CHAPTER ONE

Introduction

The objective of this book is to study a broad variety of important and useful algorithms: methods for solving problems that are suited for computer implementation. We shall deal with many different areas of application, always concentrating on fundamental algorithms that are important to know and interesting to study. We shall spend enough time on each algorithm to understand its essential characteristics and to respect its subtleties. Our goal is to learn a large number of the most important algorithms used on computers today, well enough to be able to use and appreciate them.

The strategy that we use for understanding the programs presented in this book is to implement and test them, to experiment with their variants, to discuss their operation on small examples, and to try them out on larger examples similar to what we might encounter in practice. We shall use the C++ programming language to describe the algorithms, thus providing useful implementations at the same time. Our programs have a uniform style that is amenable to translation into other modern programming languages, as well.

We also pay careful attention to performance characteristics of our algorithms, to help us develop improved versions, compare different algorithms for the same task, and predict or guarantee performance for large problems. Understanding how the algorithms perform might require experimentation or mathematical analysis or both. We consider detailed information for many of the most important algorithms, developing analytic results directly when feasible, or calling on results from the research literature when necessary.
To illustrate our general approach to developing algorithmic solutions, we consider in this chapter a detailed example comprising a number of algorithms that solve a particular problem. The problem that we consider is not a toy problem; it is a fundamental computational task, and the solution that we develop is of use in a variety of applications. We start with a simple solution, then seek to understand that solution’s performance characteristics, which help us to see how to improve the algorithm. After a few iterations of this process, we come to an efficient and useful algorithm for solving the problem. This prototypical example sets the stage for our use of the same general methodology throughout the book.

We conclude the chapter with a short discussion of the contents of the book, including brief descriptions of what the major parts of the book are and how they relate to one another.

1.1 Algorithms

When we write a computer program, we are generally implementing a method that has been devised previously to solve some problem. This method is often independent of the particular computer to be used—it is likely to be equally appropriate for many computers and many computer languages. It is the method, rather than the computer program itself, that we must study to learn how the problem is being attacked. The term algorithm is used in computer science to describe a problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: They are central objects of study in many, if not most, areas of the field.

Most algorithms of interest involve methods of organizing the data involved in the computation. Objects created in this way are called data structures, and they also are central objects of study in computer science. Thus, algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms, and thus that we must study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, the book might well have been called *Algorithms and Data Structures in C++*.
When we use a computer to help us solve a problem, we typically are faced with a number of possible different approaches. For small problems, it hardly matters which approach we use, as long as we have one that solves the problem correctly. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time or space as efficiently as possible.

The primary reason for us to learn about algorithm design is that this discipline gives us the potential to reap huge savings, even to the point of making it possible to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such an example in Section 1.2 and on numerous other occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or 100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When a huge or complex computer program is to be developed, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. Those are the types of algorithms on which we concentrate in this book. We shall study a variety of fundamental algorithms that are useful for solving huge problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so, although we might expect to be using a large fraction of the algorithms in this book, we also might expect to have to implement only a smaller fraction of them. For example, the C++ Standard Template Library contains implementations of a host of fundamental algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus to more effectively use and tune advanced versions from a library. More important, the
opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with completely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In other words, we often implement basic algorithms tailored to our problem, rather than depending on a system routine, to make our solutions more portable and longer lasting. Another common reason to reimplement basic algorithms is that, despite the advances embodied in C++, the mechanisms that we use for sharing software are not always sufficiently powerful to allow us to conveniently tailor library programs to perform effectively on specific tasks.

Computer programs are often overoptimized. It may not be worthwhile to take pains to ensure that an implementation of a particular algorithm is the most efficient possible unless the algorithm is to be used for an enormous task or is to be used many times. Otherwise, a careful, relatively simple implementation will suffice: We can have some confidence that it will work, and it is likely to run perhaps five or 10 times slower at worst than the best possible version, which means that it may run for an extra few seconds. By contrast, the proper choice of algorithm in the first place can make a difference of a factor of 100 or 1000 or more, which might translate to minutes, hours, or even more in running time. In this book, we concentrate on the simplest reasonable implementations of the best algorithms.

The choice of the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called analysis of algorithms. Many of the algorithms that we study have been shown through analysis to have excellent performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, and we strive to be aware of how our algorithms might be expected to perform.
1.2 A Sample Problem: Connectivity

Suppose that we are given a sequence of pairs of integers, where each integer represents an object of some type and we are to interpret the pair $p-q$ as meaning “$p$ is connected to $q$. ” We assume the relation “is connected to” to be transitive: If $p$ is connected to $q$, and $q$ is connected to $r$, then $p$ is connected to $r$. Our goal is to write a program to filter out extraneous pairs from the set: When the program inputs a pair $p-q$, it should output the pair only if the pairs it has seen to that point do not imply that $p$ is connected to $q$. If the previous pairs do imply that $p$ is connected to $q$, then the program should ignore $p-q$ and should proceed to input the next pair. Figure 1.1 gives an example of this process.

Our problem is to devise a program that can remember sufficient information about the pairs it has seen to be able to decide whether or not a new pair of objects is connected. Informally, we refer to the task of designing such a method as the connectivity problem. This problem arises in a number of important applications. We briefly consider three examples here to indicate the fundamental nature of the problem.

For example, the integers might represent computers in a large network, and the pairs might represent connections in the network. Then, our program might be used to determine whether we need to establish a new direct connection for $p$ and $q$ to be able to communicate, or whether we could use existing connections to set up a communications path. In this kind of application, we might need to process millions of points and billions of connections, or more. As we shall see, it would be impossible to solve the problem for such an application without an efficient algorithm.

Similarly, the integers might represent contact points in an electrical network, and the pairs might represent wires connecting the points. In this case, we could use our program to find a way to connect all the points without any extraneous connections, if that is possible. There is no guarantee that the edges in the list will suffice to connect all the points—indeed, we shall soon see that determining whether or not they will could be a prime application of our program.

Figure 1.2 illustrates these two types of applications in a larger example. Examination of this figure gives us an appreciation for the
Figure 1.2
A large connectivity example

The objects in a connectivity problem might represent connection points, and the pairs might be connections between them, as indicated in this idealized example that might represent wires connecting buildings in a city or components on a computer chip. This graphical representation makes it possible for a human to spot nodes that are not connected, but the algorithm has to work with only the pairs of integers that it is given. Are the two nodes marked with the large black dots connected?

difficulty of the connectivity problem: How can we arrange to tell quickly whether any given two points in such a network are connected?

Still another example arises in certain programming environments where it is possible to declare two variable names as equivalent. The problem is to be able to determine whether two given names are equivalent, after a sequence of such declarations. This application is an early one that motivated the development of several of the algorithms that we are about to consider. It directly relates our problem to a simple abstraction that provides us with a way to make our algorithms useful for a wide variety of applications, as we shall see.

Applications such as the variable-name-equivalence problem described in the previous paragraph require that we associate an integer with each distinct variable name. This association is also implicit in the
network-connection and circuit-connection applications that we have
described. We shall be considering a host of algorithms in Chapters
10 through 16 that can provide this association in an efficient manner.
Thus, we can assume in this chapter, without loss of generality, that
we have \( N \) objects with integer names, from 0 to \( N - 1 \).

We are asking for a program that does a specific and well-defined
task. There are many other related problems that we might want to
have solved, as well. One of the first tasks that we face in developing
an algorithm is to be sure that we have specified the \textit{problem} in a
reasonable manner. The more we require of an algorithm, the more
time and space we may expect it to need to finish the task. It is
impossible to quantify this relationship a priori, and we often modify
a problem specification on finding that it is difficult or expensive to
solve, or, in happy circumstances, on finding that an algorithm can
provide information more useful than was called for in the original
specification.

For example, our connectivity-problem specification requires
only that our program somehow know whether or not any given pair
\( p-q \) is connected, and not that it be able to demonstrate any or all
ways to connect that pair. Adding a requirement for such a specifica-
tion makes the problem more difficult, and would lead us to a different
family of algorithms, which we consider briefly in Chapter 5 and in
detail in Part 7.

The specifications mentioned in the previous paragraph ask us for
\textit{more} information than our original one did; we could also ask
for \textit{less} information. For example, we might simply want to be able
to answer the question: “Are the \( M \) connections sufficient to connect
together all \( N \) objects?” This problem illustrates that, to develop
efficient algorithms, we often need to do high-level reasoning about
the abstract objects that we are processing. In this case, a fundamental
result from graph theory implies that all \( N \) objects are connected if
and only if the number of pairs output by the connectivity algorithm
is precisely \( N - 1 \) (see Section 5.4). In other words, a connectivity
algorithm will never output more than \( N - 1 \) pairs, because, once it
has output \( N - 1 \) pairs, any pair that it encounters from that point on
will be connected. Accordingly, we can get a program that answers
the yes–no question just posed by changing a program that solves the
connectivity problem to one that increments a counter, rather than
writing out each pair that was not previously connected, answering “yes” when the counter reaches \( N - 1 \) and “no” if it never does. This question is but one example of a host of questions that we might wish to answer regarding connectivity. The set of pairs in the input is called a graph, and the set of pairs output is called a spanning tree for that graph, which connects all the objects. We consider properties of graphs, spanning trees, and all manner of related algorithms in Part 7.

It is worthwhile to try to identify the fundamental operations that we will be performing, and so to make any algorithm that we develop for the connectivity task useful for a variety of similar tasks. Specifically, each time that we get a new pair, we have first to determine whether it represents a new connection, then to incorporate the information that the connection has been seen into its understanding about the connectivity of the objects such that it can check connections to be seen in the future. We encapsulate these two tasks as abstract operations by considering the integer input values to represent elements in abstract sets, and then design algorithms and data structures that can

- **Find** the set containing a given item.
- **Replace** the sets containing two given items by their **union**.

Organizing our algorithms in terms of these abstract operations does not seem to foreclose any options in solving the connectivity problem, and the operations may be useful for solving other problems. Developing ever more powerful layers of abstraction is an essential process in computer science in general and in algorithm design in particular, and we shall turn to it on numerous occasions throughout this book. In this chapter, we use abstract thinking in an informal way to guide us in designing programs to solve the connectivity problem; in Chapter 4, we shall see how to encapsulate abstractions in C++ code.

The connectivity problem is easily solved in terms of the **find** and **union** abstract operations. After reading a new pair \( p-q \) from the input, we perform a **find** operation for each member of the pair. If the members of the pair are in the same set, we move on to the next pair; if they are not, we do a **union** operation and write out the pair. The sets represent **connected components**: subsets of the objects with the property that any two objects in a given component are connected. This approach reduces the development of an algorithmic solution for connectivity to the tasks of defining a data structure representing the
sets and developing union and find algorithms that efficiently use that data structure.

There are many possible ways to represent and process abstract sets, which we consider in more detail in Chapter 4. In this chapter, our focus is on finding a representation that can support efficiently the union and find operations that we see in solving the connectivity problem.

**Exercises**

1.1 Give the output that a connectivity algorithm should produce when given the input 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3.

1.2 List all the different ways to connect two different objects for the example in Figure 1.1.

1.3 Describe a simple method for counting the number of sets remaining after using the union and find operations to solve the connectivity problem as described in the text.

### 1.3 Union–Find Algorithms

The first step in the process of developing an efficient algorithm to solve a given problem is to implement a simple algorithm that solves the problem. If we need to solve a few particular problem instances that turn out to be easy, then the simple implementation may finish the job for us. If a more sophisticated algorithm is called for, then the simple implementation provides us with a correctness check for small cases and a baseline for evaluating performance characteristics. We always care about efficiency, but our primary concern in developing the first program that we write to solve a problem is to make sure that the program is a correct solution to the problem.

The first idea that might come to mind is somehow to save all the input pairs, then to write a function to pass through them to try to discover whether the next pair of objects is connected. We shall use a different approach. First, the number of pairs might be sufficiently large to preclude our saving them all in memory in practical applications. Second, and more to the point, no simple method immediately suggests itself for determining whether two objects are connected from the set of all the connections, even if we could save them all! We consider a basic method that takes this approach in Chapter 5, but the methods that we shall consider in this chapter are simpler, because they

![Figure 1.3](image)

*Example of quick find (slow union)*

This sequence depicts the contents of the id array after each of the pairs at left is processed by the quick-find algorithm (Program 1.1). Shaded entries are those that change for the union operation. When we process the pair p q, we change all entries with the value id[p] to have the value id[q].
Program 1.1 Quick-find solution to connectivity problem

This program reads a sequence of pairs of nonnegative integers less than \( N \) from standard input (interpreting the pair \( p, q \) to mean “connect object \( p \) to object \( q \)”) and prints out pairs representing objects that are not yet connected. It maintains an array \( \text{id} \) that has an entry for each object, with the property that \( \text{id}[p] \) and \( \text{id}[q] \) are equal if and only if \( p \) and \( q \) are connected. For simplicity, we define \( N \) as a compile-time constant. Alternatively, we could take it from the input and allocate the \( \text{id} \) array dynamically (see Section 3.2).

```
#include <iostream>
static const int N = 10000;
int main()
{
    int i, p, q, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (cin >> p >> q)
    {
        int t = id[p];
        if (t == id[q]) continue;
        for (i = 0; i < N; i++)
            if (id[i] == t) id[i] = id[q];
        cout << " " << p << " " << q << endl;
    }
}
```

solve a less difficult problem, and are more efficient, because they do not require saving all the pairs. They all use an array of integers—one corresponding to each object—to hold the requisite information to be able to implement union and find.

Arrays are elementary data structures that we shall discuss in detail in Section 3.2. Here, we use them in their simplest form: we declare that we expect to use, say, 1000 integers, by writing \( a[1000] \); then we refer to the \( i \)th integer in the array by writing \( a[i] \) for \( 0 \leq i < 1000 \).

Program 1.1 is an implementation of a simple algorithm called the quick-find algorithm that solves the connectivity problem. The basis of this algorithm is an array of integers with the property that \( p \) and \( q \) are connected if and only if the \( p \)th and \( q \)th array entries are equal. We initialize the \( i \)th array entry to \( i \) for \( 0 \leq i < N \). To implement the union operation for \( p \) and \( q \), we go through the array,
changing all the entries with the same name as \( p \) to have the same name as \( q \). This choice is arbitrary—we could have decided to change all the entries with the same name as \( q \) to have the same name as \( p \).

Figure 1.3 shows the changes to the array for the union operations in the example in Figure 1.1. To implement \textit{find}, we just test the indicated array entries for equality—hence the name \textit{quick find}. The union operation, on the other hand, involves scanning through the whole array for each input pair.

\textbf{Property 1.1} \textit{The quick-find algorithm executes at least} \( MN \) \textit{instructions to solve a connectivity problem with} \( N \) \textit{objects that involves} \( M \) \textit{union operations.}

For each of the \( M \) union operations, we iterate the for loop \( N \) times. Each iteration requires at least one instruction (if only to check whether the loop is finished).

We can execute tens or hundreds of millions of instructions per second on modern computers, so this cost is not noticeable if \( M \) and \( N \) are small, but we also might find ourselves with billions of objects and millions of input pairs to process in a modern application. The inescapable conclusion is that we cannot feasibly solve such a problem using the quick-find algorithm (see Exercise 1.10). We consider the process of quantifying such a conclusion precisely in Chapter 2.

Figure 1.4 shows a graphical representation of Figure 1.3. We may think of some of the objects as representing the set to which they belong, and all of the other objects as pointing to the representative in their set. The reason for moving to this graphical representation of the array will become clear soon. Observe that the connections between objects in this representation are not necessarily the same as the connections in the input pairs—they are the information that the algorithm chooses to remember to be able to know whether future pairs are connected.

The next algorithm that we consider is a complementary method called the \textit{quick-union algorithm}. It is based on the same data structure—an array indexed by object names—but it uses a different interpretation of the values that leads to more complex abstract structures. Each object points to another object in the same set, in a structure with no cycles. To determine whether two objects are in the same set, we follow pointers for each until we reach an object that
points to itself. The objects are in the same set if and only if this process leads them to the same object. If they are not in the same set, we wind up at different objects (which point to themselves). To form the union, then we just link one to the other to perform the union operation; hence the name quick-union.

Figure 1.5 shows the graphical representation that corresponds to Figure 1.4 for the operation of the quick-union algorithm on the example of Figure 1.1, and Figure 1.6 shows the corresponding changes to the id array. The graphical representation of the data structure makes it relatively easy to understand the operation of the algorithm—input pairs that are known to be connected in the data are also connected to one another in the data structure. As mentioned previously, it is important to note at the outset that the connections in the data structure are not necessarily the same as the connections in the application implied by the input pairs; rather, they are constructed by the algorithm to facilitate efficient implementation of union and find.

The connected components depicted in Figure 1.5 are called trees; they are fundamental combinatorial structures that we shall encounter on numerous occasions throughout the book. We shall consider the properties of trees in detail in Chapter 5. For the union and find operations, the trees in Figure 1.5 are useful because they are quick to build and have the property that two objects are connected in the tree if and only if the objects are connected in the input. By moving up the tree, we can easily find the root of the tree containing each object, so we have a way to find whether or not they are connected. Each tree has precisely one object that points to itself, which is called the root of the tree. The self-pointer is not shown in the diagrams. When we start at any object in the tree, move to the object to which it points, then move to the object to which that object points, and so forth, we eventually end up at the root, always. We can prove this property to be true by induction: It is true after the array is initialized to have every object point to itself, and if it is true before a given union operation, it is certainly true afterward.

The diagrams in Figure 1.4 for the quick-find algorithm have the same properties as those described in the previous paragraph. The difference between the two is that we reach the root from all the nodes in the quick-find trees after following just one link, whereas we might need to follow several links to get to the root in a quick-union tree.
Program 1.2 Quick-union solution to connectivity problem

If we replace the body of the \texttt{while} loop in Program 1.1 by this code, we have a program that meets the same specifications as Program 1.1, but does less computation for the \texttt{union} operation at the expense of more computation for the \texttt{find} operation. The \texttt{for} loops and subsequent \texttt{if} statement in this code specify the necessary and sufficient conditions on the \texttt{id} array for \texttt{p} and \texttt{q} to be connected. The assignment statement \texttt{id[i] = j} implements the \texttt{union} operation.

\begin{verbatim}
for (i = p; i != id[i]; i = id[i]) ;
for (j = q; j != id[j]; j = id[j]) ;
if (i == j) continue;
id[i] = j;
cout << " " << p << " " << q << endl;
\end{verbatim}

Program 1.2 is an implementation of the \texttt{union} and \texttt{find} operations that comprise the quick-union algorithm to solve the connectivity problem. The quick-union algorithm would seem to be faster than the quick-find algorithm, because it does not have to go through the entire array for each input pair; but how much faster is it? This question is more difficult to answer here than it was for quick find, because the running time is much more dependent on the nature of the input. By running empirical studies or doing mathematical analysis (see Chapter 2), we can convince ourselves that Program 1.2 is far more efficient than Program 1.1, and that it is feasible to consider using Program 1.2 for huge practical problems. We shall discuss one such empirical study at the end of this section. For the moment, we can regard quick union as an improvement because it removes quick find’s main liability (that the program requires at least \(NM\) instructions to process \(M\) \texttt{union} operations among \(N\) objects).

This difference between quick union and quick find certainly represents an improvement, but quick union still has the liability that we cannot \textit{guarantee} it to be substantially faster than quick find in every case, because the input data could conspire to make the \texttt{find} operation slow.
Property 1.2  For $M > N$, the quick-union algorithm could take more than $MN/2$ instructions to solve a connectivity problem with $M$ pairs of $N$ objects.

Suppose that the input pairs come in the order 1–2, then 2–3, then 3–4, and so forth. After $N - 1$ such pairs, we have $N$ objects all in the same set, and the tree that is formed by the quick-union algorithm is a straight line, with $N$ pointing to $N - 1$, which points to $N - 2$, which points to $N - 3$, and so forth. To execute the find operation for object $N$, the program has to follow $N - 1$ pointers. Thus, the average number of pointers followed for the first $N$ pairs is

$$0 + 1 + \ldots + (N - 1))/N = (N - 1)/2.$$  

Now suppose that the remainder of the pairs all connect $N$ to some other object. The find operation for each of these pairs involves at least $(N - 1)$ pointers. The grand total for the $M$ find operations for this sequence of input pairs is certainly greater than $MN/2$.

Fortunately, there is an easy modification to the algorithm that allows us to guarantee that bad cases such as this one do not occur. Rather than arbitrarily connecting the second tree to the first for union, we keep track of the number of nodes in each tree and always connect the smaller tree to the larger. This change requires slightly more code and another array to hold the node counts, as shown in Program 1.3, but it leads to substantial improvements in efficiency. We refer to this algorithm as the weighted quick-union algorithm.

Figure 1.7 shows the forest of trees constructed by the weighted union–find algorithm for the example input in Figure 1.1. Even for this small example, the paths in the trees are substantially shorter than for the unweighted version in Figure 1.5. Figure 1.8 illustrates what happens in the worst case, when the sizes of the sets to be merged in the union operation are always equal (and a power of 2). These tree structures look complex, but they have the simple property that the maximum number of pointers that we need to follow to get to the root in a tree of $2^n$ nodes is $n$. Furthermore, when we merge two trees of $2^n$ nodes, we get a tree of $2^{n+1}$ nodes, and we increase the maximum distance to the root to $n + 1$. This observation generalizes to provide a proof that the weighted algorithm is substantially more efficient than the unweighted algorithm.
INTRODUCTION

1.3 17

INTRODUCTION

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

1.3 17

Program 1.3 Weighted version of quick union

This program is a modification to the quick-union algorithm (see Program 1.2) that keeps an additional array sz for the purpose of maintaining, for each object with id[i] = i, the number of nodes in the associated tree, so that the union operation can link the smaller of the two specified trees to the larger, thus preventing the growth of long paths in the trees.

#include <iostream.h>
static const int N = 10000;
int main()
{
    int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++)
    {
        id[i] = i; sz[i] = 1;
    }
    while (cin >> p >> q)
    {
        for (i = p; i != id[i]; i = id[i]);
        for (j = q; j != id[j]; j = id[j]);
        if (i == j) continue;
        if (sz[i] < sz[j])
        {
            id[i] = j; sz[j] += sz[i];
        }
        else
        {
            id[j] = i; sz[i] += sz[j];
        }
        cout << " " << p << " " << q << endl;
    }
}

Property 1.3 The weighted quick-union algorithm follows at most \( \lg N \) pointers to determine whether two of \( N \) objects are connected.

We can prove that the union operation preserves the property that the number of pointers followed from any node to the root in a set of \( k \) objects is no greater than \( \lg k \). When we combine a set of \( i \) nodes with a set of \( j \) nodes with \( i \leq j \), we increase the number of pointers that must be followed in the smaller set by 1, but they are now in a set of size \( i + j \), so the property is preserved because \( 1 + \lg i = \lg(i + i) \leq \lg(i + j) \).

The practical implication of Property 1.3 is that the weighted quick-union algorithm uses at most a constant times \( M \lg N \) instructions to process \( M \) edges on \( N \) objects (see Exercise 1.9). This result is

Figure 1.8

Weighted quick union (worst case)

The worst scenario for the weighted quick-union algorithm is that each union operation links trees of equal size. If the number of objects is less than \( 2^n \), the distance from any node to the root of its tree is less than \( n \).
in stark contrast to our finding that quick find always (and quick union sometimes) uses at least $MN/2$ instructions. The conclusion is that, with weighted quick union, we can guarantee that we can solve huge practical problems in a reasonable amount of time (see Exercise 1.11). For the price of a few extra lines of code, we get a program that is literally millions of times faster than the simpler algorithms for the huge problems that we might encounter in practical applications.

It is evident from the diagrams that relatively few nodes are far from the root; indeed, empirical studies on huge problems tell us that the weighted quick-union algorithm of Program 1.3 typically can solve practical problems in linear time. That is, the cost of running the algorithm is within a constant factor of the cost of reading the input. We could hardly expect to find a more efficient algorithm.

We immediately come to the question of whether or not we can find an algorithm that has guaranteed linear performance. This question is an extremely difficult one that plagued researchers for many years (see Section 2.7). There are a number of easy ways to improve the weighted quick-union algorithm further. Ideally, we would like every node to point directly to the root of its tree, but we do not want to pay the price of changing a large number of pointers, as we did in the quick-union algorithm. We can approach the ideal simply by making all the nodes that we do examine point to the root. This step seems drastic at first blush, but it is easy to implement, and there is nothing sacrosanct about the structure of these trees: If we can modify them to make the algorithm more efficient, we should do so. We can implement this method, called path compression, easily, by adding another pass through each path during the union operation, setting the id entry corresponding to each vertex encountered along the way to point to the root. The net result is to flatten the trees almost completely, approximating the ideal achieved by the quick-find algorithm, as illustrated in Figure 1.9. The analysis that establishes this fact is extremely complex, but the method is simple and effective. Figure 1.11 shows the result of path compression for a large example.

There are many other ways to implement path compression. For example, Program 1.4 is an implementation that compresses the paths by making each link skip to the next node in the path on the way up the tree, as depicted in Figure 1.10. This method is slightly easier to implement than full path compression (see Exercise 1.16), and achieves
Introduction

1.3 Path compression by halving

We can nearly halve the length of paths on the way up the tree by taking two links at a time, and setting the bottom one to point to the same node as the top one, as shown in this example. The net result of performing this operation on every path that we traverse is asymptotically the same as full path compression.

Program 1.4 Path compression by halving

If we replace the for loops in Program 1.3 by this code, we halve the length of any path that we traverse. The net result of this change is that the trees become almost completely flat after a long sequence of operations.

```cpp
for (i = p; i != id[i]; i = id[i])
    id[i] = id[id[i]];
for (j = q; j != id[j]; j = id[j])
    id[j] = id[id[j]];
```

the same net result. We refer to this variant as weighted quick-union with path compression by halving. Which of these methods is the more effective? Is the savings achieved worth the extra time required to implement path compression? Is there some other technique that we should consider? To answer these questions, we need to look more carefully at the algorithms and implementations. We shall return to this topic in Chapter 2, in the context of our discussion of basic approaches to the analysis of algorithms.

The end result of the succession of algorithms that we have considered to solve the connectivity problem is about the best that we could hope for in any practical sense. We have algorithms that are easy to implement whose running time is guaranteed to be within a constant factor of the cost of gathering the data. Moreover, the algorithms are online algorithms that consider each edge once, using space proportional to the number of objects, so there is no limitation on the number of edges that they can handle. The empirical studies in Table 1.1 validate our conclusion that Program 1.3 and its path-compression variations are useful even for huge practical applications. Choosing which is the best among these algorithms requires careful and sophisticated analysis (see Chapter 2).

Exercises

1.4 Show the contents of the id array after each union operation when you use the quick-find algorithm (Program 1.1) to solve the connectivity problem for the sequence 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3. Also give the number of times the program accesses the id array for each input pair.

1.5 Do Exercise 1.4, but use the quick-union algorithm (Program 1.2).
Table 1.1 Empirical study of union-find algorithms

These relative timings for solving random connectivity problems using various union–find algorithms demonstrate the effectiveness of the weighted version of the quick union algorithm. The added incremental benefit due to path compression is less important. In these experiments, \( M \) is the number of random connections generated until all \( N \) objects were connected. This process involves substantially more \texttt{find} operations than \texttt{union} operations, so quick union is substantially slower than quick find. Neither quick find nor quick union is feasible for huge \( N \). The running time for the weighted methods is evidently proportional to \( N \), as it approximately doubles when \( N \) is doubled.

<table>
<thead>
<tr>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2500</td>
</tr>
<tr>
<td>5000</td>
</tr>
<tr>
<td>10000</td>
</tr>
<tr>
<td>25000</td>
</tr>
<tr>
<td>50000</td>
</tr>
<tr>
<td>100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( M )</th>
<th>( F )</th>
<th>( U )</th>
<th>( W )</th>
<th>( P )</th>
<th>( H )</th>
</tr>
</thead>
<tbody>
<tr>
<td>6206</td>
<td>14</td>
<td>25</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>20236</td>
<td>82</td>
<td>210</td>
<td>13</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>41913</td>
<td>304</td>
<td>1172</td>
<td>46</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>83857</td>
<td>1216</td>
<td>4577</td>
<td>91</td>
<td>73</td>
<td>50</td>
</tr>
<tr>
<td>309802</td>
<td>219</td>
<td>208</td>
<td>208</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>708701</td>
<td>469</td>
<td>387</td>
<td>497</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1545119</td>
<td>1071</td>
<td>1106</td>
<td>1096</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key:
- \( F \) quick find (Program 1.1)
- \( U \) quick union (Program 1.2)
- \( W \) weighted quick union (Program 1.3)
- \( P \) weighted quick union with path compression (Exercise 1.16)
- \( H \) weighted quick union with halving (Program 1.4)

\( \def \textbf{I} {1.6} \) Give the contents of the \( \textit{id} \) array after each \texttt{union} operation for the weighted quick-union algorithm running on the examples corresponding to Figure 1.7 and Figure 1.8.

\( \def \textbf{I} {1.7} \) Do Exercise 1.4, but use the weighted quick-union algorithm (Program 1.3).

\( \def \textbf{I} {1.8} \) Do Exercise 1.4, but use the weighted quick-union algorithm with path compression by halving (Program 1.4).

\( \def \textbf{I} {1.9} \) Prove an upper bound on the number of machine instructions required to process \( M \) connections on \( N \) objects using Program 1.3. You may assume, for example, that any C++ assignment statement always requires less than \( c \) instructions, for some fixed constant \( c \).
1.10 Estimate the minimum amount of time (in days) that would be required for quick find (Program 1.1) to solve a problem with $10^5$ objects and $10^6$ input pairs, on a computer capable of executing $10^9$ instructions per second. Assume that each iteration of the inner for loop requires at least 10 instructions.

1.11 Estimate the maximum amount of time (in seconds) that would be required for weighted quick union (Program 1.3) to solve a problem with $10^5$ objects and $10^6$ input pairs, on a computer capable of executing $10^9$ instructions per second. Assume that each iteration of the outer while loop requires at most 100 instructions.

1.12 Compute the average distance from a node to the root in a worst-case tree of $2^n$ nodes built by the weighted quick-union algorithm.

1.13 Draw a diagram like Figure 1.10, starting with eight nodes instead of nine.

1.14 Give a sequence of input pairs that causes the weighted quick-union algorithm (Program 1.3) to produce a path of length 4.

1.15 Give a sequence of input pairs that causes the weighted quick-union algorithm with path compression by halving (Program 1.4) to produce a path of length 4.

1.16 Show how to modify Program 1.3 to implement full path compression, where we complete each union operation by making every node that we touch point to the root of the new tree.

1.17 Answer Exercise 1.4, but using the weighted quick-union algorithm with full path compression (Exercise 1.16).

1.18 Give a sequence of input pairs that causes the weighted quick-union algorithm with full path compression (Exercise 1.16) to produce a path of length 4.

1.19 Give an example showing that modifying quick union (Program 1.2) to implement full path compression (see Exercise 1.16) is not sufficient to ensure that the trees have no long paths.

1.20 Modify Program 1.3 to use the height of the trees (longest path from any node to the root), instead of the weight, to decide whether to set $\text{id}[i] = j$ or $\text{id}[j] = i$. Run empirical studies to compare this variant with Program 1.3.

1.21 Show that Property 1.3 holds for the algorithm described in Exercise 1.20.

1.22 Modify Program 1.4 to generate random pairs of integers between 0 and $N - 1$ instead of reading them from standard input, and to loop until $N - 1$
union operations have been performed. Run your program for $N = 10^1$, $10^4$, $10^5$, and $10^6$ and print out the total number of edges generated for each value of $N$.

- **1.23** Modify your program from Exercise 1.22 to plot the number of edges needed to connect $N$ items, for $100 \leq N \leq 1000$.

- **1.24** Give an approximate formula for the number of random edges that are required to connect $N$ objects, as a function of $N$.

### 1.4 Perspective

Each of the algorithms that we considered in Section 1.3 seems to be an improvement over the previous in some intuitive sense, but the process is perhaps artificially smooth because we have the benefit of hindsight in looking over the development of the algorithms as they were studied by researchers over the years (see reference section). The implementations are simple and the problem is well specified, so we can evaluate the various algorithms directly by running empirical studies. Furthermore, we can validate these studies and quantify the comparative performance of these algorithms (see Chapter 2). Not all the problem domains in this book are as well developed as this one, and we certainly can run into complex algorithms that are difficult to compare and mathematical problems that are difficult to solve. We strive to make objective scientific judgements about the algorithms that we use, while gaining experience learning the properties of implementations running on actual data from applications or random test data.

The process is prototypical of the way that we consider various algorithms for fundamental problems throughout the book. When possible, we follow the same basic steps that we took for union-find algorithms in Section 1.2, some of which are highlighted in this list:

- Decide on a complete and specific problem statement, including identifying fundamental abstract operations that are intrinsic to the problem.
- Carefully develop a succinct implementation for a straightforward algorithm.
- Develop improved implementations through a process of step-wise refinement, validating the efficacy of ideas for improvement through empirical analysis, mathematical analysis, or both.
INTRODUCTION

- Find high-level abstract representations of data structures or algorithms in operation that enable effective high-level design of improved versions.
- Strive for worst-case performance guarantees when possible, but accept good performance on actual data when available.

The potential for spectacular performance improvements for practical problems such as those that we saw in Section 1.2 makes algorithm design a compelling field of study; few other design activities hold the potential to reap savings factors of millions or billions, or more.

More important, as the scale of our computational power and our applications increases, the gap between a fast algorithm and a slow one grows. A new computer might be 10 times faster and be able to process 10 times as much data as an old one, but if we are using a quadratic algorithm such as quick find, the new computer will take 10 times as long on the new job as the old one took to finish the old job! This statement seems counterintuitive at first, but it is easily verified by the simple identity \((10N)^2/10 = 10N^2\), as we shall see in Chapter 2. As computational power increases to allow us to take on larger and larger problems, the importance of having efficient algorithms increases, as well.

Developing an efficient algorithm is an intellectually satisfying activity that can have direct practical payoff. As the connectivity problem indicates, a simply stated problem can lead us to study numerous algorithms that are not only both useful and interesting, but also intricate and challenging to understand. We shall encounter many ingenious algorithms that have been developed over the years for a host of practical problems. As the scope of applicability of computational solutions to scientific and commercial problems widens, so also grows the importance of being able to apply efficient algorithms to solve known problems and of being able to develop efficient solutions to new problems.

Exercises

1.25 Suppose that we use weighted quick union to process 10 times as many connections on a new computer that is 10 times as fast as an old one. How much longer would it take the new computer to finish the new job than it took the old one to finish the old job?

1.26 Answer Exercise 1.25 for the case where we use an algorithm that requires \(N^3\) instructions.
1.5 Summary of Topics

This section comprises brief descriptions of the major parts of the book, giving specific topics covered and an indication of our general orientation toward the material. This set of topics is intended to touch on as many fundamental algorithms as possible. Some of the areas covered are core computer-science areas that we study in depth to learn basic algorithms of wide applicability. Other algorithms that we discuss are from advanced fields of study within computer science and related fields, such as numerical analysis and operations research—in these cases, our treatment serves as an introduction to these fields through examination of basic methods.

The first four parts of the book, which are contained in this volume, cover the most widely used set of algorithms and data structures, a first level of abstraction for collections of objects with keys that can support a broad variety of important fundamental algorithms. The algorithms that we consider are the products of decades of research and development, and continue to play an essential role in the ever-expanding applications of computation.

Fundamentals (Part 1) in the context of this book are the basic principles and methodology that we use to implement, analyze, and compare algorithms. The material in Chapter 1 motivates our study of algorithm design and analysis; in Chapter 2, we consider basic methods of obtaining quantitative information about the performance of algorithms.

Data Structures (Part 2) go hand-in-hand with algorithms: we shall develop a thorough understanding of data representation methods for use throughout the rest of the book. We begin with an introduction to basic concrete data structures in Chapter 3, including arrays, linked lists, and strings; then we consider recursive programs and data structures in Chapter 5, in particular trees and algorithms for manipulating them. In Chapter 4, we consider fundamental abstract data types (ADTs) such as stacks and queues, including implementations using elementary data structures.

Sorting algorithms (Part 3) for rearranging files into order are of fundamental importance. We consider a variety of algorithms in considerable depth, including Shellsort, quicksort, mergesort, heapsort, and radix sorts. We shall encounter algorithms for several related
problems, including priority queues, selection, and merging. Many of these algorithms will find application as the basis for other algorithms later in the book.

Searching algorithms (Part 4) for finding specific items among large collections of items are also of fundamental importance. We discuss basic and advanced methods for searching using trees and digital key transformations, including binary search trees, balanced trees, hashing, digital search trees and tries, and methods appropriate for huge files. We note relationships among these methods, comparative performance statistics, and correspondences to sorting methods.

Parts 5 through 8, which are contained in a separate volume, cover advanced applications of the algorithms described here for a diverse set of applications—a second level of abstractions specific to a number of important applications areas. We also delve more deeply into techniques of algorithm design and analysis. Many of the problems that we touch on are the subject on ongoing research.

String Processing algorithms (Part 5) include a range of methods for processing (long) sequences of characters. String searching leads to pattern matching, which leads to parsing. File-compression techniques are also considered. Again, an introduction to advanced topics is given through treatment of some elementary problems that are important in their own right.

Geometric Algorithms (Part 6) are methods for solving problems involving points and lines (and other simple geometric objects) that have only recently come into use. We consider algorithms for finding the convex hull of a set of points, for finding intersections among geometric objects, for solving closest-point problems, and for multidimensional searching. Many of these methods nicely complement the more elementary sorting and searching methods.

Graph Algorithms (Part 7) are useful for a variety of difficult and important problems. A general strategy for searching in graphs is developed and applied to fundamental connectivity problems, including shortest path, minimum spanning tree, network flow, and matching. A unified treatment of these algorithms shows that they are all based on the same procedure, and that this procedure depends on the basic priority queue ADT.
Advanced Topics (Part 8) are discussed for the purpose of relating the material in the book to several other advanced fields of study. We begin with major approaches to the design and analysis of algorithms, including divide-and-conquer, dynamic programming, randomization, and amortization. We survey linear programming, the fast Fourier transform, NP-completeness, and other advanced topics from an introductory viewpoint to gain appreciation for the interesting advanced fields of study suggested by the elementary problems confronted in this book.

The study of algorithms is interesting because it is a new field (almost all the algorithms that we study are less than 50 years old, and some were just recently discovered) with a rich tradition (a few algorithms have been known for thousands of years). New discoveries are constantly being made, but few algorithms are completely understood. In this book we shall consider intricate, complicated, and difficult algorithms as well as elegant, simple, and easy algorithms. Our challenge is to understand the former and to appreciate the latter in the context of many different potential applications. In doing so, we shall explore a variety of useful tools and develop a style of algorithmic thinking that will serve us well in computational challenges to come.