree is the problem of finding a lowest-cost way to connect all the vertices to each other
- shortest path is the problem of finding the lowest-cost way to get from one vertex to another

A string problem is a computational problem involving text or information represented as a sequence of characters. Examples include:

- matching, or the problem of searching for a text pattern within a document
- compression, or the problem of representing information using less data storage
- cryptography, or the problem of masking or obfuscating text to make it unintelligible

Modeling

Computer scientists focus on defining a problem model, often simply called a model, which is a simplified, abstract representation of more complex real-world problems. They apply the algorithmic problem-solving process mentioned previously to design algorithms when defining models. Algorithms model phenomena in the same way that data structures implement abstract data types such as lists, sets, maps, priority queues, and graphs. But unlike abstract data types, models are not necessarily purely abstract or mathematical concepts. Models are often linked to humans and social phenomena. A medical system might want to decide which drugs to administer to which patients, so the algorithm designer might decide to modelpatients as a complex data type consisting of age, sex, weight, or other physical characteristics. Because models represent

abstractions, or simplifications of real phenomena, a model must emphasize some details over others. In the case of the medical system, the algorithm designer emphasized physical characteristics of people that were deemed important and chose to ignore other characteristics, such as political views, which were deemed less important for the model.

If an algorithm is a solution to a problem, then the model is the frame through which the algorithm designer defines the rules and potential outcomes. Without models, algorithm designers would struggle with the infinite complexity and richness of the world. Imagine, for example, designing a medical system that models patients at the level of individual atoms.

This model offers a detailed representation of each patient in the most physical or literal sense. But this model is impractical because we do not know how particular configurations and collections of atoms contribute to a person's overall health. Compared to this atomic-scale model, our former model consisting of age, sex, weight, and other physical characteristics is more practical for designing algorithms, but necessarily involves erasing our individual humanity to draw certain conclusions.

In order to design algorithms, we need to be able to focus on relevant information rather than detailed representations of the real world. Further, computer science requires a philosophical mind to aid in problem solving. According to Brian Cantwell Smith, philosopher and cognitive and computer scientist, "Though this is not the place for metaphysics, it would not be too much to say that every act of conceptualization, analysis, or categorization, does a certain amount of violence to its subject matter, in order to get at the underlying regularities that group things together."[1] Without performing this "violence," there would be too many details to wade through to create a useful algorithm.

The relationship between algorithms, the software they empower, and the social outcomes they produce is currently the center of contested social and political debate. For example, all media platforms (e.g., Netflix, Hulu, and others) use some level of targeted advertising based on user preferences in order to recommend

1 B. C. Smith, "The limits of correctness." ACM SIGCAS Comput. Soc., vol. 14, 15, no. 1, 2, 3, 4, pp. 18-26, Jan. 1985. https://doi.org/ 10.1145/379486.379512.

specific movies or shows to their users. Users may not want their information to be used in this way, but there must be some degree of compromise to make these platforms attractive and useful to people.

On the one hand, the technical definition of an algorithm is that it represents complex processes as a sequence of precise instructions operating on data. This definition does not overtly suggest how algorithms encode social outcomes. On the other hand, computer programs are human-designed and socially engineered. Algorithm designers simplify complex real-world problems by removing details so that they can be modeled as computational problems. Because software encodes and automates human ideas with computers, software engineers wield immense power through their algorithms.

To further complicate the matter, software engineering is often a restrictive and formal discipline. Problem modeling is constrained by the model of computation, or the rules of the underlying computer that is ultimately responsible for executing the algorithm. Historically, computer science grew from its foundations in mathematics and formal logics, so algorithms were specialized to solve specific problems with a modest model of the underlying phenomena. This approach to algorithm design solves certain types of problems so long as they can be reasonably reduced to models that operate on a modest number of variables—however many variables the algorithm designer can keep in mind. In the case of the medical system, the algorithm designer identified certain characteristics as particularly useful for computing a result.

But there are many other problems that defy this approach, particularly tasks that involve subtle and often unconscious use of human sensory and cognitive faculties. An example of this is facial recognition. If asked to describe how we recognize a particular person’s face, an algorithm designer would be challenged to identify specific variables or combinations of variables that correspond to only a single person. The formal logic required to define an algorithm is strict and absolute, whereas our understanding human faces is defined by many subtle factors that are difficult for anyone to express using formal logic.

INDUSTRY SPOTLIGHT

Machine Learning Algorithms

A machine learning algorithm addresses these kinds of problems by using an alternative model of computation, one that focuses on generalized algorithms designed to solve problems with a massive model of the underlying phenomena. Instead of attempting to identify a few key variables for facial recognition, for instance, machine learning algorithms can take as input a digital image represented as a rectangular grid of colored pixels. While each pixel in the image offers very little information about the person in mind, the facial features unique to each human arise from the arrangements and patterns of pixels that result from seeing many images of the same person.

Think about the way your Apple iPhone or Google Pixel phone may look at you when you try to access it and have facial recognition enabled. The algorithm is not going to try to match your face to a saved picture of you because it would not work all the time if you do not look exactly like you did in the picture. Rather, it uses machine learning to extract patterns out of a person's face and match them, making it possible to recognize people all the time even if they are wearing glasses but was not wearing them when they set up facial recognition on their phone. This method does seem to mimic the way humans recognize people, even if they have not seen them for decades.

Machine learning algorithms offer a more robust approach to modeling these kinds of problems that are not easily expressed in formal logic. But in this chapter, we focus on the earlier, classical perspective on algorithmic problem-solving with the end goal of designing specialized algorithms to solve problems with modest models of the underlying phenomena.

Search Algorithms

In computer science, searching is the problem of retrieving a target element from a collection that contains many elements. There are many ways to understand search algorithms; depending on the exact context of the problem and the input data, the expected output might differ. For example, suppose we want to find the target term in a dictionary that contains thousands or millions of terms and their associated definitions. If we represent this dictionary as a list, the search algorithm would return the index of the term in the dictionary. If we represent this dictionary as a set, the search algorithm would return whether the target is in the dictionary. If we represent this dictionary as a map, the search algorithm would return the definition associated with the term. The dictionary data structure has implications on the output of the search algorithm. Algorithmic problem-solving tends to be iterative because we might sometime later realize that our data structures need to change. Changing the data structures, in turn, often also requires changing the algorithm design.

Despite these differences in output, the underlying canonical searching algorithm can still follow the same general procedure. The two most well-known canonical searching algorithms are known as sequential search and binary search, which are conducted on linear data structures, such as array lists.

- Sequential search (Figure 3.9). Open the dictionary to the first term. If that term happens to be the target, then great—we have found the target. If not, then repeat the process by reading the next term in the dictionary until we have checked all the terms in the dictionary.
- Binary search (Figure 3.10). Open the dictionary to a term in the middle of the dictionary. If that term happens to be the target, then great—we have found the target. If not, then determine whether the target comes before or after the term we just checked. If it comes before, then repeat the process except on the first half of the dictionary. Otherwise, repeat the process on the second half of the dictionary. Each time, we can ignore half of the remaining terms based on the place where we would expectto find the target in the dictionary.

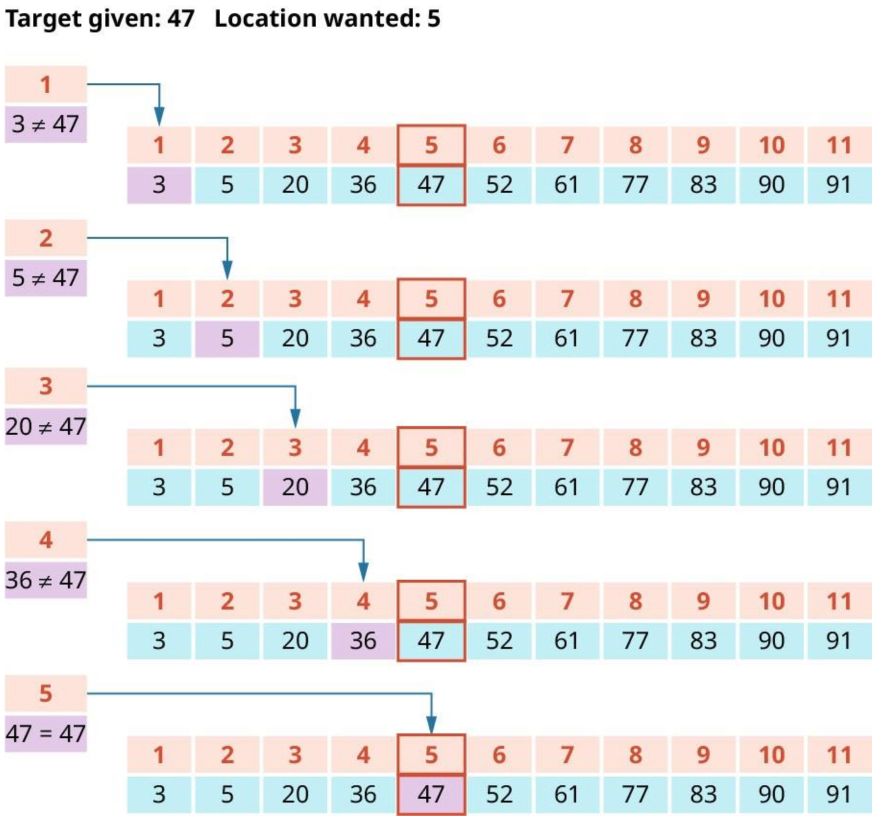


Figure 3.9 A sequential search can find the number 47 in an array by checking each number in order. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

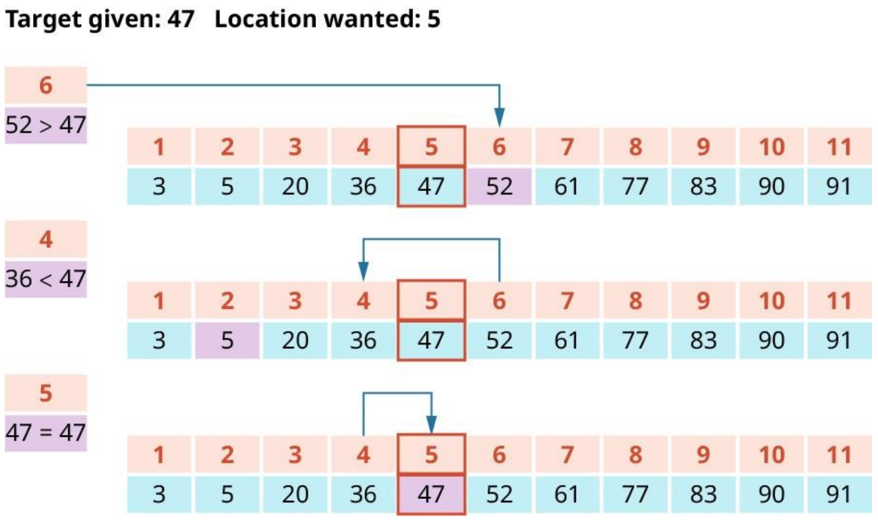


Figure 3.10 A binary search can find the number 47 in an array by determining whether the desired number comes before or after a chosen number. It eliminates half of existing data points and then searches in the remaining half, repeating the pattern, until the number is found. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Algorithm Design Patterns

This case study of canonical search algorithms demonstrates some ideas about algorithmic problem-solving, such as how algorithm design involves iterative improvement (from sequential search to binary search). But this case study does not demonstrate how algorithms are designed in practice. Algorithm designers are occasionally inspired by real-world analogies and metaphors, such as relying on sorted order to divide a dictionary into two equal halves. More often, they depend on knowledge of an existing algorithm design pattern, or a solution to a well-known computing problem, such as sorting and searching. Rather than develop wholly new ideas each time they face a new problem, algorithm designers instead apply one or more algorithm design patterns to solve new problems. By focusing on algorithm design patterns, programmers can solve a wide variety of problems without having to invent a new algorithm every time.

For example, suppose we want to design an autocomplete feature, which helps users as they type text into a program by offering word completions for a given prefix query. Algorithm designers begin by modeling the problem in terms of more familiar data types.

- The input is a prefix query, such as a string of letters that might represent the start of a word (e.g., “Sea”).
- The output is a list of matching terms (completion suggestions) for the prefix query.

In addition to the input and output data, we assume that there is a list of potential terms that the algorithm will use to select the matching terms.

There are a few different ways we could go about solving this problem. One approach is to apply the sequential search design pattern to the list of possible words (Figure 3.11). For each term in the list, we add it to the result if it matches the prefix query. Another approach is to first sortthe list of potential terms and then apply two binary searches: the first binary search to find the first matching term and the second binary search to find the last matching term. The output list is all the terms between the first match and the last match.

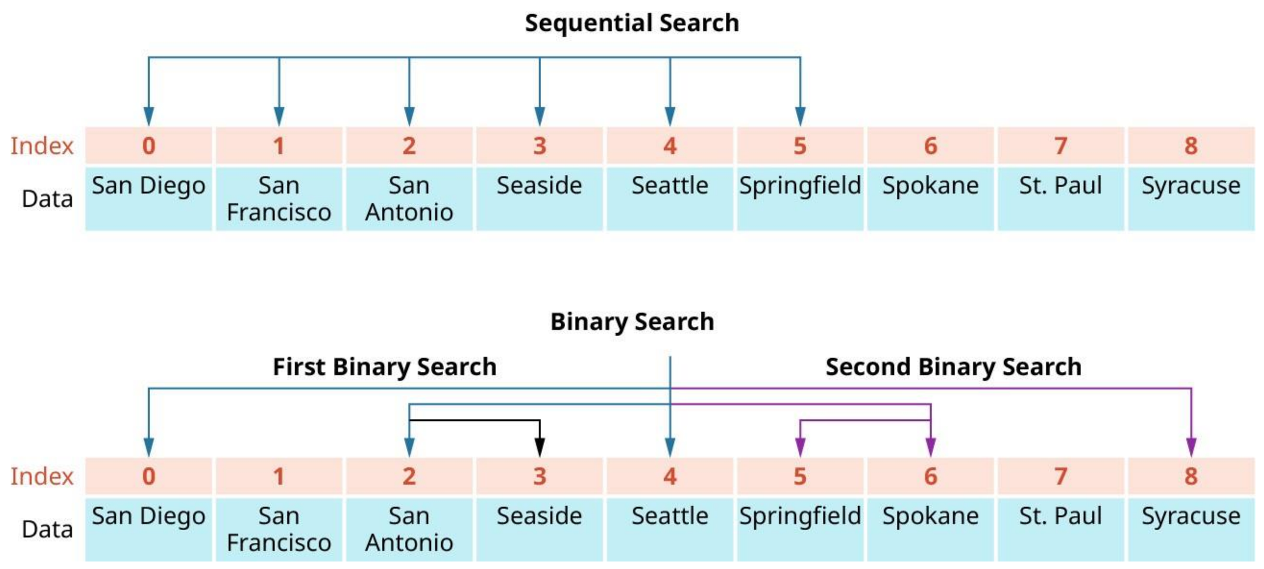


Figure 3.11 Sequential search needs to check every term to see if it matches the prefix "Sea," whereas two binary searches can be used to find the start and end points of the matching terms in the list. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

TECHNOLOGY IN EVERYDAY LIFE

Online Autocomplete Algorithms

In online mapping, autocomplete might take the prefix Seaand automatically suggest the city Seattle. We know that search algorithms solve this problem by maintaining a sorted collection of place suggestions. But many online mapping applications allow users to change maps as the places change in the real world. How can we design the autocomplete feature to support real-world user changes? To apply the binary search algorithm, all place names must be stored in a sorted array-based list. So, every change will also need to maintain the sorted order.

If we instead add all the place names to a binary search tree, what are the steps for the autocomplete algorithm? How does this choice affect additions, changes, and removals?

Algorithm Analysis

Rather than rely on a direct analogy to our human experiences, these two algorithms for the autocomplete feature compose one or more algorithm design patterns to solve the problem. How do we know which approach might be more appropriate to use? One type of analysis is known as algorithm analysis, which studies the outputs produced by an algorithm as well as how the algorithm produces those outputs. Those outputs are then evaluated for correctness, which considers whether the outputs produced by an algorithm match the expected or desired results across the range of possible inputs. An algorithm is considered correct only if its computed outputs are consistent with all the expected outputs; otherwise, the algorithm is considered incorrect.

Although this definition might sound simple, verifying an algorithm for correctness is often quite difficult in practice, because algorithms are designed to generalize and automate complex processes. The most direct way to verify correctness is to check that the algorithm computes the correct output for everypossible input. This is not only computationally difficult, but even potentially impossible to achieve, since some algorithms can accept an infinite range of inputs.

Verifying the correctness of an algorithm is difficult not only due to generality, but also due to ambiguity.

Earlier, we saw how canonical searching algorithms may have different outputs according to the input collection type. What happens if the target term contains special characters or misspellings? Should the algorithm attempt to find the closest match? Some ambiguities can be resolved by being explicit about the expected output, but there are also cases where ambiguity simply cannot be resolved in a satisfactory way. If we decide to find the closest match to the target term, how does the algorithm handle cultural differences in interpretation? If humans do not agree on the expected output, but the algorithm mustcompute some output, what output does it then compute? Or, if we do not want our algorithms to compute a certain output, how does it recognize those situations?

Correctness primarily considers consistency between the algorithm and the model, rather than the algorithm and the real world. Our autocomplete model from earlier returned all word completions that matched the incomplete string of letters. But in practice, this output would likely be unusable: a user typing "a" would see a list of all words starting with the letter "a." Since our model did not specify how to order the results, the user might get frustrated by the irrelevancy of many of the word completions. Suppose we remedy this issue by defining a relevancy metric: every time a user completes typing a word, increase that word's relevancy for future autocompletion requests. But, as Safiya Noble showed in Algorithms of Oppression, determining relevance in this universalizing way can have undesirable social impacts. Perhaps due to relevancy metrics determined by popular vote, at one point, Google search autosuggestions included ideas like:

- Women cannot: drive, be bishops, be trusted, speak in church
- Women should not: have rights, vote, work, box
- Women should: stay at home, be in the kitchen
- Women need to: be put in their places, know their place, be controlled, be disciplined[2]

Noble’s critique extends further to consider the intersection of social identities such as race and gender as they relate to the outputs of algorithms that support our daily life.

GLOBAL ISSUES IN TECHNOLOGY

Searching for Identity

In Algorithms of Oppression, Safiya Noble describes how search engines can amplify sexist, racist, and misogynistic ideas. While searching for “Black girls,” “Latina girls,” and “Asian girls” circa 2013, Safiya was startled by how many of the top search results and advertisements that appeared on the first page of Google Search led to pornographic results when her input query did not at all suggest anything pornographic. In contrast, searching for “White girls” did not include pornographic results. As algorithms become more commonplace in our daily lives, they also become a more potent force for determining certain social futures. Algorithms are immensely powerful in their ability to affect not only how we act, but also what we see, what we hear, what we believe about the world, and even what we believe about ourselves.

As the amount of input data increases, computers often need more time or storage to execute algorithms. This condition is known as complexity, which is based on the degree computational resources that an algorithm consumes during its execution in relation to the size of the input. More computational time also often means consuming more energy. Given the exponential explosion in demand for data and computation, designing efficient algorithms is not only of practical value but also existential value as computing contributes directly to global warming and resultant climate crises.

---

2 S.U. Noble, Algorithms of Oppression: How Search Engines Reinforce Racism. NYU Press, 2018.

THINK IT THROUGH

Content Moderation

Online social media platforms facilitate social relationships between users by allowing users to create and share content with each other. This user-generated content requires moderation, or methods for managing content shared between the platform users. Some researchers argue that content moderation defines a social network platform; in other words, content moderation policies determine exactly what content can be shared on the platform, which in turn defines the value of information. As social media platforms become increasingly prevalent, the information on these platforms plays an important role in influencing their users.

One approach for content moderation is to recruit human moderators to review toxic content, or content that is profane, abusive, or otherwise likely to make someone disengage. An algorithm could be developed to determine the toxicity of a piece of content, and the most toxic content could be added to a priority queue for priority moderation.

What are the consequences of this approach? How does the definition of toxicity prioritize (or de-prioritize) certain content? Who does it benefit? Consider the positionality of the users that interact with the platform:

- marginalized users of the platform, who may be further marginalized by this definition of toxicity.
- content moderators, who are each reviewing several hundred pieces of the most toxic content for hours every day.
- legal teams, who want to mitigate government regulations and legislation that are not aligned with corporate interests.
- social media hackers, or users who want to leverage the way certain content is prioritized in order to deliberately shape public opinion.

3.3 Formal Properties of Algorithms

Learning Objectives

By the end of this section, you will be able to:

- Understand time and space complexity
- Compare and contrast asymptotic analysis with experimental analysis
- Explain the Big O notation for orders of growth

Beyond analyzing an algorithm by examining its outputs, computer scientists are also interested in examining its efficiency by performing an algorithmic runtime analysis, a study of how much time it takes to run an algorithm.

If you have access to a runnable program, perhaps the most practical way to perform a runtime analysis is to time exactly how long it takes to run the program with a stopwatch. This approach, known as experimental analysis, evaluates an algorithm's runtime by recording how long it takes to run a program implementation of it. Experimental analysis is particularly effective for identifying performance bugs or code that consumes unusually large amounts of computation time or system resources, even though it produces the correct output. In e-commerce, for example, performance bugs that result in slow website responsiveness can lead to millions of dollars in lost revenue. In the worst-case scenario, performance bugs can even bring down entire websites and networks when systems are overloaded and cannot handle incoming requests. As the Internet becomes more heavily used for information and services, performance bugs can have direct impacts on health and safety if the computer infrastructure cannot keep up with demand.

While experimental analysis is useful for improving the efficiency of a program, it is hard to use if we do not already have a working program. Programming large systems can be expensive and time-consuming, so many organizations want to compare multiple algorithm designs and approaches to identify the most suitable design beforeimplementing the system. Even with sample programs to represent each algorithm design, we can get different results depending on the processing power, amount of memory available, and other features of the computer that is running the program.

Designing more efficient algorithms is not just about solving problems more quickly, but about building a more sustainable future. In this section, we will take a closer look at how to formally describe the efficiency of an algorithm without directly executing a working program.

CONCEPTS IN PRACTICE

Performance Profiling

Modern computer systems are complicated. Algorithms are just one component in a much larger ecosystem that involves communication between many other subsystems, other computers in a data center, and other systems on the Internet. Algorithmic runtime analysis focuses on the properties of the algorithm rather than all the different ways the algorithm interacts with the rest of the world. But once an algorithm is implemented as a computer program, these interactions with the computing ecosystem play an important role in determining program performance.

A profiler is a tool that measures the performance (runtime and memory usage) of a program. Profilers are commonly used to diagnose real-world performance issues by producing graphs of how computational resources are used in a program. A common graph is a flame graph (Figure 3.12) that visualizes resource utilization by each part of a program to help identify the most resource-intensive parts of a program. Saving even a few percentage points of resources can lead to significantly reduced time, money, and energy expenditure. As the global demand for computation continues to increase, performance engineers who know how to leverage profilers to analyze systems and implement resource-saving changes will be key to a green energy future.

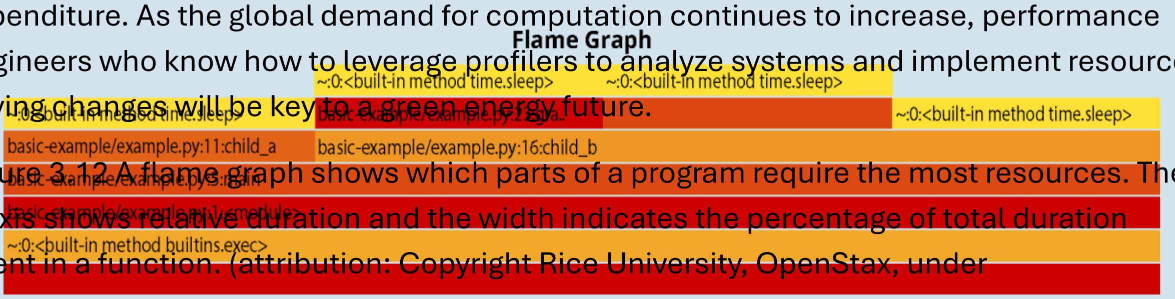


Figure 3.12 A flame graph shows which parts of a program require the most resources. The x-axis shows relative duration and the width indicates the percentage of total duration spent in a function. (attribution: Copyright Rice University, OpenStax, under

CC BY 4.0 license)

Time and Space Complexity

One way to measure the efficiency of an algorithm is through time complexity, a formal measure of how much time an algorithm requires during execution as it relates to the size of the problem. In addition to time complexity, computer scientists are also interested in space complexity, the formal measure of how much memory an algorithm requires during its execution as it relates to the size of the problem.

Both time and space complexity are formal measures of the efficiency of an algorithm as it relates to the size of the problem, particularly when we are working with large amounts of complex data. For example, gravity, the universal phenomenon by which things attract and move toward each other, can be modeled as forces that act on every pair of objects in the universe. Simulating a subset of the universe that contains only 100 astronomical bodies will take a lot less time than a much larger universe with billions, trillions, or even more bodies, all of which gravitate toward each other. The size of the problem plays a large role in determining how much time an algorithm requires to execute. We will often express the size of the problem as a positive integer number corresponding to the size of the dataset such as the number of astronomical bodies in our simulation.

The goal of time and space complexity analysis is to produce a simple and easy-to-compare characterization of the efficiency of an algorithm as it relates to the size of the problem. Consider the following description of an algorithm that searches for a target word in a list. Start from the very beginning of the list and check if the first word is the target word. If it is, we have found the word. If it is not, then continue to the next word in the list and repeat the process.

The first task is to identify a metric for representing the size of the problem. Typically, time complexity analysis assumes asymptotic analysis, focusing on evaluating the time that an algorithm takes to produce a result as the size of the input increases. There are two inputs to this algorithm: (1) the list of words, and (2) the target word. The average length of an English word is about five characters, so the size of the problem is primarily determined by the number of words in the list rather than the length of any word. (This assumption might not be right if our dataset was instead a DNA sequence consisting of millions of nucleotides—the time it takes to compare a pair of long DNA sequences might be more important than the number of DNA sequences being compared.) Identifying the size of the problem is an important first task because it determines the other factors we can consider in the following tasks.

The next task is to model the number of steps needed to execute the algorithm while considering its potential behavior on all possible inputs. A step represents a basic operation in the computer, such as looking up a single value, adding two values, or comparing two values. How does the runtime change as the size of the problem increases? We can see that the "repeat" part of our description is affected by the number of words in the list; more words can potentially lead to more repetitions.

In this case we are choosing a cost model, which is a characterization of runtime in terms of more abstract operations, such as the number of repetitions. Rather than count single steps, we instead count repetitions. Each repetition can involve several lookups and comparisons. By choosing each repetition as the cost model, we declare that the few steps needed to look up and compare elements can be effectively treated as a single operation to simplify our analysis.

However, this analysis is not quite complete. We might find the target word early in the list even if the list is very large. Although we defined the size of the problem as the number of words in the list, the size of the problem does not account for the exact words and word ordering in the list. Computer scientists say that this algorithm has a best-case situation when the word can be found at the beginning of the list, and a worst-case situation when the word can only be found at the end of the list (or, perhaps, not even in the list at all). One way to account for the variation in runtime is via case analysis, which is based on factors other than the size of the problem.

Finally, we can formalize our description using either precise English or a special mathematical notation called Big O notation, which is the most common type of asymptotic notation in computer science used to measure worst-case complexity. In precise English, we might say that the time complexity for this sequential search algorithm has two cases (Figure 3.13):

- In the best-case situation (when the target word is at the start of the list), sequential search takes just one repetition to find the word in the list.
- In the worst-case situation (when the target word is either at the end of the list or not in the list at all), sequential search takes Nrepetitions where Nis the number of words in the list.

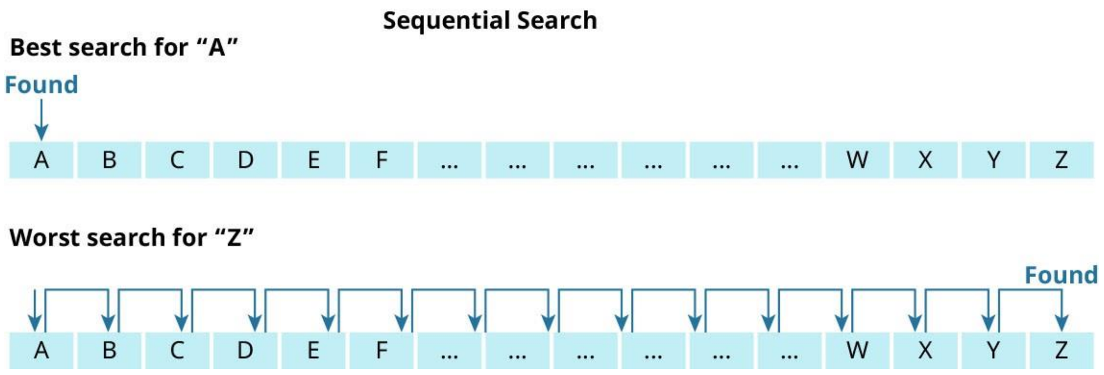


Figure 3.13 The best case for sequential search in a sorted list is to find the word at the top of the list, whereas the worst case is to find the word at the bottom of the list (or not in the list at all). (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

While this description captures all the ideas necessary to communicate the time complexity, computer scientists will typically enhance this description with mathematics to convey a geometric idea of the runtime. The order of growth is a geometric prediction of an algorithm's time or space complexity as a function of the size of the problem (Figure 3.14).

- In the best case, the sequential search has a constant order of growth that does not take more resources as the size of the problem increases.
- In the worst case, the sequential search has a linear order of growth where the resources required to run the algorithm increase at about the same rate as the size of the problem increases. This is with respect to N, the number of words in the list.

Constant and linear are two examples of orders of growth. An algorithm with a constant order of growth takes the same amount of time to execute even as the size of the problem grows larger and larger—no matter how large the dictionary is, it is possible to find the target word at the very beginning. In contrast, an algorithm with a linear time complexity will take more time to execute as the size of the problem grows larger, and we can predict that an increase in the size of the problem corresponds to roughly the same increase in the runtime.

This prediction is a useful outcome of time complexity analysis. It allows us to estimate the runtime of the sequential search algorithm on a problem of any size, before writing the program or obtaining a dictionary of words that large. Moreover, it helps us decide if we want to use this algorithm or explore other algorithm designs and approaches. We might compare this sequential search algorithm to the binary search algorithm and adjust our algorithm design accordingly.

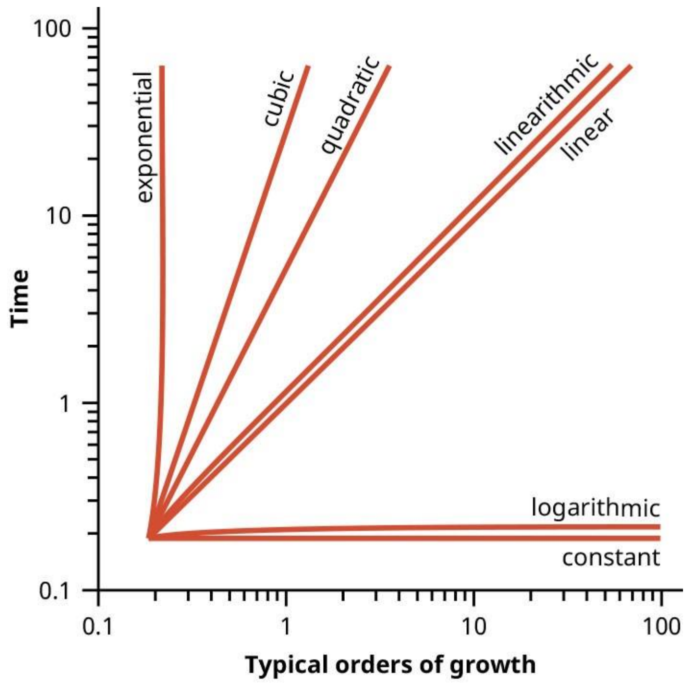


Figure 3.14 The order of growth of an algorithm is useful to estimate its runtime efficiency as the input size increases (e.g., constant, logarithmic, and other orders of growth) to help determine which algorithmic approach to take. (attribution: Copyright Rice

University, OpenStax, under CC BY 4.0 license)

Big O Notation for Orders of Growth

In the 1970s, computer scientists applied asymptotic notation, a mathematical notation that formally defines the order of growth. We can use Big O notation to describe the time complexity of the sequential search algorithm. In general, we say a function f(N) is in the class of O(g(N)), denoted by f(N) = O(g(N)) or f(N) in O(g(N)), if when N tends to infinity, the ratio f(N)/g(N) is upper bounded by some constant. The function g(N) is usually some simple function that defines the order of growth such as g(N) = 1 (constant function), g(N) = N (linear function), g(N) = log N(logarithmic function), or other functions as follows:

- In the best case, the order of growth of sequential search is in O(1).
- In the worst case, the order of growth of sequential search is in O(N) with respect to N, the number of words in the list.

The constant order of growth is described in Big O notation as “O(1)” while the linear order of growth is described in Big O notation as “O(N) with respect to N, the size of the problem.” Big O notation formalizes the concept of a prediction. Given the size of the problem, N, calculate how long it takes to run the algorithm on a problem of that size. For large lists, in order to double the worst-case runtime of sequential search, we would need to double the size of the list.

O(1) and O(N) are not the only orders of growth.

- O(1), or constant.
- O(log N), or logarithmic.
- O(N), or linear.
- O(Nlog N), or linearithmic.
- O(N^2), or quadratic.
- O(N^3), or cubic.
- O(2^N), O(3^N), . . . , or exponential.
- O(N!), or factorial.

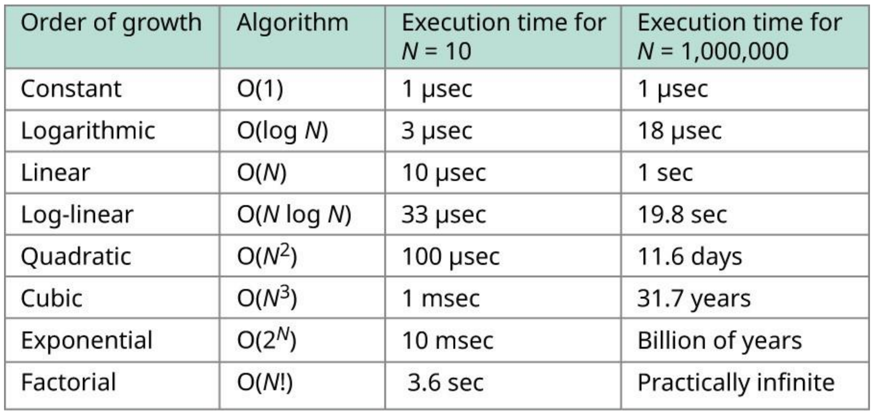
The O(log N), or logarithmic, order of growth appears quite often in algorithm analysis. The logarithm of a large number tells how many times it needs to be divided by a small number until it reaches 1. The binary logarithm, or log2, tells how many times a large number needs to be divided by 2 until it reaches 1. In the worst case, the time complexity of sequential search is in O(N) with respect to N, the number of words in the list, since each repetition of sequential search rules out one remaining element. How about binary search? In the worst case, the time complexity of binary search is in O(log N) with respect to N, the number of words in the list, since each repetition of binary search rules out half the remaining elements.

Another way to understand orders of growth is to consider how a change in the size of the problem results in a change to the resource usage. When we double the size of the input problem, algorithms in each order of growth respond differently (Figure 3.15).

- O(1) algorithms will not require any more resources.
- O(log N) algorithms will require 1 additional resource unit.
- O(N) algorithms will require 2 times the number of resources.
- O(Nlog N) algorithms will require a little more than 2 times the number of resources.
- O(N^2) algorithms will require 4 times the number of resources.
- O(N^3) algorithms will require 8 times the number of resources.
- O(2^N), O(3^N), . . . algorithms will require the squared or cubed number of resources.

- O(N!) algorithms will require even more.

This growth compounds, so quadrupling the size of the problem for an O(N^2) algorithm will require 16 times the number of resources. Algorithm design and discovery is often motivated by these massive differences between orders of growth. Note that this explanation of how each order of growth responds differently oversimplifies the problem. Rigorously speaking, a function f(N) expressed in the big-O notation as being is in the class of O(g(N)) can be much more complex than the simple function g(N). For example, f(N) = 4log N+ 100 log(log N) is in O(log N), but when Ndoubles, f(2N) is definitely not just one unit more than the original function f(N). A similar argument applies for all other functions other than the constant O(1) function.



Order of growth	Algorithm	Execution time for N = 10	Execution time for N = 1,000,000
Constant	O(1)	1 μsec	1 μsec
Logarithmic	O(log N)	3 μsec	18 μsec
Linear	O(N)	10 μsec	1 sec
Log-linear	O(N log N)	33 μsec	19.8 sec
Quadratic	O(N^2)	100 μsec	11.6 days
Cubic	O(N^3)	1 msec	31.7 years
Exponential	O(2^N)	10 msec	Billion of years
Factorial	O(N!)	3.6 sec	Practically infinite

Figure 3.15 This chart shows the time it would take for an algorithm with each of the given orders of growth to finish running on a problem of the given size, N. When an algorithm takes longer than 1025 years to compute, that means it takes longer than the current age of the universe. (data source: Geeks for Geeks, "Big O Notation Tutorial—A Guide to Big O Analysis," last updated March 29, 2024; attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

As the size of the problem increases, algorithmic complexity becomes a larger and larger factor. For problems dealing with just 1,000 elements, the time it would take to run an exponential-time algorithm on that problem exceeds the current age of the universe. In practice, across applications working with large amounts of data, O(Nlog N) is often considered the limit for real-world algorithms. Even then, O(Nlog N) algorithms cannot be run too frequently. For algorithms that need to run frequently on large amounts of data, algorithm designers target O(N), O(log N), or O(1).

TECHNOLOGY IN EVERYDAY LIFE

Arranging Invisible Icons in Quadratic Time

Have you ever been annoyed by computer slowness? For some users, opening the start menu can take 20 seconds because of an $O(N^2)$ algorithm, where Nis the number of desktop files. The Microsoft Windows computer operating system allows users to organize files directly on top of their desktop wallpaper. A quadratic-time algorithm arranges these desktop iconsin a grid layout so that they fill column-by-column starting from the left side of the screen. The algorithm is executed whenever the user opens the start menu or launches the file explorer.

Most users only keep a couple dozen desktop icons, so the $O(N^2)$ algorithm takes microseconds—practically unnoticeable. But for users with hundreds of desktop icons, the impact of each additional icon adds up. With 1,000 desktop files, launching the start menu takes 20 seconds. With 10,000 desktop icons, the runtime grows to 30 minutes!

To avoid the clutter of thousands of desktop icons, users can hide desktop icons. But this does not prevent the quadratic-time algorithm from running. Users with too many desktop icons beware: your computer slowness may be due to arranging invisible icons in quadratic time.[3]

3.4 Algorithmic Paradigms

Learning Objectives

By the end of this section, you will be able to:

- Apply the divide and conquer technique
- Explain the brute-force method
- Interpret and apply the greedy method
- Understand how to apply reductions to solve problems

Algorithm design patterns are solutions to well-known computing problems. In 3.5 Sample Algorithms by Problem, we will survey algorithm design patterns by problem. As it turns out, many of these algorithm design patterns share similarities in their approaches to solving problems. Here, we will introduce the algorithmic paradigm, which is the common concepts and ideas behind algorithm design patterns.

Divide and Conquer Algorithms

A divide and conquer algorithm is an algorithmic paradigm that breaks down a problem into smaller subproblems (divide), recursively solves each subproblem (conquer), and then combines the result of each subproblem to form the overall solution. The algorithm idea of recursion is fundamental to divide and conquer algorithms because it solves complex problems by dividing input data into smaller instances of the same problem known as subproblems. Such recursion calls terminate when the inputs become so small or so simple that other non-recursive procedures can provide the answers.

A subproblem is a smaller instance of a problem that can be solved independently, and each subproblem can be solved independently of other subproblems by reapplying the same recursive algorithm. To design recursive subproblems, algorithm designers often focus on identifying structural self-similarity in the input data. This process repeats until the input data is small enough to solve directly. Once all the subproblems have been solved, the recursive algorithm reassembles each of these independent solutions to compute the result for the original problem.

---

3 B. Dawson, "Arranging invisible icons in quadratic time," 2021. https://randomascii.wordpress.com/2021/02/16/arranginginvisible-icons-in-quadratic-time/

Earlier, we introduced binary search to find a target within a sorted list as an analogy for finding a term in a dictionary sorted alphabetically. Instead of starting from the beginning of the dictionary and checking each term, as in a sequential search, we could instead start from the middle and look left or right based on where we would expect to find the term in the dictionary. But binary search can also be understood as an example of a divide and conquer algorithm (Figure 3.16).

1. The problem of finding a target within the entire sorted list is broken down (divided) into the subproblem of finding a target within half of the list after comparing the middle element to the target. Half of the list can be ruled out based on this comparison, leaving binary search to find the target within the remaining half.

2. Binary search is repeated on the remaining half of the sorted list (conquer). This process continues recursively until the target is found in the sorted list (or reported as not in the list at all).

3. To solve the original problem of finding a target within the entire sorted list, the result of the subproblem must inform the overall solution. The original call to binary search reports the same result as its subproblem.

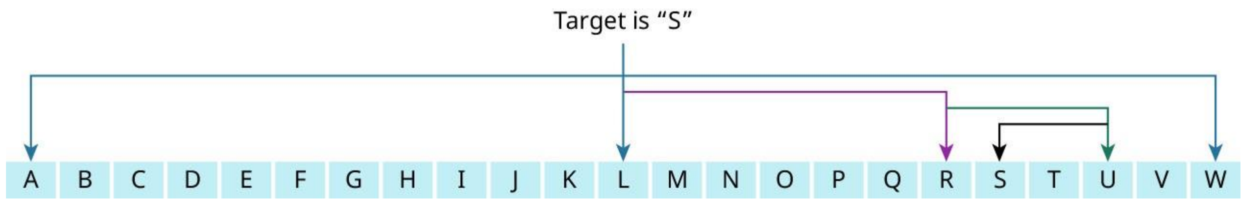


Figure 3.16 Binary search is a divide and conquer algorithm that repeatedly makes one recursive call on half of remaining elements. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Binary search makes a single recursive call on the remaining half of the list as part of the conquer step, but other divide and conquer algorithms make multiple recursive calls to solve their problems. We will also see algorithms that do a lot of work in the final step of combining results more than just reporting the same result as a subproblem.

Given a list of elements in an unknown order, a sorting algorithm should return a new list containing the same elements rearranged into a logical order, such as least to greatest. One canonical divide and conquer algorithm for comparison sorting is called merge sort. The problem of comparison sorting is grounded in the comparison operation. The comparison operation is like a traditional weighing scale that tells whether one element is heavier, lighter, or the same weight as another element, but provides no information about the exact weight or value of the element. Though this might seem like a serious restriction, comparison sorting is actually a very rich problem in computer science—it is perhaps the most deeply studied problem in computer science. Choosing comparison as the fundamental operation is also practical for complex data, where it might be hard (or even impossible) to assign an exact numeric ranking (Figure 3.17).

1. The problem of sorting the list is broken down (divided) into two subproblems: the subproblem of sorting the left half and the subproblem of sorting the right half.
2. Merge sort is repeated to sort each half (conquer). This process continues recursively until the sublists are one element long. To sort a one-element list, the algorithm does not need to do anything, since the elements in the list are already arranged in a logical order.
3. To solve the original problem of sorting the entire list, combine adjacent sorted sublists by merging them while maintaining sorted order. Merging each pair of adjacent sorted sublists repeats to form larger and larger sorted sublists until the entire list is fully sorted.

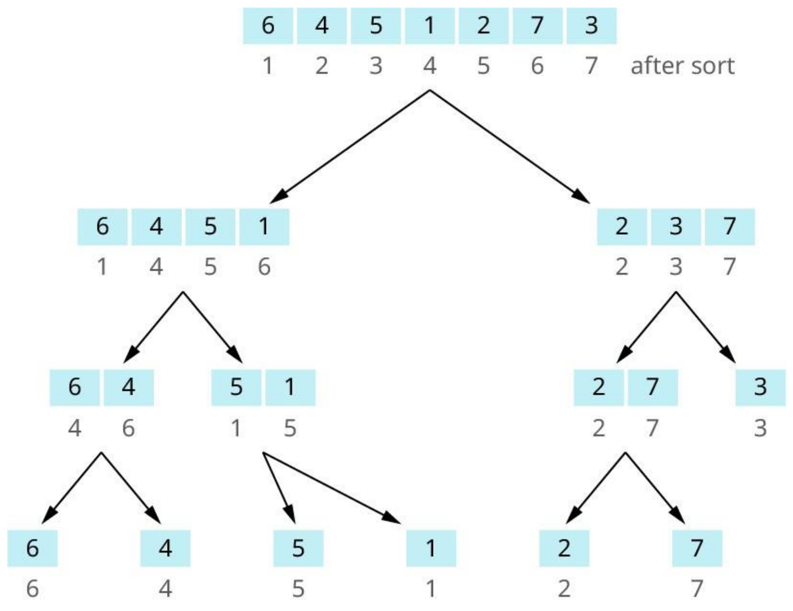


Figure 3.17 Merge sort is a divide and conquer algorithm that repeatedly makes two recursive calls on both halves of the sublist. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Brute-Force Algorithms

Solving a combinatorial problem involves identifying the best candidate solution out of a space of many potential solutions. Each solution to a combinatorial problem is represented as a complex data type. Many applications that involve simulating and comparing different options can be posed as combinatorial problems.

- Route planning in online mapping asks, “Out of all the possible routes from point A to point B, which route is the shortest?” (Shortest paths problem)
- Municipal broadband planning asks, “Out of all the possible ways to connect every real-world address to the Internet, which network of connections is the cheapest to build?” (Minimum spanning trees problem)
- The interval scheduling problem is a combinatorial problem involving a list of scheduled tasks with the goal of finding the largest non-overlapping set of tasks that can be completed.

Rather than searching for a single element, combinatorial problems focus on finding candidate solutions that might be represented as a list of navigation directions, network connections, or task schedules. And unlike sorting, where the output should be a sorted list, combinatorial problems often attempt to quantify or compare the relative quality of solutions in order to determine the best candidate solution out of a space of many potential solutions. Even if our route plan is not the “best” or shortest route, it can still be a valid solution to the problem.

A brute-force algorithm solves combinatorial problems by systematically enumerating all potential solutions in order to identify the best candidate solution. For example, a brute-force algorithm for generating valid credit card numbers might start by considering a credit card number consisting of all zeros, then all zeros except for a one in the last digit place, and so forth, to gradually explore all the possible values for each digit place.

Brute-force algorithms exist for every combinatorial problem, but they are not typically used because of runtime issues. To systematically enumerate all potential solutions, a brute-force algorithm must generate every possible combination of the input data. For example, if a credit card number has sixteen digits and each digit can have any value between zero and nine, then there are 10^{16} potential credit card numbers to enumerate. The combinatorial explosion is the exponential number of solutions to a combinatorial problem that makes brute-force algorithms unusable in practice. Despite continual improvements to how quickly computers can execute programs, exponential time brute-force algorithms are impractical except for very small problem input data.

INDUSTRY SPOTLIGHT

Protein Folding

Proteins are one of the fundamental building blocks of biological life. The 3-D shape of a protein defines what it does and how it works. Given the string of a protein's amino acids, a protein-folding problem asks us to compute the 3-D shape of the resulting protein.

Protein folding has been studied since the first images of their structures were created in 1960. In 1972, Christian Anfinsen won the Nobel Prize in Chemistry for his research into the "protein-folding problem," which involved algorithms to predict the structure of proteins. Because of the huge number of possible formations of proteins, computational studies and algorithms are better able to predict the structures. Given that a brute-force algorithm cannot solve this problem in a reasonable amount of time, computational biologists have developed algorithms that generate high-quality approximations or potential solutions that typically are not quite correct, but run in a more reasonable amount of time.

Modern protein-folding algorithms, such as Google's DeepMind AlphaFold machine-learning algorithm,[4] use machine learning to identify protein-folding patterns from millions of input amino acid sequences and corresponding output 3-D conformations. Rather than utilizing a simple rule for selecting the next element to include in the solution, these machine learning algorithms learn highly complicated rulesets from subtle patterns present in the data.

Improving our understanding of protein folding can lead to massive improvements only in medical health contexts such as drug and vaccine development, but also enable us to design biotechnologies such as enzymes for composting plastic waste, and even limit the impact of global warming by sequestering greenhouse gases from the atmosphere.[5] In 2024, the Nobel Prize Committee recognized the impact of this work by granting the Chemistry prize to Demis Hassabis and John M. Jumper for their work on DeepMind and Alphafold[6] , as well as David Baker for using a similar tool, Rosetta, to create entirely new proteins.

Greedy Algorithms

A greedy algorithm solves combinatorial problems by repeatedly applying a simple rule to select the next element to include in the solution. Unlike brute-force algorithms that solve combinatorial problems by generating all potential solutions, greedy algorithms instead focus on generating just one solution. These algorithms are greedybecause they select the next element to include based on the immediate benefit.

For example, a greedy interval scheduling algorithm might choose to work on the task that takes the least amount of time to complete; in other words, the cheapest way to mark one task as completed (Figure 3.18). If the tasks are scheduled in advance and we can only work on one task at a time, choosing the task that takes the least amount of time to complete might prevent us from completing multiple other (longer) tasks that just so happen to overlap in time. This greedy algorithm does not compute the right output—it finds a solution but not the optimal solution.

---

4 W. D. Haven, “DeepMind’s protein-folding AI has solved a 50-year-old grand challenge of biology,” 2020. https://www.technologyreview.com/2020/11/30/1012712/deepmind-protein-folding-ai-solved-biology-science-drugs-disease/ 5 Google DeepMind, “AlphaFold: A solution to a 50-year-old grand challenge in biology,” 2020." https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology

6 Google DeepMind, “AlphaFold: Demis Hassabis & John Jumper awarded Nobel Prize in Chemistry,” 2024." https://deepmind.google/discover/blog/demis-hassabis-john-jumper-awarded-nobel-prize-in-chemistry/

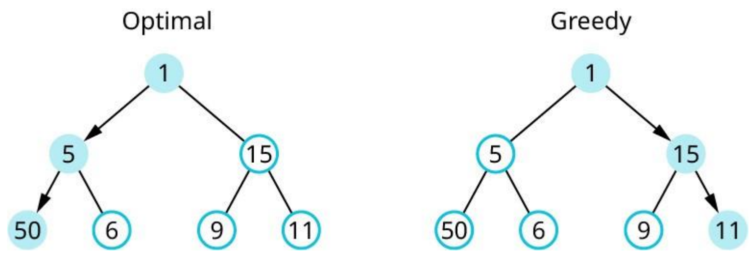


Figure 3.18 Greedy interval scheduling will not work if the simple rule repeatedly selects the shortest interval. (attribution: Copyright

Rice University, OpenStax, under CC BY 4.0 license)

The majority of greedy algorithms do not always compute the best solution. But there are also certain scenarios in which cleverly crafted greedy algorithms guarantee finding optimal solutions. The problems they solve are formulated to ensure that the greedy algorithm never makes a mistake when repeatedly applying the simple rule to select the next element.

Consider the municipal broadband planning problem or, more formally, the minimum spanning tree problem, of finding a lowest-cost way to connect all the vertices in a connected graph to each other. If we want to minimize the sum of the selected edge weights, one idea is to repeatedly select the next edge (connections between vertices) with the lowest weight so long as it extends the reach of the network. Or, in the context of the municipal broadband planning problem, we want to ensure that the next-cheapest connection that we choose to build reaches someone who needs access to the Internet (Figure 3.19).

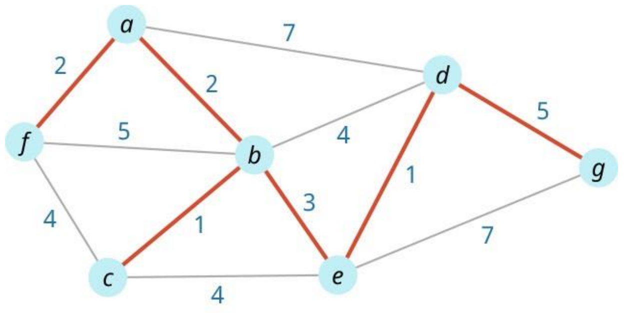


Figure 3.19 Municipal broadband planning can be represented as a minimum spanning trees graph problem where the weight of each edge represents the cost of building a connection between two vertices or places. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

GLOBAL ISSUES IN TECHNOLOGY

Municipal Broadband as a Public Utility

The municipal broadband planning problem is just one component of the larger public policy issue of Internet access. As the Internet and the use of digital platforms becomes the standard mode for communication, many people are viewing municipal broadband as a fundamental public utility and civil right. “Municipal broadband can solve access and affordability problems in areas where private ISPs [Internet service providers] have not upgraded networks to modern speeds, fail to provide service to all residents, and/or charge outrageous rates.”[7]

While a minimum spanning tree algorithm can solve the municipal broadband planning problem, the challenges of deploying municipal broadband for everyone is more political rather than algorithmic. But other algorithms can also directly contribute to the way in which society understands the problem. For example, we might use algorithms to better visualize and understand who currently has access to affordable and reliable high-speed Internet. We can design algorithms to ensure equitable distribution,

7 J. Broken, "Victory for municipal broadband as Wash. state lawmakers end restrictions," 2021. Ars Technica, https://arstechnica.com/tech-policy/2021/04/victory-for-municipal-broadband-as-wash-state-lawmakers-end-restrictions

deployment, and integration of new technologies so that marginalized communities can realize the positive economic benefits first. Or we can reconfigure the minimum spanning trees problem model to take specifically account for expanding network access in an equitable fashion.

Computer scientists have designed algorithms that repeatedly apply this simple rule to find an optimal minimum spanning tree:

- Kruskal's algorithm, a greedy algorithm which sorts the list of edges in the graph by weight.
- Prim's algorithm, a greedy algorithm that maintains a priority queue of vertices in the graph ordered by connecting edge weight.

For most problems, greedy algorithms will not produce the best solution. Instead, algorithm designers typically turn to another algorithmic paradigm called dynamic programming. Still, greedy algorithms provide a useful baseline starting point for understanding problems and designing baseline algorithms for generating potential solutions.

Reduction Algorithms

Algorithm designers spend much of their time modeling problems by selecting and adapting relevant data structures and algorithms to represent the problem and a solution in a computer program. This process of modeling often involves modifying an algorithm design pattern so that it can be applied to the problem. But there is also a different approach to algorithm design that solves problems by changing the input data and output data to fit a preexisting problem. Rather than solve the problem directly, a reduction algorithm solves problems by transforming them into other problems (Figure 3.20).

1. Preprocess: Transform the input data so that it is acceptable to an algorithm meant for the other problem.
2. Apply the algorithm meant for the other problem on the preprocessed data.
3. Postprocess: Transform the output of the algorithm meant for the other problem so that it matches the expected output for the original problem.

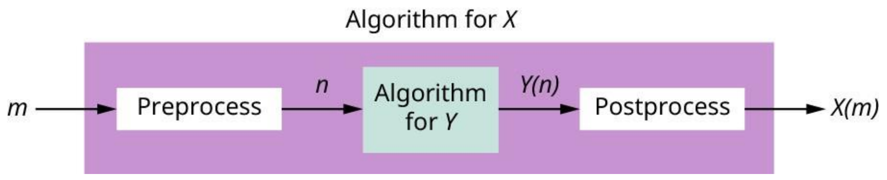


Figure 3.20 A reduction algorithm preprocesses the input data, passes it to another algorithm, and then postprocesses the algorithm’s output to solve the original problem. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license) Consider a slightly different version of the municipal broadband planning problem, where instead of only considering connections (edges), we expand the problem to consider the possibility of installing broadband nodes directly to each address without relying on potentially expensive neighboring connections. That is, all vertices installed with broadband nodes are inter-connected with each other through another broadband network. If we were to run an algorithm for solving the minimum spanning tree problem on this graph directly, then our result would never consider installing broadband nodes directly, since minimum spanning tree algorithms do not consider vertex weights (Figure 3.21).

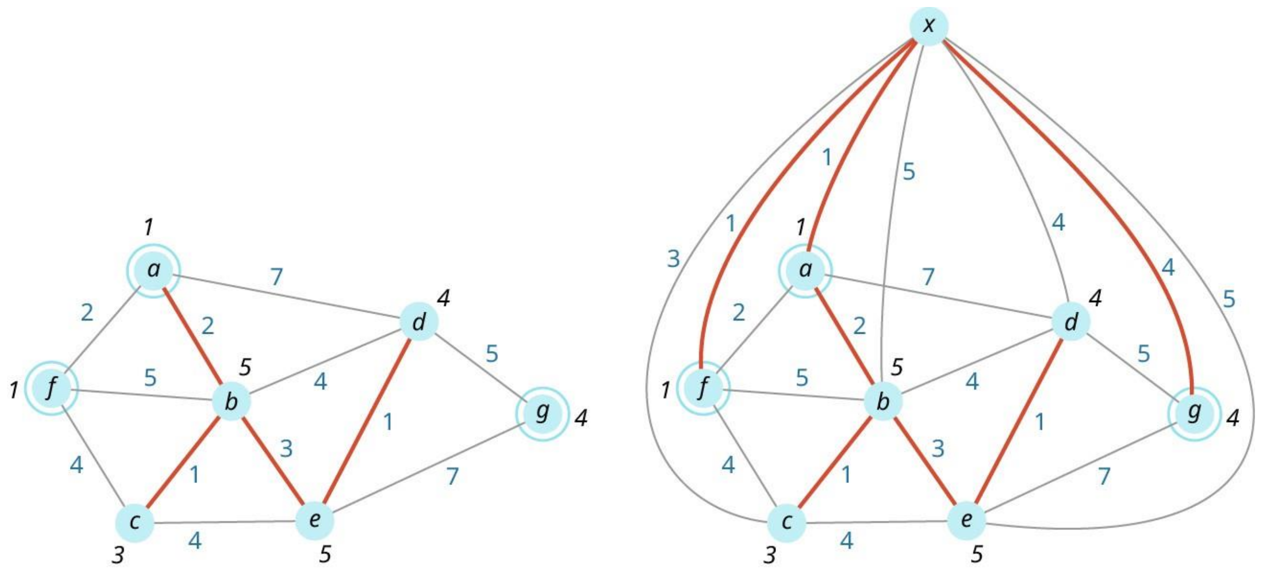


Figure 3.21 The problem of finding a minimum spanning tree in a graph with vertex weights can be reduced to the problem of finding a minimum spanning tree in a graph withoutvertex weights. (attribution: Copyright Rice University, OpenStax, under CC BY

4.0 license)

We say that this more complicated municipal broadband planning problem reduces to the minimum spanning tree problem because we can design a reduction algorithm consisting of preprocessing and postprocessing procedures.

- Preprocess: Introduce an extra vertex that does not represent a real location but connects to every address (vertex) in the city. The edge weight of each connection is the cost of installing the broadband node directly at that location.
- Postprocess: After computing a minimum spanning tree for the preprocessed graph, remove the extra vertex that we added during the processing step. Any edges connected to the extra vertex represent a direct broadband node installation, while other edges between real locations are just the same network connections that we saw earlier.

Reduction algorithms enable algorithm designers to solve problems without having to modify existing algorithms or algorithm design patterns. Reduction algorithms allow algorithm designers to rely on optimized canonical algorithms rather than designing a solution by composing algorithm design patterns, which can lead to performance or correctness bugs. Reduction algorithms also enable computer scientists to make claims about the relative difficulty of a problem. If we know that a problem Areduces to another problem B, Bis as difficult to solve as A.

3.5 Sample Algorithms by Problem

Learning Objectives

By the end of this section, you will be able to:

- Discover algorithms that solve data structure problems
- Understand graph problems and related algorithms

Earlier, we introduced several computing problems, like searching for a target value in a list or implementing an abstract data type (lists, sets, maps, priority queues, graphs). Although every computational problem is unique, these types of problems often share significant similarities with other problems. Computer scientists have identified many canonical problems that represent these common problem templates. Although each of these canonical problems may have many algorithmic solutions, computer scientists have also identified canonical algorithms for solving these problems. In this section, we will introduce canonical problems and survey canonical algorithms for each problem.

Data Structure Problems

Data structure problems are not only useful for implementing data structures, but also as fundamental algorithm design patterns for organizing data to enable efficient solutions to almost every other computing problem.

Searching

Searching is the problem of retrieving a target element from a collection of elements. Searching in a linear data structure, such as an array list, can be done using either sequential search or binary search.

Sequential Search Algorithm

A sequential search algorithm is a searching algorithm that sequentially checks the collection element-byelement for the target. The runtime of sequential search is in O(N) with respect to N, the number of elements in the list (Figure 3.22).

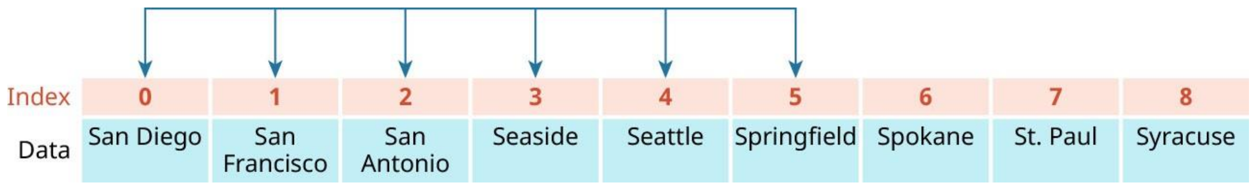


Figure 3.22 Sequential search is a search algorithm that checks the collection element by element to find a target element. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Binary Search Algorithm

A binary search algorithm recursively narrows down the possible locations for the target in the sorted list.

Initially, the algorithm has no information—the target can be anywhere in the entire range of the sorted list.

By comparing the target to the middle element of the range, we can rule-out half of the elements in the range. This process can be repeated until we have found the expected location of the target in the sorted list. The runtime of binary search is in O(log N) with respect to N, the number of elements in the sorted list, so long as indexing is a constant-time operation (see Figure 3.11).

Although the sequential search algorithm works with any collection type such as lists, sets, dictionaries, and priority queues, the binary search algorithm relies on a sorted list with access to elements by index. Consequently, binary search is efficient on array lists that provide constant-time access to the element at any index and inefficient on linked lists, which do not enable constant-time access to elements by index. Binary search relies on the structure of the sorted list to repeatedly rule-out half of the remaining elements.

Binary search trees represent the concept of binary search in a tree data structure by arranging elements in the tree in sorted order from left to right. Ideally, the root node represents the middle element in the sorted tree and each child roughly divides each subtree in half. But, in the worst case, a binary search tree can look exactly like a linked list where each node contains either zero children or one child. Although such a tree still arranges its elements in sorted order from left to right, comparing the target to each node only reduces the size of the problem by one element rather than ruling out half of the remaining elements—the worst-case binary search tree degrades to sequential search. Balanced binary search trees, such as AVL trees, addressed this worst-case scenario by maintaining the AVL property of balance between left and right subtrees.

Sorting

Sorting is the problem of rearranging elements into a logical order, typically from least-valued (smallest) to greatest-valued (largest). Sorting is a fundamental problem not only because of the tasks that it directly solves, but also because it is a foundation for many other algorithms such as the binary search algorithm or Kruskal’s algorithm for the minimum spanning tree problem.

The most common type of sorting algorithm solves the problem of comparison sorting, or sorting a list of elements where elements are not assigned numeric values but rather defined in relation to other elements. For simplicity, the data are typically assumed to be stored in an array list for indexed access, and the sorting algorithm can either return a new sorted list or rearrange the elements in the array list so that they appear in sorted order.

Merge Sort Algorithm

A merge sort algorithm recursively divides the data into sublists until sublists are one element long—which we know are sorted—and then merges adjacent sorted sublists to eventually return the sorted list. The merge operation combines two sorted sublists to produce a new, larger sorted sublist containing all the elements in sorted order. The actual rearranging of elements occurs by repeatedly applying the merge operation on pairs of adjacent sorted sublists, starting with the smallest single-element sublists, to larger two-element sublists, and eventually reaching the two halves of the entire list of elements. The runtime of merge sort is in O(Nlog N) with respect to N, the number of elements (see Figure 3.17).

Quicksort Algorithm

A quicksort algorithm recursively sorts data by applying the binary search tree algorithm design pattern to partition data around pivot elements. Whereas the merge sort algorithm rearranges elements by repeatedly merging sorted sublists after each recursive subproblem, the quicksort algorithm rearranges elements by partitioning data before each recursive subproblem. The partition operation takes a pivot and rearranges the elements into three sections, from left to right: the sublist of all elements less than the pivot, the pivot element, and the sublist of all elements greater than (or equal to) the pivot. Each of the two sublists resulting from the partition operation is a recursive quicksort subproblem; when both of the sublists are sorted, the entire list will be sorted. The runtime of quicksort depends on the choice of each pivot element during the execution of the recursive algorithm, but in practice, for most of the inputs, the runtime is in O(Nlog N) with respect to N, the number of elements (Figure 3.23).

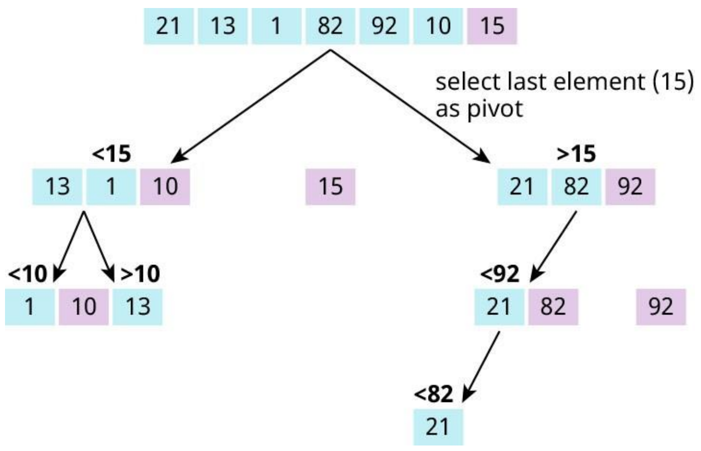


Figure 3.23 Quicksort is a divide and conquer sorting algorithm that sorts elements by recursively partitioning elements around a pivot. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Heapsort Algorithm

A heapsort algorithm adds all elements to a binary heap priority queue data structure using the comparison operation to determine priority value, and returns the sorted list by repeatedly removing from the priority queue element by element. The runtime of heapsort is in O(Nlog N) with respect to N, the number of elements. The logarithmic time factor is due to the time it takes to add or remove each element from the binary heap (Figure 3.24).

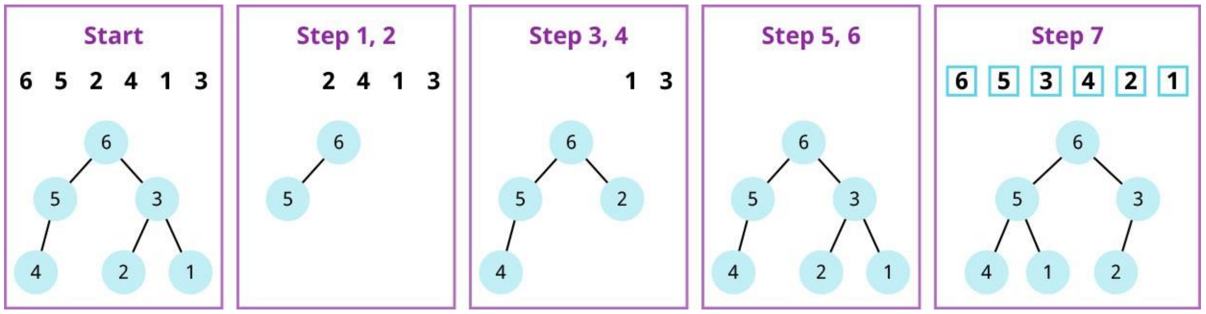


Figure 3.24 Heapsort uses the binary heap data structure to sort elements by adding and then removing all elements from the heap. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Many comparison sorting algorithms share the same O(Nlog N) runtime bound with respect to N, the number of elements. Computer scientists have shown with a combinatorial proof that, in the worst case, a comparison sorting algorithm cannot do better than O(Nlog N) comparisons. It is impossible to design a worst-case O(N) runtime comparison sorting algorithm. In fact, a commonly used version of heapsort (which is also asymptotically faster) is to first build a binary heap (i.e., first arrange the input numbers in an array, then repeatedly call a function to turn the original array into a binary heap; one can show that the running time of this part is linear in N, which is why this is faster than constructing the heap by adding numbers one by one), then remove elements one by one from the end of the heap.

But not all sorting problems are comparison sorting problems. In fact, a commonly used version of heapsort (which is also asymptotically faster) is to first build a binary heap (i.e., first arrange the input numbers in an array, then repeatedly call a function to turn the original array into a binary heap. One can show that the running time of this part is linear in N, which is why this is faster than constructing the heap by adding numbers one by one), then remove elements one by one from the end of the heap as explained previously.

Another type of sorting problem that is not restricted to pairwise comparisons is known as count sorting, or sorting a list of elements by organizing elements into categories and rearranging the categories into a logical order. For example, the problem of sorting a deck of cards can be seen as a count sorting problem if we put the cards into numeric stacks and then rearrange the stacks into a logical order. By changing the assumptions of the problem, count sorting algorithms can run in O(N) time by first assigning each element to its respective category and then unpacking each category so that elements appear in sorted order.

Hashing

Hashing is the problem of designing efficient algorithms which map each object to an integer so that most (if not all) objects will be assigned distinct integers. Although hashing algorithms are often specific to each data type, there exist some general approaches for designing hashing algorithms. The hash value of a simple data type such as an integer can just be the value of the integer itself. The hash value of a string of text can be some mathematical combination of the numeric value of each character in the string. Likewise, the hash value of a collection such as a list can be some combination of the underlying numeric data within each element in the collection.

Hashing has a variety of applications spanning computer systems, database systems, computer security, and searching algorithms. For example, hashing algorithms are often used designing secure systems for protecting stored passwords even after a security breach occurs. In the context of data structure problems, hashing offers a different approach than binary search. Instead of relying on pairwise comparisons to narrow down the expected location of an element in a sorted list in logarithmic time, hashing search algorithms can instead directly index an element by hash value in constant time. If binary search trees implement sets and maps by applying the concept of binary search, a hash table implements sets and maps by applying the concept of hashing (Figure 3.25). Rather than organize elements in sorted order from left to right as in a binary search tree, hash tables store and retrieve elements in an array indexed by hash value.

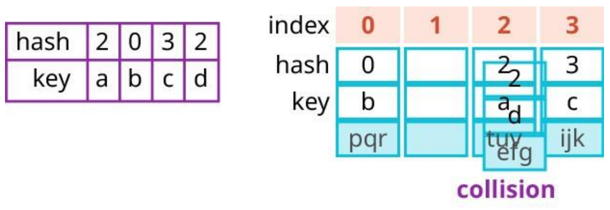


Figure 3.25 Hash tables data structures apply hashing to implement abstract data types such as sets and maps, but must handle collisions between elements that share the same hash value. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Hashing search algorithms are often preferred over binary search algorithms for their runtime benefits, but they come with unique drawbacks. Ideally, different objects would hash to different hash values. But the combinatorial explosion of possibilities for unique strings or collections of complex data means that a collision, or a situation in which multiple objects hash to the same integer index value, is inevitable since integers in a computer can typically only represent a certain, fixed range of integer numbers.

Combinatorial explosion is not only a problem for the design of efficient algorithms, but also the design of efficient data structures too.

Graph Problems

While data structure problems focus primarily on storage and retrieval of elements in a collection, graph problems include a wide variety of computing problems involving the graph data structure. Unlike other data structures, graphs include not only elements (vertices) but also relationships between elements (edges). Graph problems often ask algorithm designers to explore the graph in order to answer questions about elements and the relationships between elements. Traversal

Traversal is the problem of exploring all the vertices in a graph. Graph data structures differ from tree data structures in that there is no explicit root node to begin the traversal and edges can connect back to other parts of the graph. In general, there is no guarantee of hierarchy in a graph. Graph traversal algorithms such as depth-first search and breadth-first search begin at an arbitrary start vertex and explore outwards from the start vertex while keeping track of a set of explored vertices.

Depth-First Search

A depth-first search is a graph traversal algorithm that recursively explores each neighbor, continuing as far possible along each subproblem depth-first (Figure 3.26). Explored vertices are added to a global set to ensure that the algorithm only explores each vertex once. The runtime of depth-first search is in O(|V| + |E|) with respect to |V|, the number of vertices, and |E|, the number of edges.

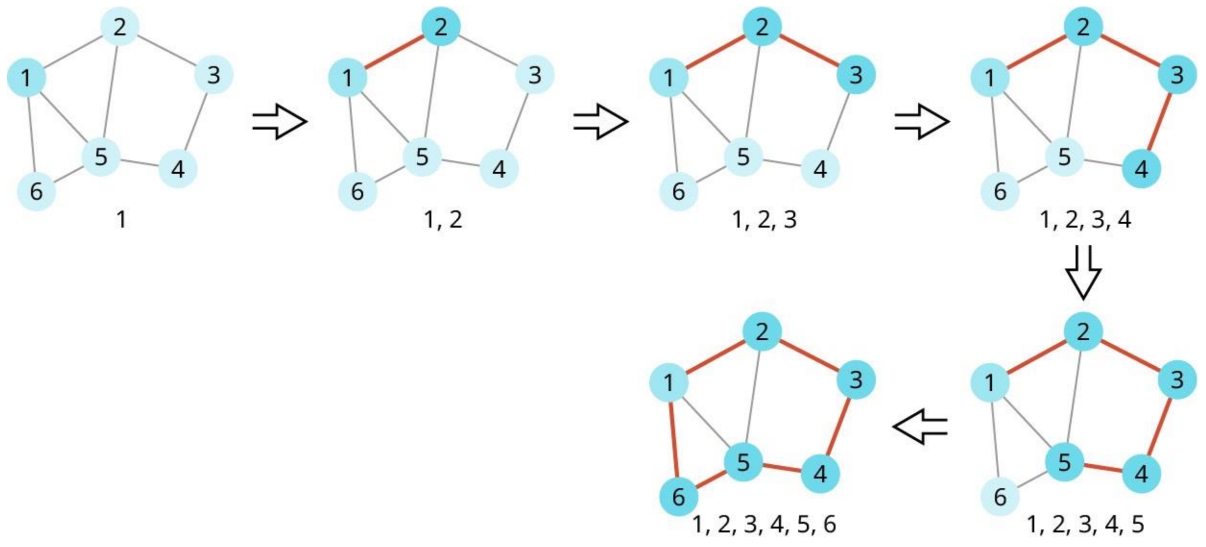


Figure 3.26 Depth-first search is a graph traversal algorithm that continues as far down a path as possible from a start vertex before backtracking. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Breadth-First Search

A breadth-first search iteratively explores each neighbor, expanding the search level-by-level breadth-first (Figure 3.27). Explored vertices are also added to a global set to ensure that the algorithm explores each vertex once and in the correct level-order. The runtime of breadth-first search is also in O(|V| + |E|) with respect to |V|, the number of vertices, and |E|, the number of edges.

Graph traversal algorithms are the foundational algorithm design patterns for most graph processing algorithms. Many algorithms require some amount of exploration, and that exploration typically starts at some vertex and continues processing each reachable vertex at most once. A reachable vertex can be reached if a path or sequence of edges from the start vertex exists. As opposed to depth-first search, breadth-first search has the benefit of exploring vertices closer to the start before exploring vertices further from the start, which can be useful for solving problems such as unweighted shortest paths.

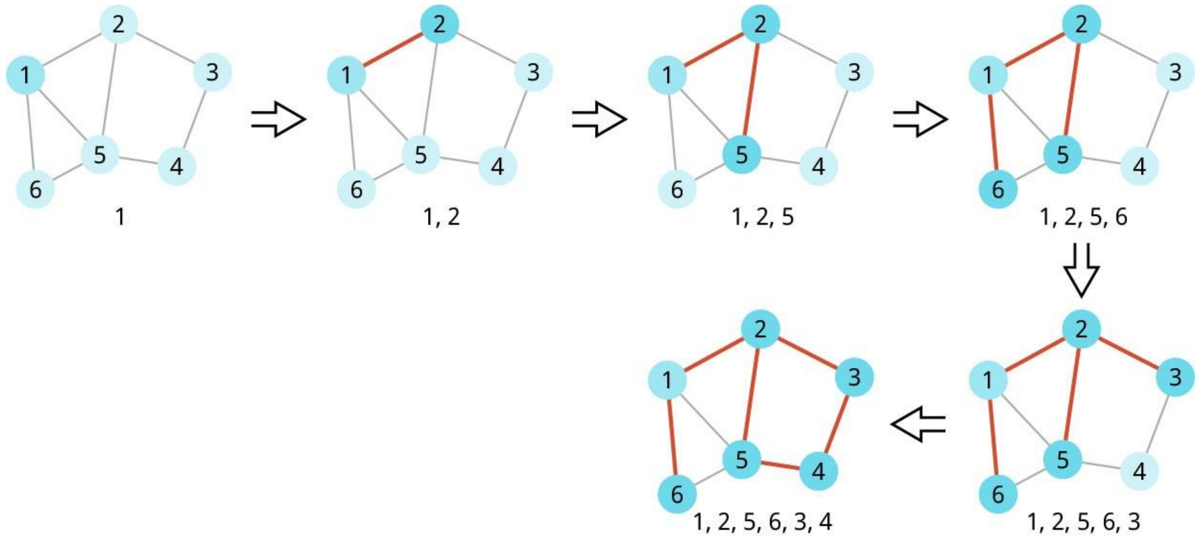


Figure 3.27 Depth-first search is a graph traversal algorithm that continues as far down a path as possible from a start vertex before backtracking. Breadth-first search is a graph traversal algorithm that explores level by level expanding outward from the start vertex. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Minimum Spanning Trees

Minimum spanning trees is the problem of finding a lowest-cost way to connect all the vertices to each other, where costis the sum of the selected edge weights. The two canonical greedy algorithms for finding a minimum spanning tree in a graph are Kruskal’s algorithm and Prim’s algorithm. Both algorithms repeatedly apply the rule of selecting the next lowest-weight edge to an unconnected part of the graph. The output of a minimum spanning tree algorithm is a set of |V| - 1 edges connecting all the vertices in the graph with the least total sum of edge weights, where |V| is the number of vertices.

Kruskal's Algorithm

Kruskal's algorithm begins by considering each vertex as an independent "island," and the goal of the algorithm is to repeatedly connect islands by selecting the lowest-cost edges. A specialized data structure (called disjoint sets) is typically used to keep track of the independent islands. The runtime of Kruskal's algorithm is in O(|E| log |E|) with respect to |E|, the number of edges (Figure 3.28).

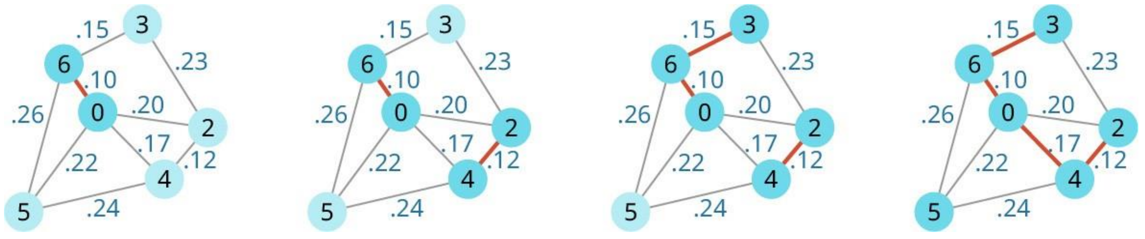


Figure 3.28 Kruskal's algorithm repeatedly selects the next lightest edge that connects two independent "islands." (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Prim's Algorithm

Prim's algorithm grows a minimum spanning tree one edge at a time by selecting the lowest-weight edge to an unexplored vertex. The runtime of Prim's algorithm is in O(|E| log |V| + |V| log |V|) with respect to |V|, the number of vertices, and |E|, the number of edges (Figure 3.29).

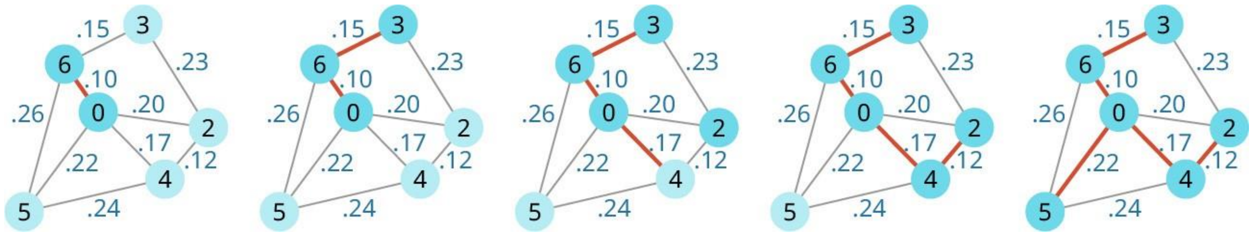


Figure 3.29 Prim's algorithm expands outward from the start vertex by repeatedly selecting the next lightest edge to an unreached vertex. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Shortest Paths

The output of a shortest paths algorithm is a shortest paths tree, the lowest-cost way to get from one vertex to every other vertex in a graph (Figure 3.30). The unweighted shortest path is the problem of finding the shortest paths in terms of the number of edges. Given a start vertex, breadth-first search can compute the unweighted shortest paths tree from the start vertex to every other reachable vertex in the graph by maintaining a map data structure of the path used to reach each vertex.

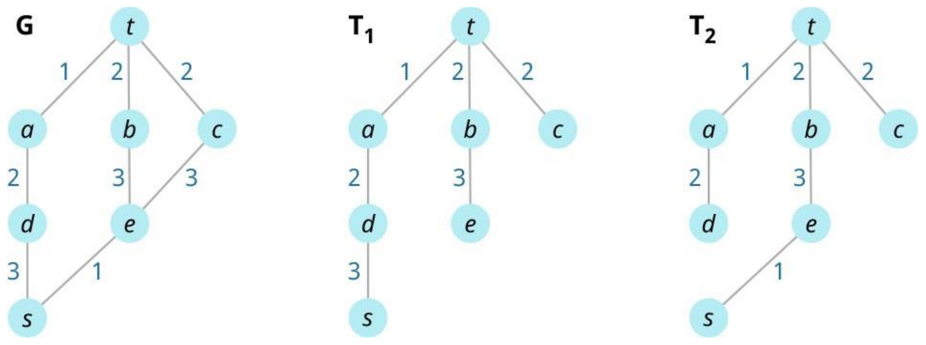


Figure 3.30 Three shortest paths trees of the lowest-cost way to get from the start vertex to every other vertex in the graph are shown. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Weighted Shortest Path

A weighted shortest path is the problem of finding the shortest paths in terms of the sum of the edge weights. Earlier, we introduced Prim's algorithm, a minimum spanning tree algorithm that maintains a priority queue of edges in the graph ordered by weight and repeatedly selects the next lowest-cost edge to an unconnected part of the graph.

THINK IT THROUGH

Weighted Shortest Paths for Navigation Directions

One of the most direct applications of the shortest paths problem is to provide recommended routes for navigation directions in real-world mapping. Many applications use distance as the metric for edge weight, so the shortest path between two points represents the real-world route with the smallest distance between the two places.

What does a distance metric not consider when providing a recommended route? What values are centered and emphasized by even using a shortest paths algorithm for recommending routes?

Dijkstra's Algorithm

Dijkstra's algorithm maintains a priority queue of vertices in the graph ordered by distance from the start and repeatedly selects the next shortest path to an unconnected part of the graph. Dijkstra's algorithm is almost identical to Prim's algorithm except processing shortest paths (sequences of edges) rather than individual edges. Dijkstra's algorithm grows a shortest paths tree one shortest path at a time by selecting the next shortest path to an unexplored vertex. The runtime of Dijkstra's algorithm is in $O(|E| \log |V| + |V| \log |V|)$ with respect to $|V|$, the number of vertices, and $|E|$, the number of edges (Figure 3.31).

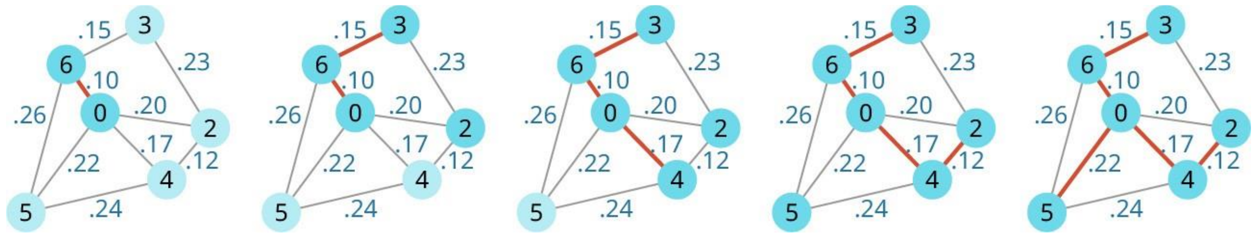


Figure 3.31 Dijkstra’s algorithm expands outward from the start vertex by repeatedly selecting the next lowest-cost path to an unreached vertex. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

3.6 Computer Science Theory

Learning Objectives

By the end of this section, you will be able to:

- Understand the models and limits of computing
- Relate Turing machines to algorithms
- Describe complexity classes
- Interpret NP-completeness
- Differentiate between P and NP

Throughout this chapter, we have introduced several techniques and canonical case studies for the design and analysis of algorithms—oftentimes focusing on the ideas and details behind individual algorithms. But the study of algorithms is more than just the study of individual algorithms, algorithm design, or even algorithm analysis.

Models of Computation

Computers include basic algorithms for solving problems like adding, subtracting, or comparing two numbers. Computers, owing to their roots in calculators, are optimized to solve these problems; these basic algorithms are constant-time operations. What programming offers is the ability to define our own algorithms that can be used to solve more complex problems, such as searching, sorting, and hashing. Unfortunately, these programmed algorithms are not as fast as basic operations. We have even seen certain problems deal with a combinatorial explosion in the number of potential solutions. For many of these problems, the best-known algorithms do not do much better than brute-force, which takes exponential time.

In the bigger picture, computer science engages the central question of how humans can encode intelligence. Our discussion of algorithm design grounded the activity in problem modeling, the process of encoding a complex real-world phenomenon or problem in a more abstract or simple form. How is problem modeling constrained by the model of computation, or the rules of the underlying computer that executes an algorithm? Why are certain problems challenging for computers to execute?

Combinatorial explosion poses a problem for computer algorithms because our model of computation assumes computers only have a single thread of execution and only execute one basic operation on each step. If we overturn some part of this assumption, either by creating computers with multiple processors or by creating more sophisticated operations, then it might be possible to deal with combinatorial explosion. Almost all of today's computer hardware, ranging from massive supercomputers to handheld smartphones, rely at least to some degree on expanding the model of computation to compute solutions to problems more efficiently. Even so, much of today's computer hardware still relies on the same fundamental programming assumptions: that there are variables to represent data and arithmetic or logical operations.

Turing Machines

In the 1800s, Charles Babbage imagined a mechanical machine—the Analytical Engine—that could automatically calculate mathematical formulas. Ada Lovelace then extrapolated that the Analytical Engine could solve more general algorithms by using loops to repeat processes and variables to represent data. Lovelace's vision of algorithms represented a synthesis between human intuition and mathematical reasoning. In the mid-1900s, Lovelace's ideas inspired Alan Turing to imagine a more general notion of algorithms and machines that could run those algorithms. A Turing machine is an abstract model of computation for executing any computer algorithm. A Turing machine describes computers in terms of three key ideas:

1. a memory bank for storing data.
2. an instruction table, where each instruction can either:
   a. store a value to the memory bank.
   b. retrieve a value from the memory bank.
   c. perform a basic operation on a value.
   d. set which instruction will be executed next by modifying the program counter.
3. a program counter that keeps track of the current instruction in the instruction table.

A Turing machine executes a computer algorithm by following each instruction specified by the program counter. An algorithm can use these basic operations to compute the sum of 1 and 1.

1. Store the value 1 to address A in the memory bank. 2. Store the value 1 to address B in the memory bank.

3. Add the values at addresses A and B and then store the result at address A.

What makes computers useful is not just the fact that they can calculate numbers, but that they can encode logic in the instructions. Instead of just computing the sum of 1 and 1, this program continues adding 1 to a growing sum stored at address A.

1. Store the value 1 to address A in the memory bank. 2. Store the value 1 to address B in the memory bank.

3. Add the values at addresses A and B and then store the result at address A.

4. Set the program counter to execute step 3 next.

The Turing machine abstract model of computation assumes a single thread of execution following each instruction in an algorithm. Although today's computers are much more efficient than the first computers that realized the Turing machine, most computers still rely on the same fundamental assumptions about how to execute algorithms. The O(N)-time sequential search algorithm, though it might execute 1,000 times faster on today's computers, still grows linearly with respect to the size of the input. An O(2^N)-time brute-force algorithm, though it might execute 1,000 times faster on today's computers, still grows exponentially with respect to the size of the input. Even as computers become faster over time, inefficient algorithms still cannot be used to solve any problems larger than a few thousand elements.

Complexity Classes

One subfield of computer science is theoretical computer science, which studies models of computation, their application to algorithms, and the complexity of problems. The complexity of a problem is the complexity (in terms of time or memory resources required) of the most efficient algorithms for solving the problem. Theoretical computer scientists are interested in understanding the difficulty of a problem in terms of time complexity, space complexity, and some other complexity measures.

In this chapter, we have focused on solving problems known to have polynomial time algorithms. Searching, sorting, hashing, traversal, minimum spanning trees, or shortest paths are all examples of problems in the polynomial (P) time complexity class because they are all problems that have runtimes with a polynomial expression such as O(1), O(log N), O(N), O(Nlog N), O(N^2), O(N^3). In general, these problems are considered tractable because computers can solve them in a reasonable amount of time. But there are many problems that are considered intractable because they do not have efficient, polynomial-time algorithms.

The nondeterministic polynomial (NP) time complexity class refers to all problems that can be solved in polynomial time by a nondeterministic algorithm. A nondeterministic algorithm is a special kind of Turing machine, which at each step can nondeterministically choose which instruction to execute, and is considered to successfully find a solution if any combination of these nondeterministic choices eventually lead to a correct solution. In other words, in contrast to a deterministic algorithm, such as a greedy algorithm, which must repeatedly apply a simple rule to deterministically select the next element in a solution, a nondeterministic algorithm is able to simultaneously explore all the possible choices. We do not yet have computers that can execute nondeterministic algorithms, but if we did, then we would be able to efficiently solve any combinatorial problem by relying on the special power of nondeterminism.

Technically, all P problems are also NP problems because we already have deterministic algorithms for solving them and therefore do not need to rely on the special power of nondeterminism. For example, Dijkstra's algorithm provides a deterministic polynomial-time solution to the shortest paths problem by building up a shortest paths tree from the start vertex outward. This application of the greedy algorithmic paradigm relies on the structure of the shortest paths tree, since the shortest path to a point further away from the start must build on the shortest path to a point closer to the start.

NP-complete Problems

NP-complete refers to all the hardest NP problems—the combinatorial problems for which we do not have deterministic polynomial-time algorithms. More precisely, a problem PI is said to be NP-complete if PI is in NP and for every problem in NP, there is a reduction that reduces the problem to PI. For example, a longest path, or the problem of finding the highest-cost way to get from one vertex to another without repeating vertices, is an NP-complete problem opposite to shortest paths (Figure 3.32). What makes longest paths so much harder to solve than shortest paths? For one, there is no underlying structure to the solution that we can use to repeatedly apply a simple rule as in a greedy algorithm. With the shortest paths problem, we could rely on the

shortest paths treeto inform the solution. But in longest paths, the goal is to wanderaround the graph. The

longest path between any two vertices will probably involve traversing as many edges as possible to maximize distance, visiting many vertices along the way. For some graphs, the longest paths might even visit all the vertices in the graph. In this situation, the longest paths do not form a tree and instead involve ordering all the vertices in the graph for each longest path. Identifying the correct longest path then requires listing out all the possible paths in the graph—a combinatorial explosion in the combinations of edges and vertices that can be selected to form a solution.

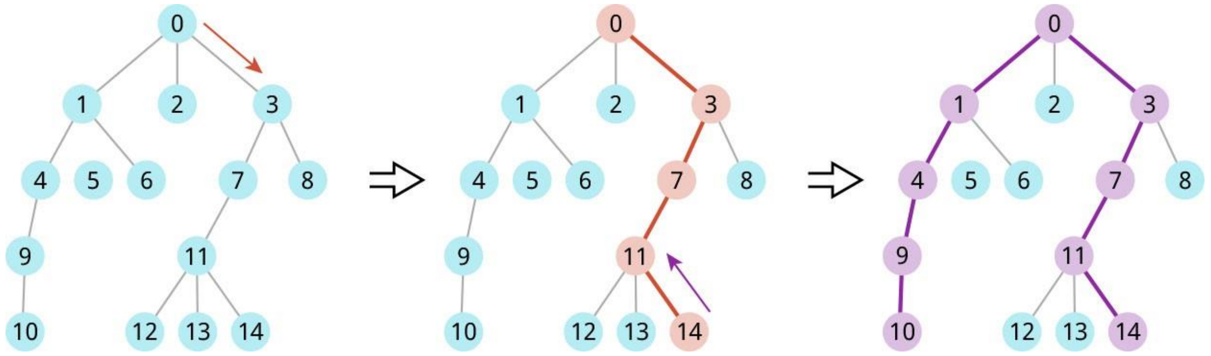


Figure 3.32 The longest path in a graph maximizes the distance, which often (but not always) involves visiting many vertices along the way. (attribution: Copyright Rice University, OpenStax, under CC BY 4.0 license)

Related to the problem of longest paths is the traveling salesperson problem (TSP), which is the problem of, given the path cost of every pair of vertices, finding the lowest-cost tour, or path from a start vertex visiting every vertex in the graph once including a return to the start vertex. Compared to the TSP, which is finding the lowest-cost tour, the longest paths problem is like finding the highest-cost tour. What makes both these problems difficult is that we do not have a simple rule for selecting the next element to include in the solution. Unless we have the special power of nondeterminism, it is hard to tell from the beginning which edge is the right one to include in the final solution. Applying a simple rule like, "Select the edge with the lowest cost," might not necessarily lead to the overall lowest-cost tour. Even though this simple rule worked for the problem of minimum spanning trees, the additional restriction of a tour rather than a tree makes the TSP a much harder problem to solve efficiently. Although we have efficient algorithms for shortest paths, we do not have efficient algorithms for the shortest tour (TSP).

INDUSTRY SPOTLIGHT

Delivery Logistics

Companies such as Amazon, FedEx, UPS, and others that rely on logistics to deliver goods to various locations seek to optimize the sequence of stops to save costs. The only way to achieve this would be to rely on an optimal algorithm for the traveling salesperson problem. The TSP aims to find the minimum distance tour that visits all the vertices such that no vertex is visited more than once. But this is not a perfect match for real-world delivery logistics. Fuel or battery efficiency, for example, is not just about distance traveled, but also the speed of travel, the time spent idling, and even the way that the route is organized. In the United States, where vehicles drive on the right side of the road, safety can be improved by reducing the number of left-hand turns across the divider. Drivers might also want to take breaks during a trip. Modeling all these factors requires carefully formulating the problem and considering the limits of TSP.

How do we know if a problem is NP-complete? Earlier, we introduced reduction as an algorithm paradigm that is not only useful for solving problems, but also for relating the difficulty of problems. A reduction from a difficult problem A to another problem B proves that B is as difficult as A. It turns out that all NP-complete problems can be reduced to all the others, so an algorithm for solving anyNP-complete problem solves every NP-complete problem.

LINK TO LEARNING

The longest paths problem and the traveling salesperson problem are just two examples of NP-complete problems. Visit A Graph of NP-Complete Reductions (https://openstax.org/r/76NPCompReuct) to visualize many NP-complete problems and their relationships to each other. For example, the longest paths problem (LPT) and the traveling salesperson problem (TS) both reduce to Hamiltonian paths (HP). In turn, Hamiltonian paths (HP) reduce to vertex cover (VC) which reduces to 3-satisfiability (3-SAT).

P versus NP

Longest paths and TSP are just two among thousands of NP-complete problems for which we do not have efficient algorithms. The question of P versus NP asks whether it is possible to design a deterministic polynomial-time algorithm for solving any—and therefore all—of NP-complete problems. There are two possible answers to the question of P versus NP:

- If P = NP, then there is a deterministic polynomial-time algorithm for solving all NP-complete problems.
- If P ≠ NP, then there are NP-complete problems that cannot be solved with a deterministic polynomialtime algorithm.

Most theoretical computer scientists believe that P ≠ NP; in other words, it is impossibleto design a deterministic polynomial-time algorithm for longest paths, TSP, or any other NP-complete problem. However, they do not have any definite proof that P = NP or P ≠ NP. An efficient algorithm for any one NP-complete problem would not only directly solve routing and logistics problems but would also enable massive advancements in drug discovery through scientific simulation, for instance. It would also break essentially all modern Internet security and password systems—among thousands of other problems.

Chapter Review

Key Terms

abstract data type (ADT) consists of all data structures that share common functionality abstraction process of simplifying a concept in order to represent it in a computer adjacent in a graph abstract data type, the relationship between two vertices connected by an edge algorithm analysis study of the results produced by the outputs as well as how the algorithm produces those outputs

algorithm design pattern solution to well-known computing problems algorithmic paradigm common concept and ideas behind algorithm design patterns

algorithmic problem solvingrefers to a particular set of approaches and methods for designing algorithms that draws on computing’s historical connections to the study of mathematical problem solving

array list data structure that stores elements next to each other in memory

asymptotic analysis evaluates the time that an algorithm takes to produce a result as the size of the input increases

asymptotic notation mathematical notation that formally defines the order of growth

AVL tree balanced binary search tree data structure often used to implement sets or maps that organizes elements according to the AVL tree property

AVL tree property requires the left and right subtrees to be balanced at every node in the tree balanced binary search tree introduces additional properties that ensure that the tree will never enter a worst-case situation by reorganizing elements to maintain balance

Big O notation most common type of asymptotic notation in computer science used to measure worst case complexity

binary heap binary tree data structure used to implement priority queues that organizes elements according to the heap property

binary logarithm tells how many times a large number needs to be divided by 2 until it reaches 1 binary search algorithm recursively narrows down the possible locations for the target in the sorted list binary search tree tree data structure often used to implement sets and maps that organizes elements according to the binary tree property and the search tree property

binary tree property requires that each node can have either zero, one, or two children breadth-first search iteratively explores each neighbor, expanding the search level-by-level breadth-first brute-force algorithm solves combinatorial problems by systematically enumerating all potential solutions in order to identify the best candidate solution

canonical algorithm well-known algorithm

case analysis way to account for variation in runtime based on factors other than the size of the problem child node descendant of another node

collision situation where multiple objects hash to the same integer index value

combinatorial explosion exponential number of solutions to a combinatorial problem that makes bruteforce algorithms unusable in practice

combinatorial problem involves identifying the best candidate solution out of a space of many potential solutions

comparison sorting sorting of a list of elements where elements are not assigned numeric values but rather defined in relation to other elements

complexity condition based on the degree of computational resources that an algorithm consumes during its execution in relation to the size of the input

compression problem of representing information using less data storage constant type of order of growth that does not take more resources as the size of the problem increases correctness whether the outputs produced by an algorithm match the expected or desired results across the range of possible inputs

cost model characterization of runtime in terms of more abstract operations such as the number of repetitions

count sorting sorting a list of elements by organizing elements into categories and rearranging the categories into a logical order

cryptography problem of masking or obfuscating text to make it unintelligible data structure complex data type with specific representation and specific functionality data structure problem computational problem involving the storage and retrieval of elements for implementing abstract data types such as lists, sets, maps, and priority queues

data type determines how computers process data by defining the possible values for data and the possible functionality or operations on that data

depth-first search graph traversal algorithm that recursively explores each neighbor, continuing as far possible along each subproblem depth-first

Dijkstra’s algorithm maintains a priority queue of vertices in the graph ordered by distance from the start and repeatedly selects the next shortest path to an unconnected part of the graph divide and conquer algorithm algorithmic paradigm that breaks down a problem into smaller subproblems (divide), recursively solves each subproblem (conquer), and then combines the result of each subproblem in order to inform the overall solution

edge relationship between vertices or nodes element individual data point

experimental analysis evaluates an algorithm’s runtime by recording how long it takes to run a program implementation of it

functionality operations such as adding, retrieving, and removing elements graph represents binary relations among collection of entities, specifically vertices and edges graph problem computational problem involving graphs that represent relationships between data greedy algorithm solves combinatorial problems by repeatedly applying a simple rule to select the next element to include in the solution

hash table implements sets and maps by applying the concept of hashing hashing problem of assigning a meaningful integer index for each object

heap property requires that the priority value of each node in the heap is greater than or equal to the priority values of its children

heapsort algorithm adds all elements to a binary heap priority queue data structure using the comparison operation to determine priority value and returns the sorted list by repeatedly removing from the priority queue element-by-element

index position or address for an element in a list

interval scheduling problem combinatorial problem involving a list of scheduled tasks with the goal of finding the largest non-overlapping set of tasks that can be completed

intractable problems that do not have efficient, polynomial-time algorithms

Kruskal’s algorithm greedy algorithm that sorts the list of edges in the graph by weight leaf node node at the bottom of a tree that has no children

linear type of order of growth where the resources required to run the algorithm increases at about the same rate as the size of the problem increases

linear data structure category of data structures where elements are ordered in a line

linked list data structure that does not necessarily store elements next to each other and instead works by maintaining, for each element, a link to the next element in the list

list ordered sequence of elements and allows adding, retrieving, and removing elements from any position in the list logarithm tells how many times a large number needs to be divided by a small number until it reaches 1 longest path problem of finding the highest-cost way to get from one vertex to another without repeating vertices map represents unordered associations between key-value pairs of elements, where each key can only appear once in the map

matching problem of searching for a text pattern within a document merge sort canonical divide and conquer algorithm for comparison sorting minimum spanning tree problem of finding a lowest-cost way to connect all the vertices to each other model of computation rules of the underlying computer that is ultimately responsible for executing the algorithm

node represents an element in a tree or graph

nondeterministic algorithm special kind of Turing machine, which at each step can non-deterministically choose which instruction to execute and is considered to successfully find a solution if any combination of these nondeterministic choices leads to a correct solution

nondeterministic polynomial (NP) time complexity classall problems that can be solved in polynomial time by a nondeterministic algorithm

NP-complete all the hardest NP problems—the combinatorial problems for which we do nothave deterministic polynomial-time algorithms

order of growth geometric prediction of an algorithm's time or space complexity as a function of the size of the problem

perfectly balanced for every node in the binary search tree, its left and right subtrees contain the same number of elements

polynomial (P) time complexity class all problems that have runtimes described with a polynomial expression such as O(1), O(log N), O(N), O(Nlog N), O(N^2), O(N^3)

Prim's algorithm greedy algorithm that maintains a priority queue of vertices in the graph ordered by connecting edge weight

priority queuerepresents a collection of elements where each element has an associated priority value problem task with specific input data and output data corresponding to each input problem model simplified, abstract representation of a more complex real-world problem program realization or implementation of an algorithm written in a formal programming language quicksort algorithm recursively sorts data by applying the binary search tree algorithm design pattern to partition data around pivot elements

reachable vertex vertex that can be reached if a path or sequence of edges from the start vertex exists

reduction algorithm solves problems by transforming them into other problems
representation particular way of organizing a collection of elements root node
node at the top of the tree

runtime analysis study of how much time it takes to run an algorithm

search tree property requires that elements in the tree are organized least-to-greatest from left-to-right searching problem of retrieving a target element from a collection of elements

sequential search algorithm searching algorithm that sequentially checks the collection element-byelement for the target

set represents an unordered collection of unique elements and allows adding, retrieving, and removing elements from the set

shortest path problem of finding a lowest-cost way to get from one vertex to another

shortest paths tree output of the shortest paths problem, the lowest-cost way to get from one vertex to every other reachable vertex in a graph

sortingproblem of rearranging elements into a logical order

space complexity formal measure of how much memory an algorithm requires during its execution as it relates to the size of the problem

step basic operation in the computer, such as looking up a single value, adding two values, or comparing two values

string problem computational problem involving text or information represented as a sequence of characters subproblem smaller instance of a problem that can be solved independently

time complexity formal measure of how much time an algorithm requires during execution as it relates to the size of the problem

tractable problems that computers can solve in a reasonable amount of time

traveling salesperson problem (TSP) problem of, given the path cost of every pair of vertices, finding the lowest-cost tour, or path from a start vertex visiting every vertex in the graph once including a return to the start vertex

traversal problem of exploring all the vertices in a graph

tree hierarchical data structure

Turing machine abstract model of computation for executing any computer algorithm unweighted shortest path problem of finding the shortest paths in terms of the number of edges vertex represents an element in a graph or special type of it such as a tree

weighted shortest path problem of finding the shortest paths in terms of the sum of the edge weights

Summary

3.1 Introduction to Data Structures and Algorithms

- Data structures represent complex data types for solving real-world problems. Data structures combine specific data representations with specific functionality.
- Abstract data types categorize data structures according to their functionality and ignore differences in data representation. Abstract data types include lists, sets, maps, priority queues, and graphs.
- To select an appropriate data structure, first select an abstract data type according to the problem requirements. Then, select an appropriate data structure implementation for the abstract data type.
- Linear data structures organize elements in a line, ideal for implementing the list abstract data type. Linear data structures include array lists and linked lists.
- Linear data structures can implement any abstract data type. The study of data structures in general focuses on opportunities to improve efficiency (in terms of execution time or memory usage) over linear data structures.
- Tree data structures organize elements in a hierarchy of levels defined by parent-child relationships. Trees are defined with a root node at the top of the tree, parent-child relationships between each level, and leaf nodes at the bottom of the tree.
- Binary search trees require that elements in the tree are organized least-to-greatest from left-to-right. Binary search trees are often used to implement the set and map abstract data types.
- Balanced binary search trees and binary heaps represent two approaches for avoiding the worst-case situation with binary search trees. Binary heaps are often used to implement the priority queue abstract data type.
- Graph data structures focus on explicitly modeling the relationships between elements. Graphs afford access not only to elements, but also to the relationships between elements.

3.2 Algorithm Design and Discovery

- Just like how many data structures can represent the same abstract data type, many algorithms exist to solve the same problem. In algorithmic problem-solving, computer scientists solve formal problems with specific input data and output data that correspond to each input.

- Modeling is the process of representing a complex phenomenon such as a real-world problem as a formal problem. Modeling is about abstraction: the simplification or erasure of details so that the problem can be solved by a computer.

- Historically, the model of computation emphasized specialized algorithms operating on a modest model of the underlying phenomenon. Modeling is a violent but also necessary act in order to simplify the problem so that it can be solved by a computer.

- Searching is the problem of retrieving a target element from a collection of many elements. Sequential search and binary search are two algorithms for solving the search problem.

- To solve real-world problems, computer scientists compose, modify, and apply algorithm design patterns, such as search algorithms.

- Algorithm analysis is the study of the outputs produced by an algorithm as well as how the algorithm produces those outputs.

- Correctness considers whether the outputs produced by an algorithm match the expected or desired results across the range of possible inputs. Correctness is defined as a match between the algorithm and the model of the problem, not between the algorithm and the real-world.

- Correctness is complicated by the complexity of social relationships, power, and inequity in the real-world. Since algorithms automate processes and operate in existing power structures, they are likely to reproduce and amplify social injustice.

- In addition to correctness, computer scientists are also interested in complexity, or measuring the computational resources that an algorithm consumes during its execution in relation to the size of the input.

3.3 Formal Properties of Algorithms

- Runtime analysis is a study of how much time it takes to run an algorithm. Experimental analysis is a runtime analysis technique that involves evaluating an algorithm's runtime by recording how long it takes to run a program implementation of it.

- Time complexity is the formal measure of how much time an algorithm requires during execution as it relates to the size of the problem. The goal of time complexity analysis is to produce a simple and easy-tocompare characterization of the runtime of an algorithm as it relates to the size of the problem.

- Space complexity is the formal measure of how much memory an algorithm requires during execution as it relates to the size of the problem.

- Steps in time complexity analysis are to identify a metric for representing the size of the problem; to model the number of steps needed to execute the algorithm; and to formalize the model using either precise English or asymptotic notation to define the order of growth. Big O notation is the most common type of asymptotic notation in computer science.

- Differences in orders of growth are massive: as the input size grows, the difference between orders of growth becomes more and more vast. For problems dealing with just 1,000 elements, the time it would take to run an exponential-time algorithm on that problem exceeds the current age of the universe—whereas that same-size problem running on the same computer would take just 1 second on a quadratic-time algorithm.

- In practice, across applications working with large amounts of data, $O(N^2)$ is often considered the limit for real-world algorithms. For algorithms that need to run frequently on large amounts of data, algorithm designers target O(N), O(log N), or O(1).

<u>3.4 Algorithmic Paradigms</u>

- Algorithmic paradigms are the common concepts and ideas behind algorithm design patterns, such as divide and conquer algorithms, brute-force algorithms, greedy algorithms, and reduction algorithms.

- Divide and conquer algorithms break down a problem into smaller subproblems (divide), recursively solve each subproblem (conquer), and then combine the result of each subproblem to inform the overall solution. Recursion is an algorithm idea fundamental to divide and conquer algorithms that solves complex problems by dividing input data into smaller, independent instances of the same problem known as subproblems.

- Binary search is an example of divide and conquer algorithm with a single recursive subproblem. Merge sort is an example of a divide and conquer algorithm with two recursive subproblems.

- Brute-force algorithms solve combinatorial problems by systematically enumerating all potential solutions in order to identify the best candidate solution. Combinatorial problems identify the best candidate solution out of a space of many potential solutions.

- Brute-force algorithms exist for every combinatorial problem, but they are not typically used in practice because of long run time issues. To enumerate all potential solutions, a brute-force algorithm must generate every possible combination of the input data.

- Greedy algorithms solve combinatorial problems by repeatedly applying a simple rule to select the next element to include in the solution. Unlike brute-force algorithms that solve combinatorial problems by generating all potential solutions, greedy algorithms instead focus on generating just one solution.
- Greedy algorithms are not always guaranteed to compute the best solution depending on the assumptions and goals of the problem. A greedy algorithm for the interval scheduling problem will not compute the correct result if we choose to complete the shortest tasks.
- Kruskal's algorithm and Prim's algorithm are two examples of greedy algorithms for the minimum spanning trees problem. These algorithms are a rare example of a greedy algorithm that is guaranteed to compute the correct result.
- Reduction algorithms solve problems by transforming them into other problems. In other words, reduction algorithms delegate most of the work of solving the problem to another algorithm meant for a different problem.
- Reduction algorithms allow algorithm designers to rely on optimized canonical algorithms rather than designing a solution by composing algorithm design patterns, which can lead to performance or correctness bugs. Reduction algorithms also enable computer scientists to make claims about the relative difficulty of a problem.

3.5 Sample Algorithms by Problem

- Data structure problems focus on the storage and retrieval of elements for implementing abstract data types such as lists, sets, maps, and priority queues. Data structure problems include sorting, searching, and hashing.
- Searching is the problem of retrieving a target element from a collection of elements. Searching in a linear data structure such as an array list can be done using either sequential search or binary search.
- Sorting is the problem of rearranging elements into a logical order, typically from least-valued (smallest) to greatest-valued (largest). Sorting is a fundamental problem not only because of the tasks that it directly solves, but also because it is a foundation for many other algorithms such as the binary search algorithm or Kruskal's algorithm for the minimum spanning tree problem.
- Merge sort and quicksort are two examples of divide and conquer algorithms for sorting. Heapsort is a sorting algorithm that relies on adding to a heap and then repeatedly removing each element in sorted order.

- Hashing is the problem of assigning a meaningful integer index (hash value) for each object. Hash tables are a data structure for implementing sets and maps by applying the concept of hashing.
- Graph problems include a wide variety of problems involving the graph data type. Graph problems include traversal, minimum spanning trees, and shortest paths.
- Traversal is the problem of exploring all the vertices in a graph. Depth-first search and breadth-first search are both graph traversal algorithms that expand outward from a start vertex, ultimately visiting every reachable vertex.
- Minimum spanning trees is the problem of finding a lowest-cost way to connect all the vertices to each other, where cost is the sum of the selected edge weights. The two canonical greedy algorithms for finding a minimum spanning tree in a graph are Kruskal’s algorithm and Prim’s algorithm.
- Shortest paths is the problem of finding a lowest-cost way to get from one vertex to another. The output of a shortest paths algorithm is a shortest paths tree from the start vertex to every other vertex in the graph.
- Breadth-first search computes the unweighted shortest paths tree, the shortest paths in terms of the number of edges. Dijkstra’s algorithm computes the weighted shortest paths tree, the shortest paths in terms of the sum of the edge weights.

3.6 Computer Science Theory

- Problem modeling is constrained by the model of computation, or the rules of the underlying computer that is ultimately responsible for executing the algorithm. Combinatorial explosion poses a problem for computer algorithms because our model of computation assumes computers only have a single thread of execution and only execute one basic operation on each step.
- A Turing machine is an abstract model of computation for executing any computer algorithm. A Turing machine describes computers in terms of three key ideas: a memory bank, an instruction table, and a program counter.
- Although today’s computers are much more efficient than the first computers that realized the Turing machine, most computers still rely on the same fundamental assumptions about how to execute algorithms. Even as computers become faster over time inefficient algorithms still cannot be used to solve any problems larger than a few thousand elements.
- The complexity of a problem is the complexity (i.e., the time or memory resources required) of the most efficient algorithms for solving the problem. In this chapter, we have focused on solving problems known to have polynomial time

algorithms that can be described with a polynomial expression such as O(1), O(log N), O(N), O(N log N), O(N2), O(N3).

- Nondeterministic polynomial (NP) time complexity class refers to all problems that can be solved in polynomial time by a nondeterministic algorithm. A nondeterministic algorithm is a kind of algorithm that can rely on the special power of exploring infinitely many possible “alternate universes” in order to complete a computation.

- Technically, all P problems are also NP problems because we already have deterministic algorithms for solving them and therefore do not need to rely on the special power of nondeterminism. NP-complete refers to all the hardest NP problems—the combinatorial problems for which we do not have deterministic polynomial-time algorithms.

- Longest paths and the traveling salesperson problem (TSP) are two well-known examples of NP-complete problems. What makes both these problems difficult is that we do not have a simple rule for selecting the next element to include in the solution.

- All NP-complete problems can be reduced to all the others, so an algorithm for solving any NP-complete problem solves every NP-complete problem. The question of P versus NP asks whether it is possible to design a deterministic polynomial-time algorithm for solving any—and therefore all—of these NPcomplete problems.

- Most theoretical computer scientists believe that it is impossible to design an efficient algorithm for longest paths, TSP, or any other NP-complete problems. An efficient algorithm for any one NP-complete problems would not only directly solve routing and logistics problems but would also enable massive advancements in drug discovery through scientific simulation, for instance. It would also break essentially all modern Internet security and password systems—among thousands of other problems.