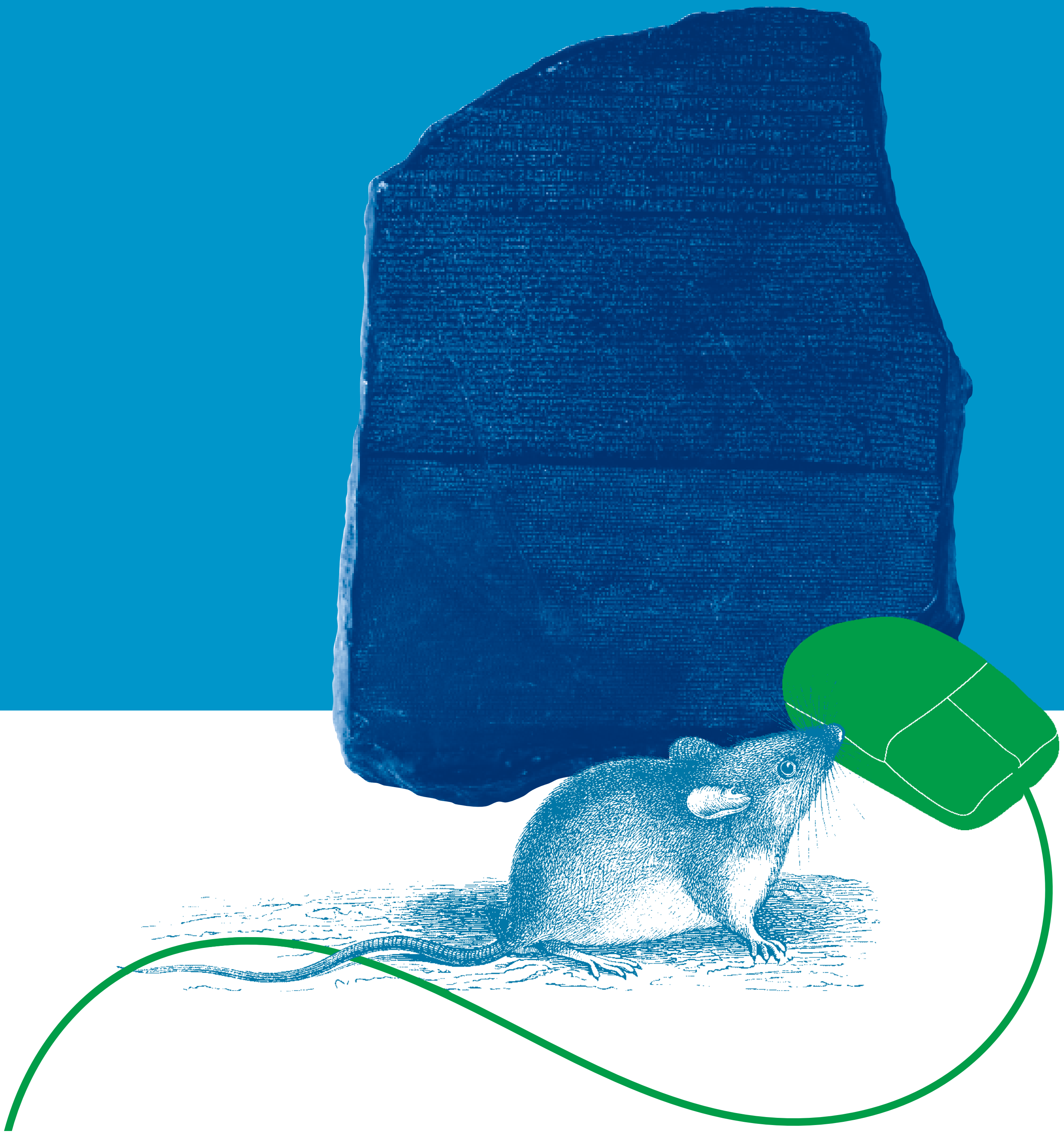
AN INTRODUCTION TO

BIOINFORMATICS ALGORITHMS

NEIL C. JONES AND PAVEL A. PEVZNER



1 *Introduction*

Imagine Alice, Bob, and two piles of ten rocks. Alice and Bob are bored one Saturday afternoon so they play the following game. In each turn a player may either take one rock from a single pile, or one rock from both piles. Once the rocks are taken, they are removed from play; the player that takes the last rock wins the game. Alice moves first.

It is not immediately clear what the winning strategy is, or even if there is one. Does the first player (or the second) always have an advantage? Bob tries to analyze the game and realizes that there are too many variants in the game with two piles of ten rocks (which we will refer to as the *10+10 game*). Using a reductionist approach, he first tries to find a strategy for the simpler 2+2 game. He quickly sees that the second player—himself, in this case—wins any 2+2 game, so he decides to write the "winning recipe":

> If Alice takes one rock from each pile, I will take the remaining rocks and win. If Alice takes one rock, I will take one rock from the same pile. As a result, there will be only one pile and it will have two rocks in it, so Alice's only choice will be to take one of them. I will take the remaining rock to win the game.

Inspired by this analysis, Bob makes a leap of faith: the second player (i.e., himself) wins in any $n+n$ game, for $n \geq 2$. Of course, every hypothesis must be confirmed by experiment, so Bob plays a few rounds with Alice. It turns out that sometimes he wins and sometimes he loses. Bob tries to come up with a simple recipe for the 3+3 game, but there are a large number of different game sequences to consider, and the recipe quickly gets too complicated. There is simply no hope of writing a recipe for the 10+10 game because the number of different strategies that Alice can take is enormous.

Meanwhile, Alice quickly realizes that she will always lose the 2+2 game, but she does not lose hope of finding a winning strategy for the 3+3 game.

Moreover, she took Algorithms 101 and she understands that recipes written in the style that Bob uses will not help very much: recipe-style instructions are not a sufficiently expressive language for describing algorithms. Instead, she begins by drawing the following table filled with the symbols $\uparrow$, $\leftarrow$, $\nwarrow$, and $*$. The entry in position (i, j) (i.e., the ith row and the jth column) describes the moves that Alice will make in the $i + j$ game, with i and j rocks in piles A and B respectively. A $\leftarrow$ indicates that she should take one stone from pile B. A $\uparrow$ indicates that she should take one stone from pile A. A $\nwarrow$ indicates that she should take one stone from each pile, and $*$ indicates that she should not bother playing the game because she will definitely lose against an opponent who has a clue.

	0	1	2	3	4	5	6	7	8	9	10
0	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
1	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
2	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
3	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
4	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
5	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
6	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
7	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
8	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
9	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
10	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$

For example, if she is faced with the 3+3 game, she finds a $\nwarrow$ in the third row and third column, indicating that she should take a rock from each pile. This makes Bob take the first move in a 2+2 game, which is marked with a $*$. No matter what he does, Alice wins. Suppose Bob takes a rock from pile B—this leads to the 2+1 game. Alice again consults the table by reading the entry at (2,1), seeing that she should also take a rock from pile B leaving two rocks in A. However, if Bob had instead taken a rock from pile A, Alice would consult entry (1,2) to find $\uparrow$. She again should also take a rock from pile A, leaving two rocks in pile B.

Impressed by the table, Bob learns how to use it to win the 10+10 game. However, Bob does not know how to construct a similar table for the 20+20 game. The problem is not that Bob is stupid, but that he has not studied algorithms. Even if, through sheer luck, Bob figured how to always win the

20+20 game, he could neither say with confidence that it would work no matter what Alice did, nor would he even be able to write down the recipe for the general $n + n$ game. More embarrassing to Bob is that the a general 10+10+10 game with three piles would turn into an impossible conundrum for him.

There are two things Bob could do to remedy his situation. First, he could take a class in algorithms to learn how to solve problems like the rock puzzle. Second, he could memorize a suitably large table that Alice gives him and use that to play the game. Leading questions notwithstanding, what would you do as a biologist?

Of course, the answer we expect to hear from most rational people is "Why in the world do I care about a game with two nerdy people and a bunch of rocks? I'm interested in biology, and this game has nothing to do with me." This is not actually true: the rock game is in fact the ubiquitous *sequence alignment* problem in disguise. Although it is not immediately clear what DNA sequence alignment and the rock game have in common, the computational idea used to solve both problems is the same. The fact that Bob was not able to find the strategy for the game indicates that he does not understand how alignment algorithms work either. He might disagree if he uses alignment algorithms or BLAST[1] on a daily basis, but we argue that since he failed to come up with a strategy for the 10+10 rock game, he will also fail when confronted with a new flavor of alignment problem or a particularly complex similarity analysis. More troubling to Bob, he may find it difficult to compete with the scads of new biologists who think algorithmically about biological problems.[2]

Many biologists are comfortable using algorithms like BLAST without really understanding how the underlying algorithm works. This is not substantially different from a diligent robot following Alice's winning strategy table, but it does have an important consequence. BLAST solves a particular problem only approximately and it has certain systematic weaknesses. We're not picking on BLAST here: the reason that BLAST has these limitations is, in part, because of the particular problem that it solves. Users who do not know how BLAST works might misapply the algorithm or misinterpret the results it returns. Biologists sometimes use bioinformatics tools simply as computational protocols in quite the same way that an uninformed mathematician

1. BLAST is a database search tool—a Google for biological sequences—that will be introduced later in this book.
2. These "new biologists" have probably already found another even more elegant solution of the rocks problem that does not require the construction of a table.

might use experimental protocols without any background in biochemistry or molecular biology. In either case, important observations might be missed or incorrect conclusions drawn. Besides, intellectually interesting work can quickly become mere drudgery if one does not really understand it.

Many recent bioinformatics books cater to this sort of protocol-centric practical approach to bioinformatics. They focus on parameter settings, specific features of application, and other details without revealing the ideas behind the algorithms. This trend often follows the tradition of biology books of presenting material as a collection of facts and discoveries. In contrast, introductory books in algorithms usually focus on ideas rather than on the details of computational recipes.

Since bioinformatics is a computational science, a bioinformatics textbook should strive to present the principles that drive an algorithm's design, rather than list a stamp collection of the algorithms themselves. We hope that describing the intellectual content of bioinformatics will help retain your interest in the subject. In this book we attempt to show that a handful of algorithmic ideas can be used to solve a large number of bioinformatics problems. We feel that focusing on ideas has more intellectual value and represents a better long-term investment: protocols change quickly, but the computational ideas don't seem to.

We pursued a goal of presenting both the foundations of algorithms and the important results in bioinformatics under the same cover. A more thorough approach for a student would be to take an Introduction to Algorithms course followed by a Bioinformatics course, but this is often an unrealistic expectation in view of the heavy course load biologists have to take. To make bioinformatics ideas accessible to biologists we appeal to the innate algorithmic intuition of the student and try to avoid tedious proofs. The technical details are hidden unless they are absolutely necessary.[3]

This book covers both new and old areas in computational biology. Some topics, to our knowledge, have never been discussed in a textbook before, while others are relatively old-fashioned and describe some experimental approaches that are rarely used these days. The reason for including older topics is twofold. First, some of them still remain the best examples for introducing algorithmic ideas. Second, our goal is to show the progression of ideas in the field, with the implicit warning that hot areas in bioinformatics seem to come and go with alarming speed.

3. In some places we hide important computational and biological details and try to simplify the presentation. We will unavoidably be blamed later for "trivializing" bioinformatics.

One observation gained from teaching bioinformatics classes is that the interest of computer science students, who usually know little of biology, fades quickly when the students are introduced to biology without links to computational issues. The same happens to biologists if they are presented with seemingly unnecessary formalism with no links to real biological problems. To hold a student's interest, it is necessary to introduce biology and algorithms simultaneously. Our rather eclectic table of contents is a demonstration that attempts to reach this goal result in a somewhat interleaved organization of the material. However, we have tried to maintain a consistent algorithmic theme (e.g., graph algorithms) throughout each chapter.

Molecular biology and computer science are complex fields whose terminology and nomenclature can be formidable to the outsider. Bioinformatics merges the two fields, and adds a healthy dose of statistics, combinatorics, and other branches of mathematics. Like modern biologists who have to master the dense language of mathematics and computer science, mathematicians and computer scientists working in bioinformatics have to learn the language of biology. Although the question of who faces the bigger challenge is a topic hotly debated over pints of beer, this is not the first "invasion" of foreigners into biology; seventy years ago a horde of physicists infested biology labs, ultimately to revolutionize the field by deciphering the mystery of DNA.

Two influential scientists are credited with crossing the barrier between physics and biology: Max Delbrück and Erwin Schrödinger. Trained as physicists, their entrances into the field of biology were remarkably different. Delbrück, trained as an atomic physicist by Niels Bohr, quickly became an expert in genetics; in 1945 he was already teaching genetics to other biologists.[4] Schrödinger, on the other hand, never turned into a "certified" geneticist and remained somewhat of a biological dilettante. However, his book *What Is Life?*, published in 1944, was influential to an entire generation of physicists and biologists. Both James Watson (a biology student who wanted to be a naturalist) and Francis Crick (a physicist who worked on magnetic mines) switched careers to DNA science after reading Shrödinger's book. Another Nobel laureate, Sydney Brenner, even admitted to stealing a copy from the public library in Johannesburg, South Africa.

Like Delbrück and Schrödinger, there is great variety in the biological background of today's computer scientists-turned-bioinformaticians. Some of them have become experts in biology—though very few put on lab coats

4. Delbrück founded the famous phage genetics courses at Cold Spring Harbor Laboratory.

and perform experiments—while others remain biological dilettantes. Although there exists an opinion that every bioinformatician should be an expert in *both* biology and computer science, we are not sure that this is feasible. First, it takes a lot of work just to master one of the two, so perhaps understanding two in equal amounts is a bit much. Second, it is good to recall that the first pioneers of DNA science were, in fact, self-proclaimed dilettantes. James Watson knew almost no organic or physical chemistry before he started working on the double helix; Francis Crick, being a physicist, knew very little biology. Neither saw any need to know about (let alone memorize) the chemical structure of the four nucleotide bases when they discovered the structure of DNA.[5] When asked by Erwin Chargaff how they could possibly expect to resolve the structure of DNA without knowing the structures of its constituents, they responded that they could always look up the structures in a book if the need arose. Of course, they *understood* the physical principles behind a compound's structure.

The reality is that even the most biologically oriented bioinformaticians are experts only in some specific area of biology. Like Delbrück, who probably would never have passed an exam in biology in the 1930s (when zoology and botany remained the core of the mainstream biological curriculum), a typical modern-day bioinformatician is unlikely to pass the sequence of organic chemistry, biochemistry, and structural biochemistry classes that a "real" biologist has to take. The question of how much biology a good computer scientist–turned–bioinformatician has to know seems to be best answered with "enough to deeply understand the biological problem and to turn it into an adequate computational problem." This book provides a very brief introduction to biology. We do not claim that this is the best approach. Fortunately, an interested reader can use Watson's approach and look up the biological details in the books when the need arises, or read pages 1 through 1294 of Alberts and colleagues' (including Watson) book *Molecular Biology of the Cell* (3).

This book is what we, as computer scientists, believe that a modern biologist ought to know about computer science if he or she would be a successful researcher.

5. Accordingly, we do not present anywhere in this book the chemical structures of either nucleotides or amino acids. No algorithm in this book requires knowledge of their structure.

2 *Algorithms and Complexity*

This book is about how to design algorithms that solve biological problems. We will see how popular bioinformatics algorithms work and we will see what principles drove their design. It is important to understand how an algorithm works in order to be confident in its results; it is even more important to understand an algorithm's design methodology in order to identify its potential weaknesses and fix them.

Before considering any algorithms in detail, we need to define loosely what we mean by the word "algorithm" and what might qualify as one. In many places throughout this text we try to avoid tedious mathematical formalisms, yet leave intact the rigor and intuition behind the important concept.

2.1 What Is an Algorithm?

Roughly speaking, an *algorithm* is a sequence of instructions that one must perform in order to solve a well-formulated *problem*. We will specify problems in terms of their *inputs* and their *outputs*, and the algorithm will be the method of translating the inputs into the outputs. A well-formulated problem is unambiguous and precise, leaving no room for misinterpretation.

In order to solve a problem, some entity needs to carry out the steps specified by the algorithm. A human with a pen and paper would be able to do this, but humans are generally slow, make mistakes, and prefer not to perform repetitive work. A computer is less intelligent but can perform simple steps quickly and reliably. A computer cannot understand English, so algorithms must be rephrased in a programming language such as C or Java in order to give specific instructions to the processor. Every detail must be specified to the computer in exactly the right format, making it difficult to describe algorithms; trifling details that a person would naturally understand must be specified. If a computer were to put on shoes, one would need to tell it to find a pair that both matches and fits, to put the left shoe on the left foot, the right shoe on the right, and to tie the laces.[1] In this book, however, we prefer to simply leave it at "Put on a pair of shoes."

However, to understand how an algorithm works, we need some way of listing the steps that the algorithm takes, while being neither too vague nor too formal. We will use *pseudocode*, whose elementary operations are summarized below. Pseudocode is a language computer scientists often use to describe algorithms: it ignores many of the details that are required in a programming language, yet it is more precise and less ambiguous than, say, a recipe in a cookbook. Individually, the operations do not solve any particularly difficult problems, but they can be grouped together into minialgorithms called *subroutines* that do.

In our particular flavor of pseudocode, we use the concepts of *variables*, *arrays*, and *arguments*. A variable, written as x or $total$, contains some numerical value and can be assigned a new numerical value at different points throughout the course of an algorithm. An array of n elements is an ordered collection of n variables $a_1, a_2, \ldots, a_n$. We usually denote arrays by boldface letters like $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and write the individual elements as a_i where i is between 1 and n. An algorithm in pseudocode is denoted by a name, followed by the list of arguments that it requires, like $\text{MAX}(a, b)$ below; this is followed by the statements that describe the algorithm's actions. One can *invoke* an algorithm by passing it the appropriate values for its arguments. For example, $\text{MAX}(1, 99)$ would return the larger of 1 and 99. The operation **return** reports the result of the program or simply signals its end. Below are brief descriptions of the elementary commands that we use in the pseudocode throughout this book.[2]

Assignment

Format: $a \leftarrow b$

Effect: Sets the variable a to the value b.

1. It is surprisingly difficult to write an unambiguous set of instructions on how to tie a shoelace.
2. An experienced computer programmer might be confused by our not using "**end if**" or "**end for**", which is the conventional practice. We rely on indentation to demarcate blocks of pseudocode.

Example:
```
$b \leftarrow 2$
$a \leftarrow b$
```

Result: The value of a is 2

Arithmetic

Format: $a + b$, $a - b$, $a \cdot b$, a/b, a^b

Effect: Addition, subtraction, multiplication, division, and exponentiation of numbers.

Example:
```
DIST($x1, y1, x2, y2$)
1 $dx \leftarrow (x2 - x1)^2$
2 $dy \leftarrow (y2 - y1)^2$
3 **return** $\sqrt{(dx + dy)}$
```

Result: DIST$(x1, y1,x2, y2)$ computes the Euclidean distance between points with coordinates $(x1, y1)$ and $(x2, y2)$. DISTANCE$(0, 0, 3, 4)$ returns 5.

Conditional

Format:
```
**if** $A$ is true
    **B**
**else**
    **C**
```

Effect: If statement A is true, executes instructions **B**, otherwise executes instructions **C**. Sometimes we will omit "**else C**," in which case this will either execute **B** or not, depending on whether A is true.

Example:
```
MAX($a, b$)
1 **if** $a < b$
2     **return** $b$
3 **else**
4     **return** $a$
```

Result: MAX(a, b) computes the maximum of the numbers a and b. For example, MAX$(1, 99)$ returns 99.

`for` loops

Format:
```
for $i \leftarrow a$ to $b$
    B
```

Effect: Sets i to a and executes instructions $\mathbf{B}$. Sets i to $a+1$ and executes instructions $\mathbf{B}$ again. Repeats for $i = a+2, a+3, \ldots, b-1, b$.[3]

Example: SUMINTEGERS(n)
```
1 $sum \leftarrow 0$
2 for $i \leftarrow 1$ to $n$
3     $sum \leftarrow sum + i$
4 return $sum$
```

Result: SUMINTEGERS(n) computes the sum of integers from 1 to n. SUMINTEGERS(10) returns $1 + 2 + \cdots + 10 = 55$.

`while` loops

Format:
```
while $A$ is true
    B
```

Effect: Checks the condition A. If it is true, then executes instructions $\mathbf{B}$. Checks A again; if it's true, it executes $\mathbf{B}$ again. Repeats until A is not true.

Example: ADDUNTIL(b)
```
1 $i \leftarrow 1$
2 $total \leftarrow i$
3 while $total \leq b$
4     $i \leftarrow i + 1$
5     $total \leftarrow total + i$
6 return $i$
```

Result: ADDUNTIL(b) computes the smallest integer i such that $1 + 2 + \cdots + i$ is larger than b. For example, ADDUNTIL(25) returns 7, since

3. If a is larger than b, this loop operates in the reverse order: it sets i to a and executes instructions $\mathbf{B}$, then repeats for $i = a-1, a-2, \ldots, b+1, b$.

$1+2+\cdots+7=28$, which is larger than 25, but $1+2+\cdots+6=21$, which is smaller than 25.

Array access

Format: a_i

Effect: The ith number of array $\mathbf{a} = (a_1, \ldots a_i, \ldots a_n)$. For example, if $\mathbf{F} = (1, 1, 2, 3, 5, 8, 13)$, then $F_3 = 2$, and $F_4 = 3$.

Example:
```
FIBONACCI($n$)
1 $F_1 \leftarrow 1$
2 $F_2 \leftarrow 1$
3 for $i \leftarrow 3$ to $n$
4     $F_i \leftarrow F_{i-1} + F_{i-2}$
5 return $F_n$
```

Result: FIBONACCI(n) computes the nth Fibonacci number. FIBONACCI(8) returns 21.

While computer scientists are accustomed to the pseudocode jargon above, we fear that some biologists reading it might decide that this book is too cryptic and therefore useless. Although modern biologists deal with algorithms on a daily basis, the language they use to describe an algorithm might be closer to the language used in a cookbook, like the pumpkin pie recipe in figure 2.1. Accordingly, some bioinformatics books are written in this familiar lingo as an effort to make biologists feel at home with different bioinformatics concepts. Unfortunately, the cookbook language is insufficient to describe more complex algorithmic ideas that are necessary for even the simplest tools in bioinformatics. The problem is that natural languages are not suitable for communicating algorithmic ideas more complex than the pumpkin pie. Computer scientists have yet to invent anything better than pseudocode for this purpose, so we use it in this book.

To illustrate more concretely the distinction between pseudocode and an informal language, we can write an "algorithm" to create a pumpkin pie that mimics the recipe shown in figure 2.1. The admittedly contrived pseudocode below, MAKEPUMPKINPIE, is quite a bit more explicit.

$1\frac{1}{2}$ cups canned or cooked pumpkin
1 cup brown sugar, firmly packed
$\frac{1}{2}$ teaspoon salt
2 teaspoons cinnamon
1 teaspoon ginger
2 tablespoons molasses
3 eggs, slightly beaten
12 ounce can of evaporated milk
1 unbaked pie crust

Combine pumpkin, sugar, salt, ginger, cinnamon, and molasses. Add eggs and milk and mix thoroughly. Pour into unbaked pie crust and bake in hot oven (425 degrees Fahrenheit) for 40 to 45 minutes, or until knife inserted comes out clean.

Figure 2.1 A recipe for pumpkin pie.

```
MAKEPUMPKINPIE($pumpkin, sugar, salt, spices, eggs, milk, crust$)
1  PREHEATOVEN(425)
2  $filling$ ← MIXFILLING($pumpkin, sugar, salt, spices, eggs, milk$)
3  $pie$ ← ASSEMBLE($crust, filling$)
4  while knife inserted does not come out clean
5      BAKE($pie$)
6  output "Pumpkin pie is complete"
7  return $pie$
```

```
MIXFILLING($pumpkin, sugar, salt, spices, eggs, milk$)
 1  $bowl$ ← Get a bowl from cupboard
 2  PUT($pumpkin, bowl$)
 3  PUT($sugar, bowl$)
 4  PUT($salt, bowl$)
 5  PUT($spices, bowl$)
 6  STIR($bowl$)
 7  PUT($eggs, bowl$)
 8  PUT($milk, bowl$)
 9  STIR($bowl$)
10  $filling$ ← Contents of $bowl$
11  return $filling$
```

MAKEPUMPKINPIE *calls* (i.e., activates) the subroutine MIXFILLING, which uses **return** to return the pie filling. The operation **return** terminates the execution of the subroutine and returns a result to the routine that called it, in this case MAKEPUMPKINPIE. When the pie is complete, MAKEPUMPKINPIE notifies and returns the pie to whomever requested it. The entity pie in MAKEPUMPKINPIE is a *variable* that represents the pie in the various stages of cooking.

A subroutine, such as MIXFILLING, will normally need to return the result of some important calculation. However, in some cases the inputs to the subroutine might be invalid (e.g., if you gave the algorithm watermelon instead of pumpkin). In these cases, a subroutine may return no value at all and **output** a suitable error message. When an algorithm is finished calculating a result, it naturally needs to output that result and stop executing. The operation **output** displays information to an interested user.[4]

A subtle observation is that MAKEPUMPKINPIE does not in fact make a pumpkin pie, but only tells you *how* to make a pumpkin pie at a fairly abstract level. If you were to build a machine that follows these instructions, you would need to make it specific to a particular kitchen and be tirelessly explicit in all the steps (e.g., how many times and how hard to stir the filling, with what kind of spoon, in what kind of bowl, etc.) This is exactly the difference between pseudocode (the abstract sequence of steps to solve a well-formulated computational problem) and computer code (a set of detailed instructions that one particular computer will be able to perform). We reiterate that the function of pseudocode in this book is only to communicate the idea behind an algorithm, and that to actually use an algorithm in this book you would need to turn the pseudocode into computer code, which is not always easy.

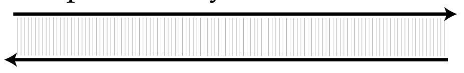
We will often avoid tedious details in the specification of an algorithm by specifying parts of it in English (e.g., "Get a bowl from cupboard"), using operations that are not listed in our description of pseudocode, or by omitting certain details that are unimportant. We assume that, in the case of confusion, the reader will fill in the details using pseudocode operations in a sensible way.

4. Exactly how this is done remains beyond the scope of pseudocode and really does not matter.

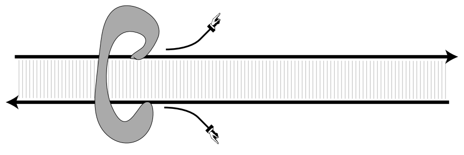
2.2 Biological Algorithms versus Computer Algorithms

Nature uses algorithm-like procedures to solve biological problems, for example, in the process of *DNA replication*. Before a cell can divide, it must first make a complete copy of all its genetic material.

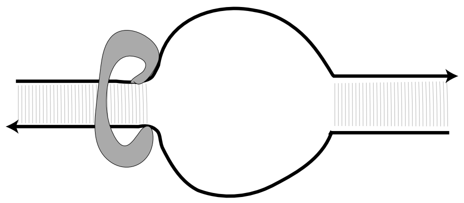
DNA replication proceeds in phases, each of which requires an elaborate cooperation between different types of molecules. For the sake of simplicity, we describe the replication process as it occurs in bacteria, rather than the replication process in humans or other mammals, which is quite a bit more involved. The basic mechanism was proposed by James Watson and Francis Crick in the early 1950s, but could only be verified through the ingenious Meselson-Stahl experiment of 1957. The replication process starts from a pair of complementary[5] strands of DNA and ends up with two pairs of complementary strands.[6]



1. A molecular machine (in other words, a protein complex) called a *DNA helicase*, binds to the DNA at certain positions called *replication origins*.

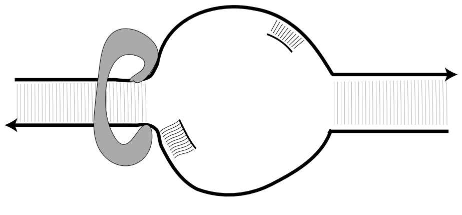


2. Helicase wrenches apart the two strands of DNA, creating a so-called *replication fork*. The two strands are complementary and run in opposite directions (one strand is denoted $3' \rightarrow 5'$, the other $5' \rightarrow 3'$). Two other molecular machines, *topoisomerase* and *single-strand binding protein* (not shown) bind to the single strands to help relieve the instability of single-stranded DNA.

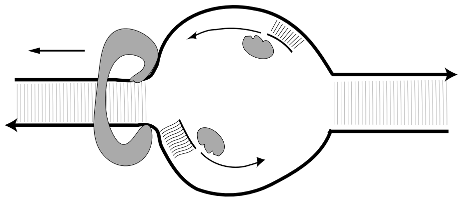


5. Complementarity is described in chapter 3.
6. It is possible that computer scientists will spontaneously abort due to the complexity of this system. While biologists feel at home with a description of DNA replication, computer scientists may find it too overloaded with unfamiliar terms. This example only illustrates what biologists use as "pseudocode;" the terms here are not crucial for understanding the rest of the book.

3. *Primers*, which are short single strands of RNA, are synthesized by a protein complex called *primase* and latch on to specific positions in the newly opened strands, providing an anchor for the next step. Without primers, the next step cannot begin.

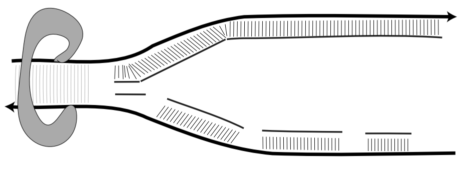


4. A *DNA polymerase* (yet another molecular machine) binds to each freshly separated *template* strand of the DNA; the DNA polymerase traverses the parent strands only in the $3' \rightarrow 5'$ direction. Therefore, the DNA polymerases attached to the two DNA strands move in opposite directions.

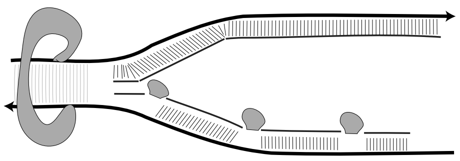


5. At each nucleotide, DNA polymerase matches the template strand's nucleotide with the complementary base, and adds it to the growing synthesized chain. Prior to moving to the next nucleotide, DNA polymerase checks to ensure that the correct base has been paired at the current position; if not, it removes the incorrect base and retries.

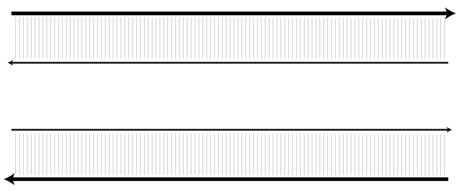
   Since DNA polymerase can only traverse DNA in the $3' \rightarrow 5'$ direction, and since the two strands of DNA run in opposite directions, only one strand of the template DNA can be used by polymerase to continuously synthesize its complement; the other strand requires occasional stopping and restarting. This results in short segments called *Okazaki fragments*.



6. Another molecular machine, *DNA ligase*, repairs the gaps in the newly synthesized DNA's backbone, effectively linking together all Okazaki fragments into a single molecule and cleaning any breaks in the primary strand.



7. When all the DNA has been copied in such a manner, the original strands separate, so that two pairs of DNA strands are formed, each pair consisting of one old and one newly synthesized strand.



Obviously, an astounding amount of molecular logistics is required to ensure completely accurate DNA replication: DNA helicase separates strands, DNA polymerase ensures proper complementarity, and so on. However, in terms of the logic of the process, none of this complicated molecular machinery actually matters—to mimic this process in an algorithm we simply need to take a string which represents the DNA and return a copy of it.

String Duplication Problem:
Given a string of letters, return a copy.

Input: A string $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ of length n, as an array of characters.

Output: A string representing a copy of $\mathbf{s}$.

Of course, this is a particularly easy problem to solve and yields absolutely no interesting algorithmic intuition. However it is still illustrative to write the pseudocode. The STRINGCOPY program below uses the string $\mathbf{t}$ to hold a copy of the input string $\mathbf{s}$, and returns the result $\mathbf{t}$.

```
STRINGCOPY($\mathbf{s}, n$)
1  for $i \leftarrow 1$ to $n$
2      $t_i \leftarrow s_i$
3  return $\mathbf{t}$
```

While STRINGCOPY is a trivial algorithm, the number of operations that a real computer performs to copy a string is surprisingly large. For one particular computer architecture, we may end up issuing thousands of instructions to a computer processor. Computer scientists distance themselves from this complexity by inventing programming languages that allow one to ignore many of these details. Biologists have not yet invented a similar "language" to describe biological algorithms working in the cell.

The amount of "intelligence" that the simplest organism, such as a bacterium, exhibits to perform any routine task—including replication—is amazing. Unlike STRINGCOPY, which only performs abstract operations, the bacterium really *builds* new DNA using materials that are floating near the replication fork. What would happen if it ran out? To prevent this, a bacterium examines the surroundings, imports new materials from outside, or moves off to forage for food. Moreover, it waits to begin copying its DNA until sufficient materials are available. These observations, let alone the coordination between the individual molecules, lead us to wonder whether even the most sophisticated computer programs can match the complicated behavior displayed by even a single-celled organism.

2.3 The Change Problem

The first—and often the most difficult—step in solving a computational problem is to identify precisely what the problem is. By using the techniques described in this book, you can then devise an algorithm that solves it. However, you cannot stop there. Two important questions to ask are: "Does it work correctly?" and "How long will it take?" Certainly you would not be satisfied with an algorithm that only returned correct results half the time, or took 600 years to arrive at an answer. Establishing reasonable expectations for an algorithm is an important step in understanding how well the algorithm solves the problem, and whether or not you trust its answer.

A problem describes a class of computational tasks. A problem *instance* is one particular input from that class. To illustrate the difference between a problem and an instance of a problem, consider the following example. You find yourself in a bookstore buying a fairly expensive pen for \$4.23 which you pay for with a \$5 bill (fig. 2.2). You would be due 77 cents in change, and the cashier now makes a decision as to exactly how you get it.[7] You would be annoyed at a fistful of 77 pennies or 15 nickels and 2 pennies, which raises the question of how to make change in the least annoying way. Most cashiers try

7. A penny is the smallest denomination in U.S. currency. A dollar is 100 pennies, a quarter is 25 pennies, a dime is 10, and a nickel is 5.

Figure 2.2 The subtle difference between a problem (top) and an instance of a problem (bottom).

to minimize the number of coins returned for a particular quantity of change. The example of 77 cents represents an instance of the United States Change problem, which we can formulate as follows.[8]

8. Though this problem is not at particularly relevant to biology, it serves as a useful tool to illustrate a number of different algorithmic approaches.

United States Change Problem:
Convert some amount of money into the fewest number of coins.

Input: An amount of money, M, in cents.

Output: The smallest number of quarters q, dimes d, nickels n, and pennies p whose values add to M (i.e., $25q + 10d + 5n + p = M$ and $q + d + n + p$ is as small as possible).

The algorithm that is used by cashiers all over the United States to solve this problem is simple:

```
USCHANGE($M$)
1  while $M > 0$
2      $c \leftarrow$ Largest coin that is smaller than (or equal to) $M$
3      Give coin with denomination $c$ to customer
4      $M \leftarrow M - c$
```

A slightly more detailed description of this algorithm is:

```
USCHANGE($M$)
1  Give the integer part of $M/25$ quarters to customer.
2  Let $remainder$ be the remaining amount due the customer.
3  Give the integer part of $remainder/10$ dimes to customer.
4  Let $remainder$ be the remaining amount due the customer.
5  Give the integer part of $remainder/5$ nickels to customer.
6  Let $remainder$ be the remaining amount due the customer.
7  Give $remainder$ pennies to customer.
```

A pseudocode version of the above algorithm is:

```
USCHANGE($M$)
1  $r \leftarrow M$
2  $q \leftarrow r/25$
3  $r \leftarrow r - 25 \cdot q$
4  $d \leftarrow r/10$
5  $r \leftarrow r - 10 \cdot d$
6  $n \leftarrow r/5$
7  $r \leftarrow r - 5 \cdot n$
8  $p \leftarrow r$
9  **return** $(q, d, n, p)$
```

When $r/25$ is not a whole number, we take the *floor* of $r/25$, that is, the integer part[9] of $r/25$. When the cashier runs USCHANGE(77), it returns three quarters, no dimes or nickels, and two pennies, which is the desired result (there is no other combination that has fewer coins and adds to 77 cents). First, the variable r is set to 77. Then q, the number of quarters, is set to the value 3, since $\lfloor 77/25 \rfloor = 3$. The variable r is then updated in line 3 to be equal to 2, which is the difference between the amount of money the cashier is changing (77 cents) and the three quarters he has chosen to return. The variables d and n—dimes and nickels, respectively—are subsequently set to 0 in lines 4 and 6, since $\lfloor 2/10 \rfloor = 0$ and $\lfloor 2/5 \rfloor = 0$; r remains unchanged on lines 5 and 7 since d and n are 0. Finally, the variable p, which stands for "pennies," is set to 2, which is the amount in variable r. The values of four variables—q, d, n, and p—are returned as the solution to the problem.[10]

2.4 Correct versus Incorrect Algorithms

As presented, USCHANGE lacks elegance and generality. Inherent in the algorithm is the assumption that you are changing United States currency, and that the cashier has an unlimited supply of each denomination—generally quarters are harder to come by than dimes. We would like to generalize the algorithm to accommodate different denominations without requiring a completely new algorithm for each one. To accomplish this, however, we must first generalize the problem to provide the algorithm with the denominations that it can change M into. The new Change problem below assumes

9. The floor of 77/25, denoted $\lfloor 3.08 \rfloor$, is 3.
10. Inevitably, an experienced computer programmer will wring his or her hands at returning multiple, rather than single, answers from a subroutine. This is not actually a problem, but how this really works inside a computer is irrelevant to our discussion of algorithms.

that there are d denominations, rather than the four of the previous problem. These denominations are represented by an array $\mathbf{c} = (c_1, \ldots, c_d)$. For simplicity, we assume that the denominations are given in decreasing order of value. For example, in the case of the United States Change problem, $\mathbf{c} = (25, 10, 5, 1)$, whereas in the European Union Change problem, $\mathbf{c} = (20, 10, 5, 2, 1)$.

Change Problem:
Convert some amount of money M into given denominations, using the smallest possible number of coins.

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

We can solve this problem with an even simpler five line pseudocode than the previous USCHANGE algorithm.[11]

```
BETTERCHANGE($M, \mathbf{c}, d$)
1  $r \leftarrow M$
2  for $k \leftarrow 1$ to $d$
3        $i_k \leftarrow r/c_k$
4        $r \leftarrow r - c_k \cdot i_k$
5  return $(i_1, i_2, \ldots, i_d)$
```

We say that an algorithm is *correct* when it can translate every input instance into the correct output. An algorithm is *incorrect* when there is at least one input instance for which the algorithm does not produce the correct output. At first this seems unbalanced: if an algorithm fails on even a single input instance, then the whole algorithm is judged incorrect. This reflects a critical, yet healthy, pessimism that you should maintain when designing an algorithm: unless you can justify that an algorithm always returns correct results, you should consider it to be wrong.[12]

11. This is a trap! Try to figure out why this is wrong. That is, find some set of inputs for which this new algorithm does not return the correct answer.
12. Some problems are so difficult, however, that no practical algorithm that is correct has been found. Often, researchers rely on *approximation* algorithms (described in chapter 5) to produce

BETTERCHANGE is not a correct algorithm. Suppose we were changing 40 cents into coins with denominations of $c_1 = 25$, $c_2 = 20$, $c_3 = 10$, $c_4 = 5$, and $c_5 = 1$. BETTERCHANGE would incorrectly return 1 quarter, 1 dime, and 1 nickel, instead of 2 twenty-cent pieces. As contrived as this may seem, in 1875 a twenty-cent coin existed in the United States. Between 1865 and 1889, the U.S. Treasury even produced three-cent coins. How sure can we be that BETTERCHANGE returns the minimal number of coins for our modern currency, or for foreign countries? Determining the conditions under which BETTERCHANGE is a correct algorithm is left as a problem at the end of this chapter.

To correct the BETTERCHANGE algorithm, we could consider every possible combination of coins with denominations $c_1, c_2, \ldots, c_d$ that adds to M, and return the combination with the fewest. We do not need to consider any combinations with $i_1 > M/c_1$, or $i_2 > M/c_2$ (in general, i_k should not exceed M/c_k), because we would otherwise be returning an amount of money strictly larger than M. The pseudocode below uses the symbol $\sum$, meaning summation: $\sum_{i=1}^{m} a_i = a_1 + a_2 + a_3 + \cdots + a_m$. The pseudocode also uses the notion of "infinity" (∞) as an initial value for $smallestNumberOfCoins$; there are a number of ways to carry this out in a real computer, but the details are not important here.

```
BRUTEFORCECHANGE($M, \mathbf{c}, d$)
1  $smallestNumberOfCoins \leftarrow \infty$
2  for each $(i_1, \ldots, i_d)$ from $(0, \ldots, 0)$ to $(M/c_1, \ldots, M/c_d)$
3      $valueOfCoins \leftarrow \sum_{k=1}^{d} i_k c_k$
4      if $valueOfCoins = M$
5          $numberOfCoins \leftarrow \sum_{k=1}^{d} i_k$
6          if $numberOfCoins < smallestNumberOfCoins$
7              $smallestNumberOfCoins \leftarrow numberOfCoins$
8              bestChange $\leftarrow (i_1, i_2, \ldots, i_d)$
9  return (bestChange)
```

Line 2 iterates over every combination $(i_1, i_2, \ldots, i_d)$ of the d indices,[13] and

results. The implicit acknowledgment that we make in using those types of algorithms is that some better solution probably exists, but we were unable to find it.

13. An *array index* points to an element in an array. For example, if $\mathbf{c} = \{1, 1, 2, 3, 5, 8, 13, 21, 34\}$, then the index of element 8 is 6, while the index of element 34 is 9.

stops when it has reached $(M/c_1, M/c_2, \ldots, M/c_{d-1}, M/c_d)$:

$$\begin{array}{lllllll}
(& 0, & 0, & \ldots, & 0, & 0 &) \\
(& 0, & 0, & \ldots, & 0, & 1 &) \\
(& 0, & 0, & \ldots, & 0, & 2 &) \\
 & & & \vdots & & & \\
(& 0, & 0, & \ldots, & 0, & \frac{M}{c_d} &) \\
(& 0, & 0, & \ldots, & 1, & 0 &) \\
(& 0, & 0, & \ldots, & 1, & 1 &) \\
(& 0, & 0, & \ldots, & 1, & 2 &) \\
 & & & \vdots & & & \\
(& 0, & 0, & \ldots, & 1, & \frac{M}{c_d} &) \\
 & & & \vdots & & & \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}} - 1, & 0 &) \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}} - 1, & 1 &) \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}} - 1, & 2 &) \\
 & & & \vdots & & & \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}} - 1, & \frac{M}{c_d} &) \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}}, & 0 &) \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}}, & 1 &) \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}}, & 2 &) \\
 & & & \vdots & & & \\
(& \frac{M}{c_1}, & \frac{M}{c_2}, & \ldots, & \frac{M}{c_{d-1}}, & \frac{M}{c_d} &)
\end{array}$$

We have omitted some details from the BRUTEFORCECHANGE algorithm. For example, there is no pseudocode operation that performs summation of d integers at one time, nor does it include any way to iterate over every combination of d indices. These subroutines are left as problems at the end of this chapter because they are instructive to work out in detail. We have made the hidden assumption that given any set of denominations we can change any amount of money M. This may not be true, for example in the (unlikely) case that the monetary system has no pennies (that is, $c_d > 1$).

How do we know that BRUTEFORCECHANGE does not suffer from the same problem as BETTERCHANGE did, namely that some input instance returns an incorrect result? Since BRUTEFORCECHANGE explores *all possible* combinations of denominations, it will eventually come across an optimal solution and record it as such in the **bestChange** array. Any combination of coins that adds to M must have at least as many coins as the optimal combination, so BRUTEFORCECHANGE will never overwrite **bestChange** with a suboptimal solution.

We revisit the Change problem in future chapters to improve on this solution. So far we have answered only one of the two important algorithmic questions ("Does it work?", but not "How fast is it?"). We shall see that BRUTEFORCECHANGE is not particularly speedy.

2.5 Recursive Algorithms

Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

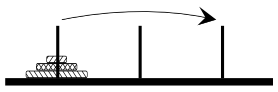
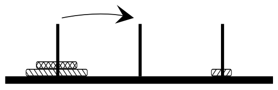
The *Towers of Hanoi* puzzle, introduced in 1883 by a French mathematician, consists of three pegs, which we label from left to right as 1, 2, and 3, and a number of disks of decreasing radius, each with a hole in the center. The disks are initially stacked on the left peg (peg 1) so that smaller disks are on top of larger ones. The game is played by moving one disk at a time between pegs. You are only allowed to place smaller disks on top of larger ones, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3.

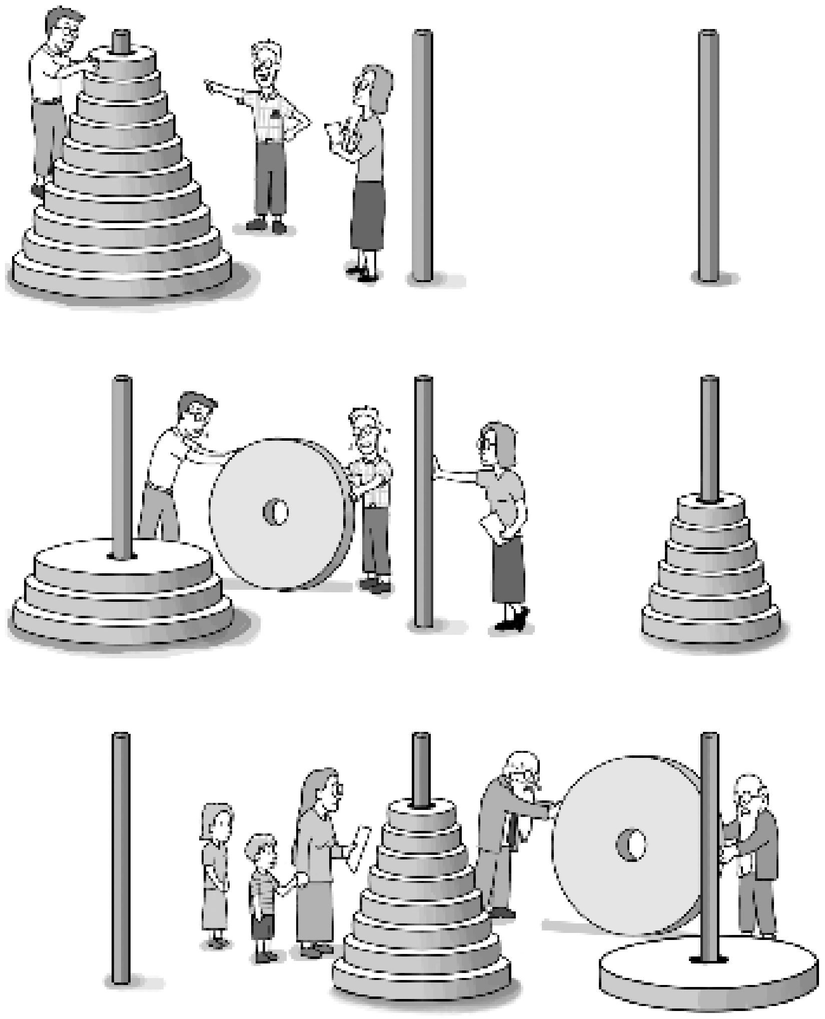
Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

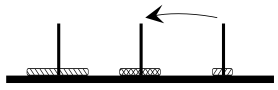
Input: An integer n.

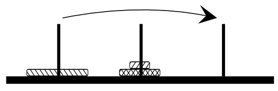
Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.

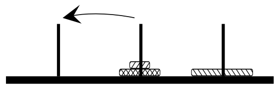
Solving the puzzle with one disk is easy: move the disk to the right peg. The two-disk puzzle is not much harder: move the small disk to the middle peg, then the large disk to the right peg, then the small disk to the right peg to rest on top of the large disk. The three-disk puzzle is somewhat harder, but the following sequence of seven moves solves it:

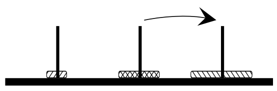
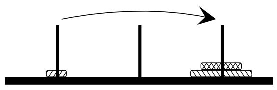
- Move disk from peg 1 to peg 3 
- Move disk from peg 1 to peg 2 



- Move disk from peg 3 to peg 2 

- Move disk from peg 1 to peg 3 

- Move disk from peg 2 to peg 1 

- Move disk from peg 2 to peg 3 
- Move disk from peg 1 to peg 3 

Now we will figure out how many steps are required to solve a four-disk puzzle. You cannot complete this game without moving the largest disk. However, in order to move the largest disk, we first had to move all the smaller disks to an empty peg. If we had four disks instead of three, then we would first have to move the top three to an empty peg (7 moves), then move the largest disk (1 move), then again move the three disks from their temporary peg to rest on top of the largest disk (another 7 moves). The whole procedure will take $7 + 1 + 7 = 15$ moves. More generally, to move a stack of size n from the left to the right peg, you first need to move a stack of size $n - 1$ from the left to the middle peg, and then from the middle peg to the right peg once you have moved the nth disk to the right peg. To move a stack of size $n - 1$ from the middle to the right, you first need to move a stack of size $n - 2$ from the middle to the left, then move the $(n - 1)$th disk to the right, and then move the stack of $n - 2$ from the left to the right peg, and so on.

At first glance, the Towers of Hanoi problem looks difficult. However, the following recursive algorithm solves the Towers of Hanoi problem with n disks. The iterative version of this algorithm is more difficult to write and analyze, so we do not present it here.

```
HANOITOWERS($n, fromPeg, toPeg$)
1 if $n = 1$
2     output "Move disk from peg $fromPeg$ to peg $toPeg$"
3     return
4 $unusedPeg \leftarrow 6 - fromPeg - toPeg$
5 HANOITOWERS($n - 1, fromPeg, unusedPeg$)
6 output "Move disk from peg $fromPeg$ to peg $toPeg$"
7 HANOITOWERS($n - 1, unusedPeg, toPeg$)
8 return
```

The variables $fromPeg$, $toPeg$, and $unusedPeg$ refer to the three different pegs so that HANOITOWERS$(n, 1, 3)$ moves n disks from the first peg to the third peg. The variable $unusedPeg$ represents which of the three pegs can

Table 2.1 The result of $6 - fromPeg - toPeg$ for all possible values of $fromPeg$ and $toPeg$.

$fromPeg$	$toPeg$	$unusedPeg$
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

serve as a temporary destination for the first $n-1$ disks. Note that $fromPeg+toPeg+unusedPeg$ is always equal to $1+2+3 = 6$, so the value of the variable $unusedPeg$ can be computed as $6 - fromPeg - toPeg$ which is determined in line 4 (see table 2.1). The subsequent statements (lines 5–7) then solve the smaller problem of moving the stack of size $n-1$ first to the temporary space, moving the largest disk, and then moving the $n-1$ small disks to the final destination. Note that we do not have to specify *which* disk the player should move from $fromPeg$ to $toPeg$: it is always the top disk currently residing on $fromPeg$ that gets moved.

Although the solution can be expressed in 8 lines of pseudocode, it requires a surprisingly long time to run. To solve a five-disk tower requires 31 moves, but to solve a hundred-disk tower would require more moves than there are atoms in the universe. The fast growth of the number of moves that HANOITOWERS requires is easy to see by noticing that every time HANOITOWERS$(n, 1, 3)$ is called, it calls itself twice for $n-1$, which in turn triggers four calls for $n-2$, and so on. We can illustrate this situation in a *recursion tree*, which is shown in figure 2.3. A call to HANOITOWERS$(4, 1, 3)$ results in calls HANOITOWERS$(3, 1, 2)$ and HANOITOWERS$(3, 2, 3)$; each of these results in calls to HANOITOWERS$(2, 1, 3)$, HANOITOWERS$(2, 3, 2)$ and HANOITOWERS$(2, 2, 1)$, HANOITOWERS$(2, 1, 3)$, and so on. Each call to the subroutine HANOITOWERS requires some amount of time, so we would like to know how much time the algorithm will take. This is determined in section 2.7.

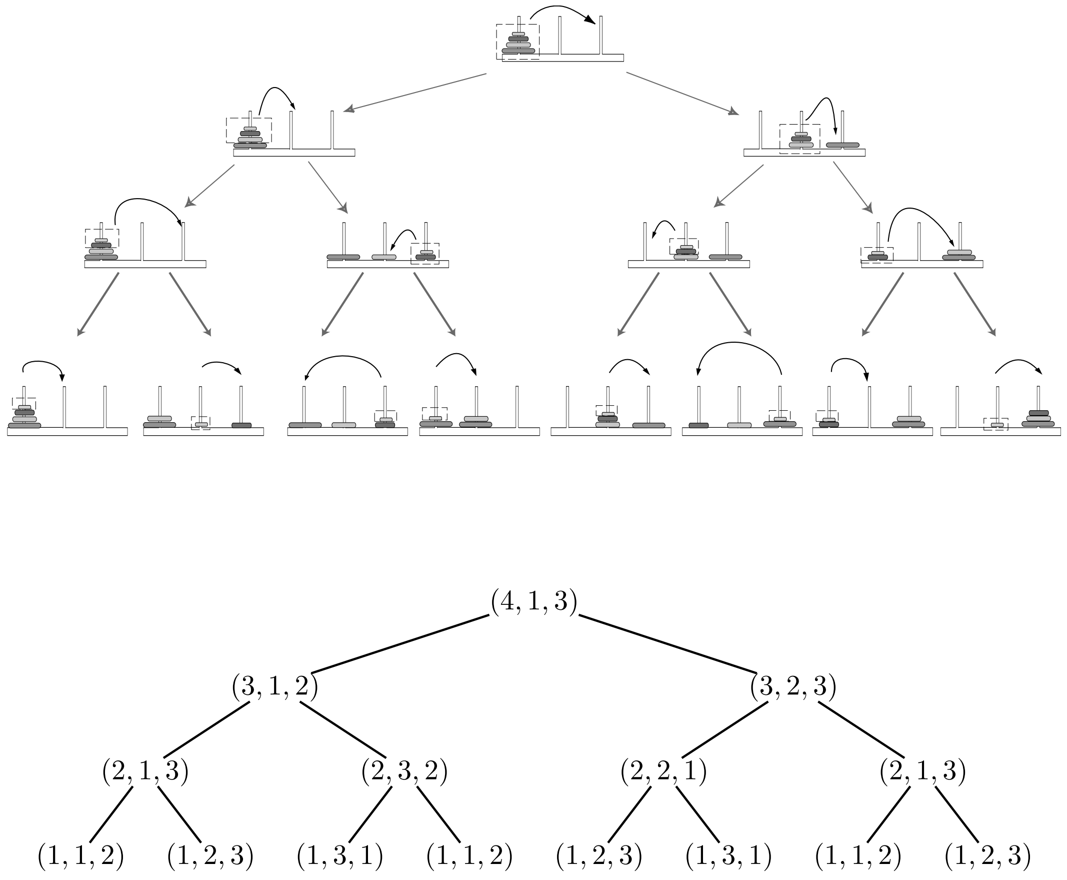


Figure 2.3 The recursion tree for a call to HANOITOWERS $(4, 1, 3)$, which solves the Towers of Hanoi problem of size 4. At each point in the tree, (i, j, k) stands for HANOITOWERS (i, j, k).

2.6 Iterative versus Recursive Algorithms

Recursive algorithms can often be rewritten to use iterative loops instead, and vice versa; it is a matter of elegance and clarity that dictates which technique is easier to use. Consider the problem of sorting a list of integers into ascending order.

Sorting Problem:
Sort a list of integers.

> **Input:** A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.
>
> **Output:** Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

The following algorithm, called SELECTIONSORT, is a naive but simple iterative method to solve the Sorting problem. First, SELECTIONSORT finds the smallest element in $\mathbf{a}$, and moves it to the first position by swapping it with whatever happens to be in the first position (i.e., a_1). Next, SELECTIONSORT finds the second smallest element in $\mathbf{a}$, and moves it to the second position, again by swapping with a_2. At the ith iteration, SELECTIONSORT finds the ith smallest element in $\mathbf{a}$, and moves it to the ith position. This is an intuitive approach at sorting, but is not the best-known one. If $\mathbf{a} = (7, 92, 87, 1, 4, 3, 2, 6)$, SELECTIONSORT$(\mathbf{a}, 8)$ takes the following seven steps:

$$(\overleftrightarrow{7, 92, 87, 1}, 4, 3, 2, 6)$$

$$(1, \overleftrightarrow{92, 87, 7, 4, 3, 2}, 6)$$

$$(1, 2, \overleftrightarrow{87, 7, 4, 3}, 92, 6)$$

$$(1, 2, 3, \overleftrightarrow{7, 4}, 87, 92, 6)$$

$$(1, 2, 3, 4, \overleftrightarrow{7, 87, 92, 6})$$

$$(1, 2, 3, 4, 6, \overleftrightarrow{87, 92, 7})$$

$$(1, 2, 3, 4, 6, 7, \overleftrightarrow{92, 87})$$

$$(1, 2, 3, 4, 6, 7, 87, 92)$$

```
SELECTIONSORT($\mathbf{a}, n$)
1  for $i \leftarrow 1$ to $n - 1$
2      $a_j \leftarrow$ Smallest element among $a_i, a_{i+1}, \ldots a_n$.
3      Swap $a_i$ and $a_j$
4  return $\mathbf{a}$
```

Line 2 of SELECTIONSORT finds the smallest element over all elements of $\mathbf{a}$ that come after i, and fits nicely into a subroutine as follows. The subroutine

INDEXOFMIN($\mathbf{array}, first, last$) works with **array** and returns the index of the smallest element between positions $first$ and $last$ by examining each element from $array_{first}$ to $array_{last}$.

```
INDEXOFMIN($\mathbf{array}, first, last$)
1  $index \leftarrow first$
2  for $k \leftarrow first + 1$ to $last$
3      if $array_k < array_{index}$
4          $index \leftarrow k$
5  return $index$
```

For example, if $\mathbf{a} = (7, 92, 87, 1, 4, 3, 2, 6)$, then INDEXOFMIN($\mathbf{a}, 1, 8$) would be 4, since $a_4 = 1$ is smaller than any other element in $(a_1, a_2, \ldots, a_8)$. Similarly, INDEXOFMIN($\mathbf{a}, 5, 8$) would be 7, since $a_7 = 2$ is smaller than any other element in (a_5, a_6, a_7, a_8). We can now write SELECTIONSORT using this subroutine.

```
SELECTIONSORT($\mathbf{a}, n$)
1  for $i \leftarrow 1$ to $n - 1$
2      $j \leftarrow$ INDEXOFMIN($\mathbf{a}, i, n$)
3      Swap elements $a_i$ and $a_j$
4  return a
```

To illustrate the similarity between recursion and iteration, we could instead have written SELECTIONSORT recursively (reusing INDEXOFMIN from above):

```
RECURSIVESELECTIONSORT($\mathbf{a}, first, last$)
1  if $first < last$
2      $index \leftarrow$ INDEXOFMIN($\mathbf{a}, first, last$)
3      Swap $a_{first}$ with $a_{index}$
4      $\mathbf{a} \leftarrow$ RECURSIVESELECTIONSORT($\mathbf{a}, first + 1, last$)
5  return a
```

In this case, RECURSIVESELECTIONSORT($\mathbf{a}, 1, n$) performs exactly the same operations as SELECTIONSORT($\mathbf{a}, n$).

It may seem contradictory at first that RECURSIVESELECTIONSORT calls itself to get an answer, but the key to understanding this algorithm is to realize that each time it is called, it works on a smaller set of elements from the list until it reaches the end of the list; at the end, it no longer needs to recurse.

The reason that the recursion does not continue indefinitely is because the algorithm works toward a point at which it "bottoms out" and no longer needs to recurse—in this case, when $first = last$.

As convoluted as it may seem at first, recursion is often the most natural way to solve many computational problems as it was in the Towers of Hanoi problem, and we will see many recursive algorithms in the coming chapters. However, recursion can often lead to very inefficient algorithms, as this next example shows.

The Fibonacci sequence is a mathematically important, yet very simple, progression of numbers. The series was first studied in the thirteenth century by the early Italian mathematician Leonardo Pisano Fibonacci, who tried to compute the number of offspring of a pair of rabbits over the course of a year (fig. 2.4). Fibonacci reasoned that one pair of adult rabbits could create a new pair of rabbits in about the same time that it takes bunnies to grow into adults. Thus, in any given period, each pair of adult rabbits produces a new pair of baby rabbits, and all baby rabbits grow into adult rabbits.[14] If we let F_n represent the number of rabbits in period n, then we can determine the value of F_n in terms of F_{n-1} and F_{n-2}. The number of adult rabbits at time period n is equal to the number of rabbits (adult and baby) in the previous time period, or F_{n-1}. The number of baby rabbits at time period n is equal to the number of adult rabbits in F_{n-1}, which is F_{n-2}. Thus, the total number of rabbits at time period n is the number of adults plus the number of babies, that is, $F_n = F_{n-1} + F_{n-2}$, with $F_1 = F_2 = 1$. Consider the following problem:

Fibonacci Problem:
Calculate the nth Fibonacci number.

Input: An integer n.

Output: The nth Fibonacci number $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$).

The simplest recursive algorithm, shown below, calculates F_n by calling itself to compute F_{n-1} and F_{n-2}. As figure 2.5 shows, this approach results in a large amount of duplicated effort: in calculating F_{n-1} we find the value

14. Fibonacci faced the challenge of adequately formulating the problem he was studying, one of the more difficult parts of bioinformatics research. The Fibonacci view of rabbit life is overly simplistic and inadequate: in particular, rabbits never die in his model. As a result, after just a few generations, the number of rabbits will be larger than the number of atoms in the universe.

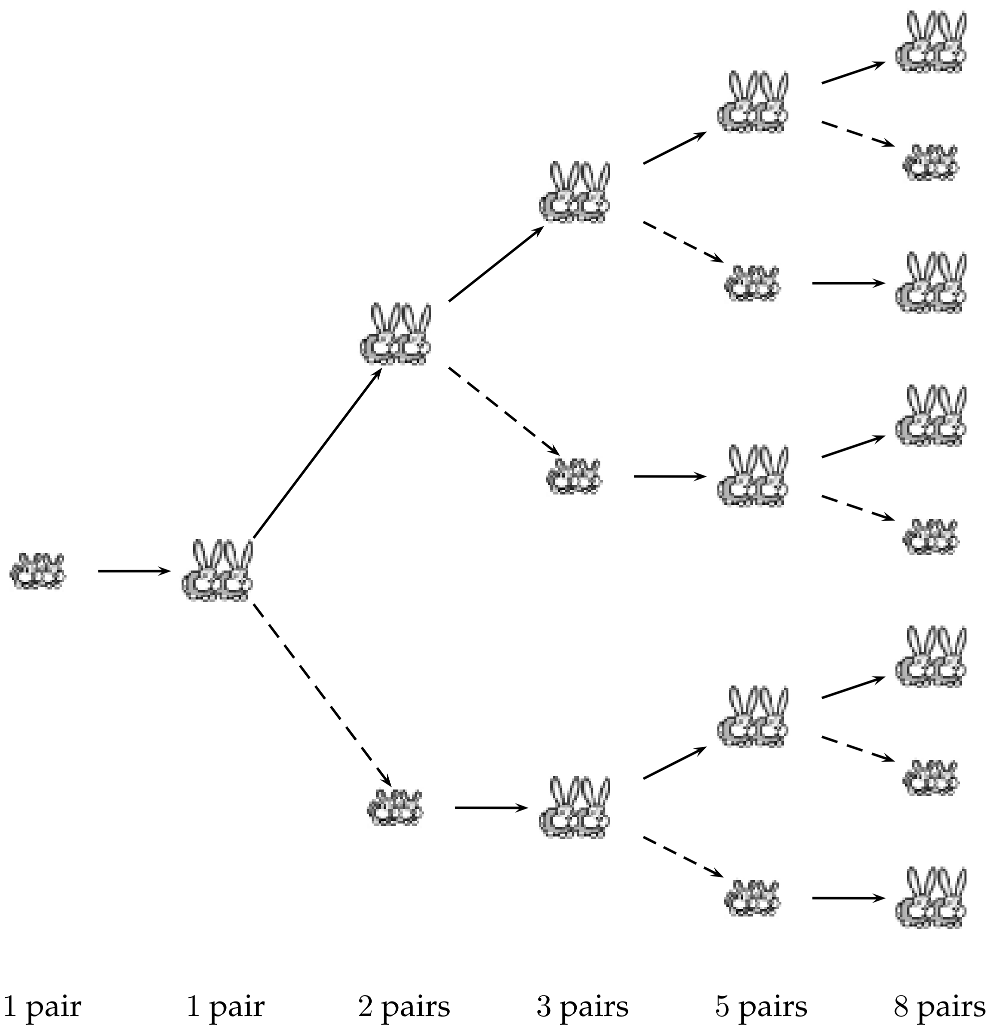


Figure 2.4 Fibonacci's model of rabbit expansion. A dashed line from a pair of big rabbits to a pair of little rabbits means that the pair of adult rabbits had bunnies.

of F_{n-2}, but we calculate it again from scratch in order to determine F_n. Therefore, most of the effort in this algorithm is wasted recomputing values that are already known.

```
RECURSIVEFIBONACCI($n$)
1  if $n = 1$ or $n = 2$
2      return 1
3  else
4      $a \leftarrow$ RECURSIVEFIBONACCI($n - 1$)
5      $b \leftarrow$ RECURSIVEFIBONACCI($n - 2$)
6      return $a + b$
```

However, by using an array to save previously computed Fibonacci numbers, we can calculate the nth Fibonacci number without repeating work.

```
FIBONACCI($n$)
1  $F_1 \leftarrow 1$
2  $F_2 \leftarrow 1$
3  for $i \leftarrow 3$ to $n$
4      $F_i \leftarrow F_{i-1} + F_{i-2}$
5  return $F_n$
```

In the language of the next section, FIBONACCI is a linear-time algorithm, while RECURSIVEFIBONACCI is an exponential-time algorithm. What this example has shown is not that an iterative algorithm is superior to a recursive algorithm, but that the two methods may lead to algorithms that require different amounts of time to solve the same problem instance.

2.7 Fast versus Slow Algorithms

Real computers require a certain amount of time to perform an operation such as addition, subtraction, or testing the conditions in a **while** loop. A supercomputer might take 10^{-9} second to perform an addition, while a hand calculator might take 10^{-5} second. Suppose that you had a computer that took 10^{-9} second to perform an elementary operation such as addition, and that you knew how many operations a particular algorithm would perform. You could estimate the running time of the algorithm simply by taking the product of the number of operations and the time per operation. However, computing devices are constantly improving, leading to a decreasing time per operation, so your notion of the running time would soon be outdated. Rather than computing an algorithm's running time on every computer, we rely on the total number of operations that the algorithm performs to de-

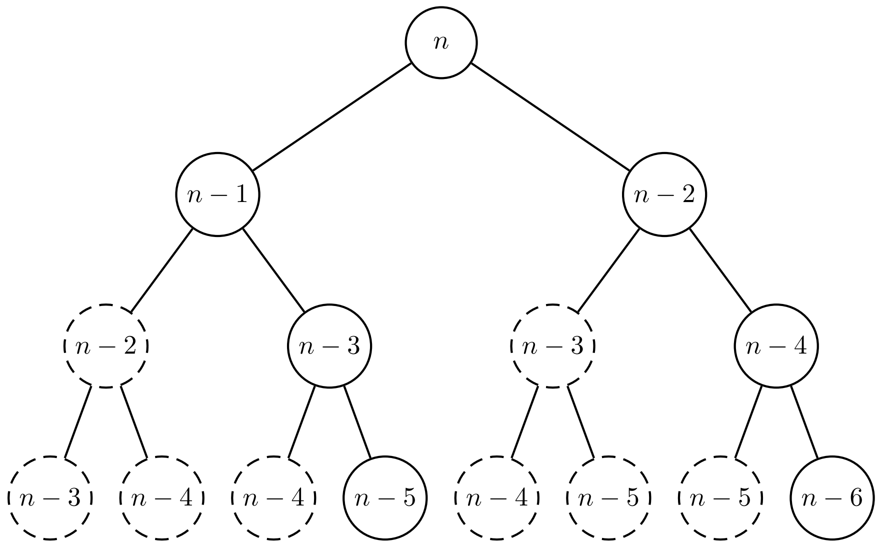


Figure 2.5 The recursion tree for RECURSIVEFIBONACCI(n). Vertices enclosed in dashed circles represent duplicated effort—the same value had been calculated in another vertex in the tree at a higher level. As the tree grows larger, the number of dashed vertices increases exponentially ($2^i - 2$ at level i), while the number of regular vertices increases linearly (2 per level).

scribe its running time, since this is an attribute of the algorithm, and not an attribute of the computer you happen to be using.

Unfortunately, determining how many operations an algorithm will perform is not always easy. We can see that USCHANGE will always perform 17 operations (one for each assignment, subtraction, multiplication, and division), but this is a very simple algorithm. An algorithm like SELECTIONSORT, on the other hand, will perform a different number of operations depending on what it receives as input: it will take less time to sort a 5-element list than it will to sort a 5000-element list. You might be tempted to think that SELECTIONSORT will take 1000 times longer to sort a 5000-element array than it will to sort a 5-element array. But you would be wrong. As we will see, it actually takes on the order of $1000^2 = 1,000,000$ times longer, no matter what kind of computer you use. It is typically the case that the larger the input is, the longer the algorithm takes to process it.

If we know how to compute the number of basic operations that an algorithm performs, then we have a basis to compare it against a different algorithm that solves the same problem. Rather than tediously count every multiplication and addition, we can perform this comparison by gaining a high-level understanding of the growth of each algorithm's operation count as the size of the input increases. To illustrate this, suppose an algorithm $\mathcal{A}$ performs $11n^3$ operations on an input of size n, and a different algorithm, $\mathcal{B}$, solves the same problem in $99n^2 + 7$ operations. Which algorithm, $\mathcal{A}$ or $\mathcal{B}$, is faster? Although, $\mathcal{A}$ may be faster than $\mathcal{B}$ for some small n (e.g., for n between 0 and 9), $\mathcal{B}$ will become faster with large n (e.g., for all $n \geq 10$). Since n^3 is, in some sense, a "faster-growing" function than n^2 with respect to n, the constants 11, 99, and 7 do not affect the competition between the two algorithms for large n (see figure 2.6). We refer to $\mathcal{A}$ as a cubic algorithm and to $\mathcal{B}$ as a quadratic algorithm, and say that $\mathcal{A}$ is less efficient than $\mathcal{B}$ because it performs more operations to solve the same problem when n is large. Thus, we will often be somewhat imprecise when we count operations in algorithms—the behavior of algorithms on small inputs does not matter.

Let us estimate how long BRUTEFORCECHANGE will take on an input instance of M cents, and denominations $(c_1, c_2, \ldots, c_d)$. To calculate the total number of operations in the **for** loop, we can take the approximate number of operations performed in each iteration and multiply this by the total number of iterations. Since there are roughly $\frac{M}{c_1} \cdot \frac{M}{c_2} \cdots \frac{M}{c_d}$ iterations, the **for** loop performs on the order of $d \cdot \frac{M^d}{c_1 \cdot c_2 \cdots c_d}$ operations, which dwarfs the other operations in the algorithm.

This type of algorithm is often referred to as an *exponential* algorithm in contrast to quadratic, cubic, or other *polynomial* algorithms. The expression for the running time of exponential algorithms includes a term like M^d, where d is a *parameter* of the problem (i.e., d may deliberately be made arbitrarily large by changing the input to the algorithm), while the running time of a polynomial algorithm is bounded by a term like M^k where k is a constant not related to the size of any parameters. For example, an algorithm with running time M^1 (linear), M^2 (quadratic), M^3 (cubic), or even M^{2005} is polynomial. Of course, an algorithm with running time M^{2005} is not very practical, perhaps less so than some exponential algorithms, and much effort in computer science goes into designing faster and faster polynomial algorithms. Since d may be large when the algorithm is called with a long list of denominations [e.g., $\mathbf{c} = (1, 2, 3, 4, 5, \ldots, 100)$], we see that BRUTEFORCECHANGE can take a very long time to execute.

We have seen that the running time of an algorithm is often related to the *size* of its input. However, the running time of an algorithm can also vary among inputs of the *same* size. For example, suppose SELECTIONSORT first

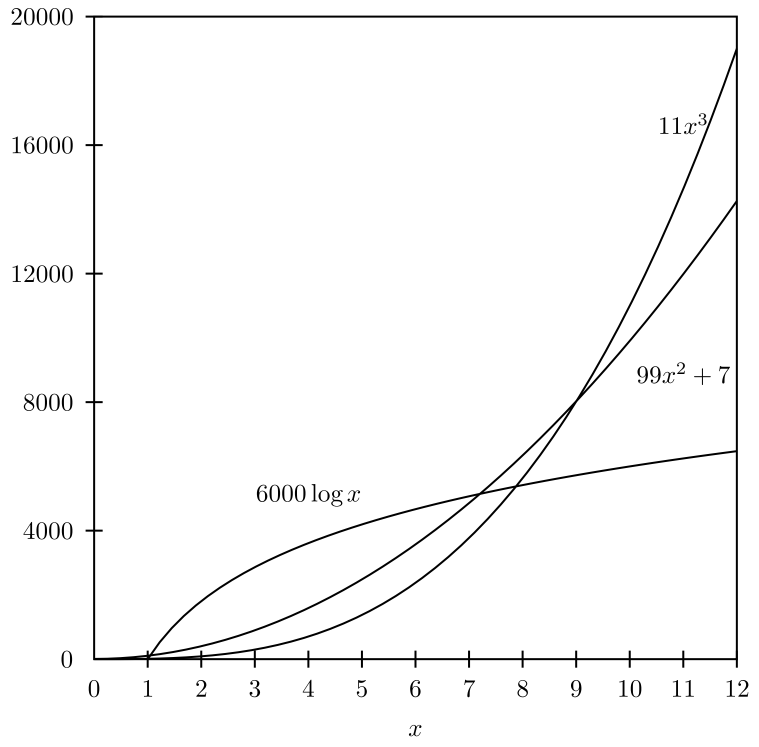


Figure 2.6 A comparison of a logarithmic ($h(x) = 6000 \log x$), a quadratic ($f(x) = 99x^2 + 7$), and a cubic ($g(x) = 11x^3$) function. After $x = 8$, both $f(x)$ and $g(x)$ are larger than $h(x)$. After $x = 9$, $g(x)$ is larger than $f(x)$, even though for values 0 through 9, $f(x)$ is larger than $g(x)$. The functions that we chose here are irrelevant and arbitrary: any three (positive-valued) functions with leading terms of $\log x$, x^2, and x^3 respectively would exhibit the same basic behavior, though the crossover points might be different.

checked to see if its input were already sorted. It would take this modified SELECTIONSORT less time to sort an ordered list of 5000 elements than it would to sort an unordered list of 5000 elements. As we see in the next section, when we speak of the running time of an algorithm as a function of input size, we refer to that one input—or set of inputs—of a particular size that the algorithm will take the longest to process. In the modified SELECTIONSORT, that input would be any not-already-sorted list.

2.8 Big-O Notation

Computer scientists use the *Big-O* notation to describe concisely the running time of an algorithm. If we say that the running time of an algorithm is quadratic, or $O(n^2)$, it means that the running time of the algorithm on an input of size n is limited by a quadratic function of n. That limit may be $99.7n^2$ or $0.001n^2$ or $5n^2+3.2n+99993$; the main factor that describes the growth rate of the running time is the term that grows the fastest with respect to n, for example n^2 when compared to terms like $3.2n$, or 99993. All functions with a leading term of n^2 have more or less the same rate of growth, so we lump them into one class which we call $O(n^2)$. The difference in behavior between two quadratic functions in that class, say $99.7n^2$ and $5n^2 + 3.2n + 99993$, is negligible when compared to the difference in behavior between two functions in different classes, say $5n^2 + 3.2n$ and $1.2n^3$. Of course, $99.7n^2$ and $5n^2$ are different functions and we would prefer an algorithm that takes $5n^2$ operations to an algorithm that takes $99.7n^2$. However, computer scientists typically ignore the leading constant and pay attention only to the fastest-growing term.

When we write $f(n) = O(n^2)$, we mean that the function $f(n)$ does not grow faster than a function with a leading term of cn^2, for a suitable choice of the constant c. A formal definition of Big-O notation, which is helpful in analyzing an algorithm's running time, is given in figure 2.7.

The relationship $f(n) = O(n^2)$ tells us that $f(n)$ does not grow faster than some quadratic function, but it does not tell us whether $f(n)$ grows slower than any quadratic function. In other words, $2n = O(n^2)$, but this valid statement is not as informative as it could be; $2n = O(n)$ is more precise. We say that the Big-O relationship establishes an *upper bound* on the growth of a function: if $f(n) = O(g(n))$, then the function f grows no faster than the function g. A similar concept exists for *lower bounds*, and we use the notation $f(n) = \Omega(g(n))$ to indicate that f grows no slower than g. If, for some function g, an algorithm's time grows no faster than g *and* no slower than g, then we say that g is a *tight bound* for the algorithm's running time. For example, if an algorithm requires $2n \log n$ time, then technically, it is an $O(n^2)$ algorithm even though this is a misleadingly loose bound. A tight bound on the algorithm's running time is actually $O(n \log n)$. Unfortunately, it is often easier to prove a loose bound than a tight one.

In keeping with the healthy dose of pessimism toward an algorithm's correctness, we measure an algorithm's efficiency as its *worst case* efficiency, which is the largest amount of time an algorithm can take given the worst

A function $f(x)$ is "Big-O of $g(x)$", or $O(g(x))$, when $f(x)$ is less than or equal to $g(x)$ to within some constant multiple c. If there are a few points x such that $f(x)$ is not less than $c \cdot g(x)$, this does not affect our overall understanding of f's growth. Mathematically speaking, the Big-O notation deals with *asymptotic* behavior of a function as its input grows arbitrarily large, beyond some (arbitrary) value x_0.

Definition 2.1 *A function $f(x)$ is $O(g(x))$ if there are positive real constants c and x_0 such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$.*

For example, the function $3x = O(.2x^2)$, but at $x = 1$, $3x > .2x^2$. However, for all $x > 15$, $.2x^2 > 3x$. Here, $x_0 = 15$ represents the point at which $3x$ is bounded above by $.2x^2$. Notice that this definition blurs the advantage gained by mere constants: $5x^2 = O(x^2)$, even though it would be wrong to say that $5x^2 \leq x^2$.
Like Big-O notation, which governs an upper bound on the growth of a function, we can define a relationship that reflects a lower bound on the growth of a function.

Definition 2.2 *A function $f(x)$ is $\Omega(g(x))$ if there are positive real constants c and x_0 such that $f(x) \geq cg(x)$ for all values of $x \geq x_0$.*

If $f(x) = \Omega(g(x))$, then f is said to grow "faster" than g.
Now, if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$ then we know very precisely how $f(x)$ grows with respect to $g(x)$. We call this the Θ relationship.

Definition 2.3 *A function $f(x)$ is $\Theta(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.*

Figure 2.7 Definitions of the Big-O, Ω, and Θ notations.

possible input of a given size. The advantage to considering the worst case efficiency of an algorithm is that we are guaranteed that our algorithm will never behave worse than our worst case estimate, so we are never surprised or disappointed. Thus, when we derive a Big-O bound, it is a bound on the worst case efficiency.

We illustrate the above notion of efficiency by analyzing the two sorting algorithms, SELECTIONSORT and RECURSIVESELECTIONSORT. The parameter that describes the input size is n, the number of integers in the input list, so we wish to determine the efficiency of the algorithms as a function of n.

The SELECTIONSORT algorithm makes $n-1$ iterations in the **for** loop and analyzes $n-i+1$ elements $a_i, \ldots, a_n$ in iteration i. In the first iteration, it analyzes all n elements, at the next one it analyzes $n-1$ elements, and so on. Therefore, the approximate number of operations performed in SELECTIONSORT is: $n+(n-1)+(n-2)+\cdots+2+1=1+2+\cdots+n = n(n+1)/2$.[15] At each iteration, the same swapping of array elements occurs, so SELECTIONSORT requires roughly $n(n+1)/2 + 3n$ operations, which is $O(n^2)$ operations.[16] Again, because we can safely ignore multiplicative constants and terms that are smaller than the fastest-growing term, our calculations are somewhat imprecise but yield an overall picture of the function's growth.

We will now consider RECURSIVESELECTIONSORT. Let $T(n)$ denote the amount of time that RECURSIVESELECTIONSORT takes on an n-element array. Calling RECURSIVESELECTIONSORT on an n-element array involves finding the smallest element (roughly n operations), followed by a recursive call on a list with $n-1$ elements, which performs $T(n-1)$ operations. Calling RECURSIVESELECTIONSORT on a 1-element list requires 1 operation (one for the **if** statement), so the following equations hold.

$$\begin{aligned} T(n) &= n + T(n-1) \\ T(1) &= 1 \end{aligned}$$

Therefore,

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1) \\ &= O(n^2). \end{aligned}$$

Thus, calling RECURSIVESELECTIONSORT on an n element array will require roughly the same $O(n^2)$ time as calling SELECTIONSORT. Since RECURSIVESELECTIONSORT always performs the same operations on a list of size n, we can be certain that this is a tight analysis of the running time of the algorithm. This is why using SELECTIONSORT to sort a 5000-element array takes $1,000,000$ times longer than it does to sort a 5-element array: $5,000^2 = 1,000,000 \cdot 5^2$.

15. Here we rely on the fact that $1 + 2 + \cdots + n = n(n+1)/2$.
16. Each swapping requires three (rather than two) operations.

Of course, this does not show that the Sorting problem requires $O(n^2)$ time to solve. All we have shown so far is that two particular algorithms, RECURSIVESELECTIONSORT and SELECTIONSORT, require $O(n^2)$ time; in fact, we will see a different sorting algorithm in chapter 7 that runs in $O(n \log n)$ time.

We can use the same technique to calculate the running time of HANOITOWERS called on a tower of size n. Let $T(n)$ denote the number of disk moves that HANOITOWERS(n) performs. The following equations hold.

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 1 \\ T(1) &= 1 \end{aligned}$$

This recurrence relation produces the following sequence: $1, 3, 7, 15, 31, 63$, and so on. We can solve it by adding 1 to both sides and noticing

$$T(n) + 1 = 2 \cdot T(n-1) + 1 + 1 = 2(T(n-1) + 1).$$

If we introduce a new variable, $U(n) = T(n) + 1$, then $U(n) = 2 \cdot U(n-1)$. Thus, we have changed the problem to the following recurrence relation.

$$\begin{aligned} U(n) &= 2 \cdot U(n-1) \\ U(1) &= 2 \end{aligned}$$

This gives rise to the sequence $2, 4, 8, 16, 32, 64, \ldots$ and it is easy to see that $U(n) = 2^n$. Since $T(n) = U(n) - 1$, we see that $T(n) = 2^n - 1$. Thus, HANOITOWERS is an exponential algorithm, which we hinted at in section 2.5.

2.9 Algorithm Design Techniques

Over the last forty years, computer scientists have discovered that many algorithms share similar ideas, even though they solve very different problems. There appear to be relatively few basic techniques that can be applied when designing an algorithm, and we cover some of them later in this book in varying degrees of detail. For now we will mention the most common algorithm design techniques, so that future examples can be categorized in terms of the algorithm's design methodology.

To illustrate the design techniques, we will consider a very simple problem that plagues nearly everyone with a cordless phone. Suppose your cordless phone rings, but you have misplaced the handset somewhere in your home.

How do you find it? To complicate matters, you have just walked into your home with an armful of groceries, and it is dark out, so you cannot rely solely on eyesight.

2.9.1 Exhaustive Search

An *exhaustive search*, or *brute force*, algorithm examines every possible alternative to find one particular solution.

For example, if you used the brute force algorithm to find the ringing telephone, you would ignore the ringing of the phone, as if you could not hear it, and simply walk over every square inch of your home checking to see if the phone was present. You probably would not be able to answer the phone before it stopped ringing, unless you were very lucky, but you would be guaranteed to eventually find the phone no matter where it was.

BRUTEFORCECHANGE is a brute force algorithm, and chapter 4 introduces some additional examples of such algorithms—these are the easiest algorithms to design and understand, and sometimes they work acceptably for certain practical problems in biology. In general, though, brute force algorithms are too slow to be practical for anything but the smallest instances

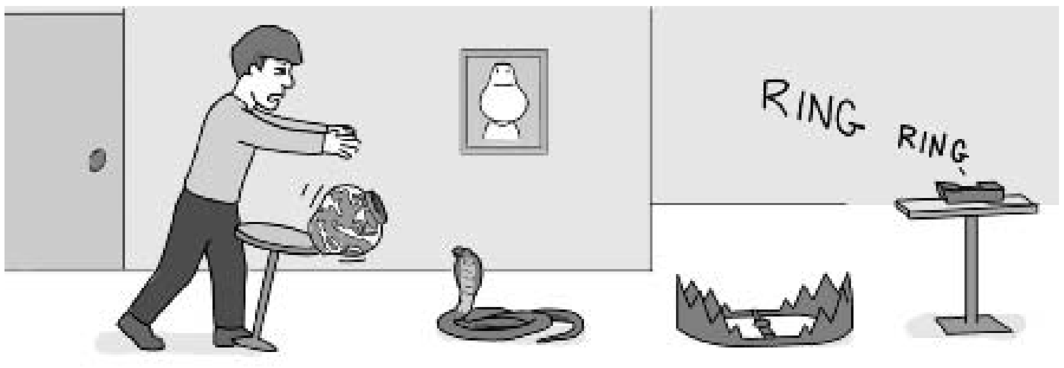
and we will spend most of this book demonstrating how to avoid the brute force algorithms or how to finesse them into faster versions.

2.9.2 Branch-and-Bound Algorithms

In certain cases, as we explore the various alternatives in a brute force algorithm, we discover that we can omit a large number of alternatives, a technique that is often called *branch-and-bound*, or *pruning*.

Suppose you were exhaustively searching the first floor and heard the phone ringing above your head. You could immediately rule out the need to search the basement or the first floor. What may have taken three hours may now only take one, depending on the amount of space that you can rule out.

2.9.3 Greedy Algorithms



Many algorithms are iterative procedures that choose among a number of alternatives at each iteration. For example, a cashier can view the Change problem as a series of decisions he or she has to make: which coin (among d denominations) to return first, which to return second, and so on. Some of these alternatives may lead to correct solutions while others may not. Greedy algorithms choose the "most attractive" alternative at each iteration, for example, the largest denomination possible. USCHANGE used quarters, then dimes, then nickels, and finally pennies (in that order) to make change for M. By greedily choosing the largest denomination first, the algorithm avoided any combination of coins that included fewer than three quarters to make change for an amount larger than or equal to 75 cents. Of course, we showed that the generalization of this greedy strategy, BETTERCHANGE, produced incorrect results when certain new denominations were included.

In the telephone example, the corresponding greedy algorithm would simply be to walk in the direction of the telephone's ringing until you found it. The problem here is that there may be a wall (or an expensive and fragile vase) between you and the phone, preventing you from finding it. Unfortunately, these sorts of difficulties frequently occur in most realistic problems. In many cases, a greedy approach will seem "obvious" and natural, but will be subtly wrong.

2.9.4 Dynamic Programming

Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct the solution of the larger one. During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the running time. Dynamic programming organizes computations to avoid re-computing values that you already know, which can often save a great deal of time. The Ringing Telephone problem does not lend itself to a dynamic programming solution, so we consider a different problem to illustrate the technique.

Suppose that instead of answering the phone you decide to play the "Rocks" game from the previous chapter with two piles of rocks, say ten in each. We remind the reader that in each turn, one player may take either one rock (from either pile) or two rocks (one from each pile). Once the rocks are taken, they are removed from play. The player that takes the last rock wins the game. You make the first move.

To find the winning strategy for the $10+10$ game, we can construct a table, which we can call **R**, shown below. Instead of solving a problem with 10 rocks in each pile, we will solve a more general problem with n rocks in one pile and m rocks in another (the $n+m$ game) where n and m are arbitrary. If Player 1 can always win the game of $5+6$, then we would say $R_{5,6} = W$, but if Player 1 has no winning strategy against a player that always makes the right moves, we would write $R_{5,6} = L$. Computing $R_{n,m}$ for an arbitrary n and m seems difficult, but we can build on smaller values. Some games, notably $R_{0,1}$, $R_{1,0}$, and $R_{1,1}$, are clearly winning propositions for Player 1 since in the first move, Player 1 can win. Thus, we fill in entries $(1,1)$, $(0,1)$ and $(1,0)$ as W.

	0	1	2	3	4	5	6	7	8	9	10
0		W									
1	W	W									
2											
3											
4											
5											
6											
7											
8											
9											
10											

After the entries $(0,1)$, $(1,0)$, and $(1,1)$ are filled, one can try to fill other entries. For example, in the $(2,0)$ case, the only move that Player 1 can make leads to the $(1,0)$ case that, as we already know, is a winning position for his opponent. A similar analysis applies to the $(0,2)$ case, leading to the following result:

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W									
2	L										
3											
4											
5											
6											
7											
8											
9											
10											

In the $(2,1)$ case, Player 1 can make three different moves that lead respectively to the games of $(1,1)$, $(2,0)$, or $(1,0)$. One of these cases, $(2,0)$, leads to a losing position for his opponent and therefore $(2,1)$ is a winning position. The case $(1,2)$ is symmetric to $(2,1)$, so we have the following table:

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W	W								
2	L	W									
3											
4											
5											
6											
7											
8											
9											
10											

Now we can fill in $R_{2,2}$. In the $(2,2)$ case, Player 1 can make three different moves that lead to entries $(2,1)$, $(1,2)$, and $(1,1)$. All of these entries are winning positions for his opponent and therefore $R_{2,2} = L$

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W	W								
2	L	W	L								
3											
4											
5											
6											
7											
8											
9											
10											

We can proceed filling in $\mathbf{R}$ in this way by noticing that for the entry (i, j) to be L, the entries above, diagonally to the left and directly to the left, must be W. These entries ($(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$) correspond to the three possible moves that player 1 can make.

	0	1	2	3	4	5	6	7	8	9	10
0		W	L	W	L	W	L	W	L	W	L
1	W	W	W	W	W	W	W	W	W	W	W
2	L	W	L	W	L	W	L	W	L	W	L
3	W	W	W	W	W	W	W	W	W	W	W
4	L	W	L	W	L	W	L	W	L	W	L
5	W	W	W	W	W	W	W	W	W	W	W
6	L	W	L	W	L	W	L	W	L	W	L
7	W	W	W	W	W	W	W	W	W	W	W
8	L	W	L	W	L	W	L	W	L	W	L
9	W	W	W	W	W	W	W	W	W	W	W
10	L	W	L	W	L	W	L	W	L	W	L

The ROCKS algorithm determines if Player 1 wins or loses. If Player 1 wins in an n+m game, ROCKS returns W. If Player 1 loses, ROCKS returns L. The ROCKS algorithm introduces an artificial initial condition, $R_{0,0} = L$ to simplify the pseudocode.

```
ROCKS($n, m$)
1  $R_{0,0} = L$
2  for $i \leftarrow 1$ to $n$
3      if $R_{i-1,0} = W$
4          $R_{i,0} \leftarrow L$
5      else
6          $R_{i,0} \leftarrow W$
7  for $j \leftarrow 1$ to $m$
8      if $R_{0,j-1} = W$
9          $R_{0,j} \leftarrow L$
10     else
11         $R_{0,j} \leftarrow W$
12 for $i \leftarrow 1$ to $n$
13     for $j \leftarrow 1$ to $m$
14         if $R_{i-1,j-1} = W$ and $R_{i,j-1} = W$ and $R_{i-1,j} = W$
15             $R_{i,j} \leftarrow L$
16         else
17             $R_{i,j} \leftarrow W$
18 return $R_{n,m}$
```

In point of fact, a faster algorithm to solve the Rocks puzzle relies on the simply pattern in **R**, and checks to see if n and m are both even, in which case the player loses.

```
FASTROCKS($n, m$)
1  if $n$ and $m$ are both even
2      return $L$
3  else
4      return $W$
```

However, though FASTROCKS is more efficient than ROCKS, it may be difficult to modify it for other games, for example a game in which each player can move up to three rocks at a time from the piles. This is one example where the slower algorithm is more instructive than a faster one. But obviously, it is usually better to use the faster one when you really need to solve the problem.

2.9.5 Divide-and-Conquer Algorithms

One big problem may be hard to solve, but two problems that are half the size may be significantly easier. In these cases, *divide-and-conquer* algorithms fare well by doing just that: splitting the problem into smaller subproblems, solving the subproblems independently, and combining the solutions of subproblems into a solution of the original problem. The situation is usually more complicated than this and after splitting one problem into subproblems, a divide-and-conquer algorithm usually splits these subproblems into even smaller sub-subproblems, and so on, until it reaches a point at which it no longer needs to recurse. A critical step in many divide-and-conquer algorithms is the recombining of solutions to subproblems into a solution for a larger problem. Often, this merging step can consume a considerable amount of time. We will see examples of this technique in chapter 7.

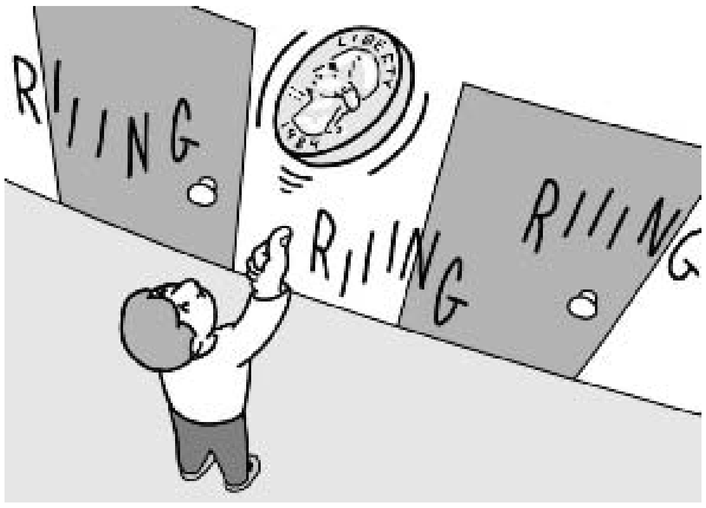
2.9.6 Machine Learning

Another approach to the phone search problem is to collect statistics over the course of a year about where you leave the phone, learning where the phone tends to end up most of the time. If the phone was left in the bathroom 80% of the time, in the bedroom 15% of the time, and in the kitchen 5% of the time, then a sensible time-saving strategy would be to start the search in the bathroom, continue to the bedroom, and finish in the kitchen. Machine learning algorithms often base their strategies on the computational analysis of previously collected data.

2.9.7 Randomized Algorithms

If you happen to have a coin, then before even starting to search for the phone, you could toss it to decide whether you want to start your search on the first floor if the coin comes up heads, or on the second floor if the coin comes up tails. If you also happen to have a die, then after deciding on the second floor, you could roll it to decide in which of the six rooms on the second floor to start your search.[17] Although tossing coins and rolling dice may be a fun way to search for the phone, it is certainly not the intuitive thing to do, nor is it at all clear whether it gives you any algorithmic advantage over a deterministic algorithm. We will learn how randomized algorithms

17. Assuming that you have a large house, of course.



help solve practical problems, and why some of them have a competitive advantage over deterministic algorithms.

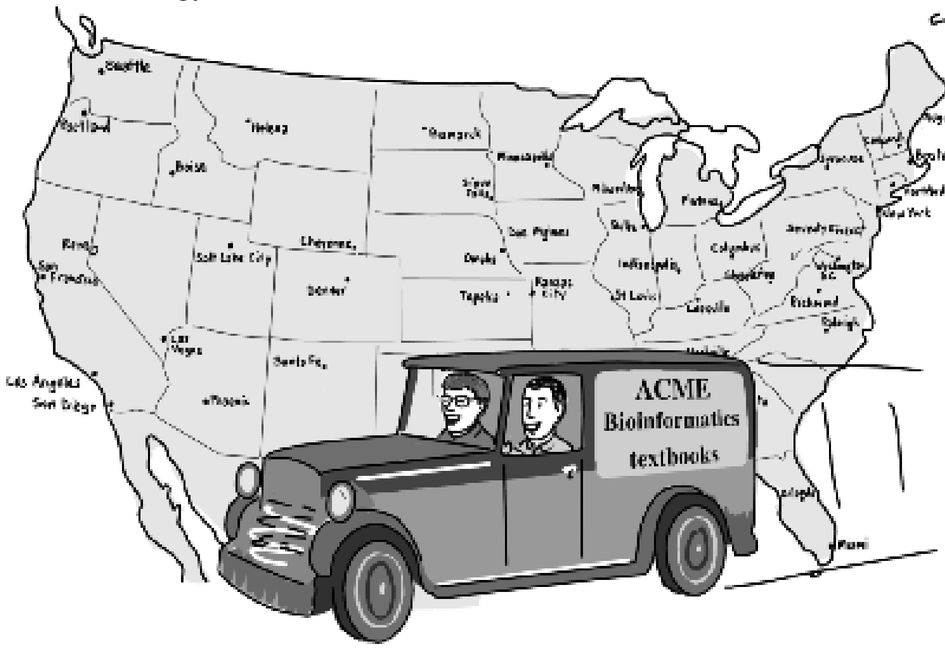
2.10 Tractable versus Intractable Problems

We have described a correct algorithm that solves the Change problem, but requires exponential time to do so. This does not mean that all algorithms that solve the Change problem will require exponential time. Showing that a particular algorithm requires exponential time is much different than showing that a *problem* cannot be solved by *any* algorithm in less than exponential time. For example, we showed that RECURSIVEFIBONACCI required exponential time to compute the nth Fibonacci number, while FIBONACCI solves the same problem in linear $O(n)$ time.[18]

We have seen that algorithms can be categorized according to their complexity. In the early 1970s, computer scientists discovered that *problems* could also be categorized according to their inherent complexity. It turns out that some problems, such as listing every subset of an n-element set, require exponential time—no algorithm, no matter how clever, could possibly solve the problem in less than exponential time. Other problems, such as sorting a list of integers, require only polynomial time. Somewhere between the polynomial problems and the exponential problems lies one particularly important category of problems called the $\mathcal{NP}$*-complete* problems. These are problems

18. There exists an algorithm even faster than $O(n)$ to compute the n-th Fibonacci number; it does not calculate all of the Fibonacci numbers in the sequence up to n.

that appear to be quite difficult, in that no polynomial-time algorithm for any of these problems has yet been found. However, nobody can seem to prove that polynomial-time algorithms for these problems are impossible, so nobody can rule out the possibility that these problems are actually efficiently solvable. One particularly famous example of an $\mathcal{NP}$-complete problem is the Traveling Salesman problem, which has a wide variety of practical applications in biology.



Traveling Salesman Problem:
Find the shortest path through a set of cities, visiting each city only one time.

> **Input:** A map of cities, roads between the cities, and distances along the roads.
>
> **Output:** A sequence of roads that will take a salesman through every city on the map, such that he will visit each city exactly once, and will travel the shortest total distance.

The critical property of $\mathcal{NP}$-complete problems is that, if one $\mathcal{NP}$-complete problem is solvable by a polynomial-time algorithm, then *all* $\mathcal{NP}$-complete problems can be solved by minor modifications of the same algorithm. The fact that nobody has yet found that magic algorithm, after half a century of research, suggests that it may not exist. However, this has not yet been proven mathematically. It turns out that thousands of algorithmic problems are actually instances of the Traveling Salesman problems in disguise.[19] Taken together, these problems form the class of $\mathcal{NP}$-complete problems.

Despite the lack of mathematical proof, attempting to find a polynomial algorithm for an $\mathcal{NP}$-complete problem will likely result in failure and a whole lot of wasted time. Unfortunately, proving that a problem really is $\mathcal{NP}$-complete, and not just superficially difficult, is somewhat of an undertaking. In fact, it is not yet known whether or not some important bioinformatics problems are $\mathcal{NP}$-complete.

2.11 Notes

The word "algorithm" derived from the name of the ninth-century Arab mathematician, al-Khwarizmi, of the court of Caliph Mamun in Baghdad. al-Khwarizmi was a scholar in the House of Wisdom, where Greek scientific works were translated; much of the mathematical knowledge in medieval Europe was derived from Latin translations of his books. More recent books on the general topic of algorithms are Knuth, 1998 (57); Aho, Hopcroft, and Ullman, 1983 (1); and Cormen et al., 2001 (24).

The notion of $\mathcal{NP}$-completeness was proposed in the early 1970s by Stephen Cook (23), and Leonid Levin (65), and was further analyzed by Richard Karp in 1972 (53) who demonstrated a rich variety of $\mathcal{NP}$-complete problems. Garey and Johnson (39) authored an encyclopedic reference of $\mathcal{NP}$-complete problems.

19. Although these problems may have nothing in common with the Traveling Salesman problem—no cities, no roads, no distances—the Traveling Salesman problem can be converted into each one, and vice versa.

Richard Karp, born 1935 in Boston, is a Professor at the University of California at Berkeley, with a principal appointment in computer science and additional appointments in mathematics, bioengineering, and operations research. He attended Boston Latin School and Harvard University, where he received a PhD in Applied Mathematics in 1959. From 1959 to 1968 he was a member of the Mathematical Sciences Department of the IBM Research Center in Yorktown Heights, NY. He has been a faculty member at the University of California at Berkeley since 1968 (with the exception of the period 1995–99, when he was a professor at the University of Washington). Since 1988 he has also been a research scientist at the International Computer Science Institute, a non-profit research company in Berkeley. Karp says:

> Ever since my undergraduate days I have had a fascination with combinatorial algorithms. These puzzle-like problems involve searching through a finite but vast set of possibilities for a pattern or structure that meets certain requirements or is of minimum cost. Examples in bioinformatics include sequence assembly, multiple alignment of sequences, phylogeny construction, the analysis of genomic rearrangements, and the modeling of gene regulation. For some combinatorial problems, there are elegant and efficient algorithms that proceed like clockwork to find the required solution, but most are less tractable and require either a very long computation or a compromise on a solution that may not be optimal.

In 1972, Karp developed an approach to showing that many seemingly hard combinatorial problems are equivalent in the sense that either all of them or none of them are efficiently solvable (a problem is considered efficiently solvable if it can be solved by a polynomial algorithm). These problems are the "$\mathcal{NP}$-Complete" problems. Over the years, thousands of examples have been added to his original list of twenty-one $\mathcal{NP}$-complete problems, yet despite intensive effort none of these problems has been shown to be efficiently solvable. Many computer scientists (including Karp) believe that none of them ever will be.

Karp began working in bioinformatics circa 1991, attracted by the belief that computational methods might reveal the secret inner workings of living organisms. He says:

> [I hoped] that my experience in studying combinatorial algorithms could be useful in cracking those secrets. I have indeed been able to apply my skills in this new area, but only after coming to understand that solving biological problems requires far more than clever algorithms: it involves a creative partnership between biologists and mathematical scientists to arrive at an appropriate mathematical model, the acquisition and use of diverse sources of data, and statistical methods to show that the biological patterns and regularities that we discover could not be due to chance. My recent work is concerned with analyzing the transcriptional regulation of genes, discovering conserved regulatory pathways, and analyzing genetic variation in humans.
>
> There have been spectacular advances in biology since 1991, most notable being the sequencing of genomes. I believe that we are now poised to understand—and possibly even reprogram— the gene regulatory networks and the metabolic networks that control cellular processes. By comparing many related organisms, we hope to understand how these networks evolved. Effectively, we are trying to find the genetic basis of complex diseases so that we can develop more effective modes of treatment.

2.12 Problems

Problem 2.1

Write an algorithm that, given a list of n numbers, returns the largest and smallest numbers in the list. Estimate the running time of the algorithm. Can you design an algorithm that performs only $3n/2$ comparisons to find the smallest and largest numbers in the list?

Problem 2.2

Write two algorithms that iterate over every index from $(0, 0, \ldots, 0)$ to $(n_1, n_2, \ldots, n_d)$. Make one algorithm recursive, and the other iterative.

Problem 2.3

Is $\log n = O(n)$? Is $\log n = \Omega(n)$? Is $\log n = \Theta(n)$?

Problem 2.4

You are given an unsorted list of $n - 1$ distinct integers from the range 1 to n. Write a linear-time algorithm to find the missing integer.

Problem 2.5

Though FIBONACCI(n) is fast, it uses a fair amount of space to store the array $\mathbf{F}$. How much storage will it require to calculate the nth Fibonacci number? Modify the algorithm to require a constant amount of storage, regardless of the value of n.

Problem 2.6

Prove that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \overline{\phi}^n)$$

where F_n is the nth Fibonacci number, $\phi = \frac{1+\sqrt{5}}{2}$ and $\overline{\phi} = \frac{1-\sqrt{5}}{2}$.

Problem 2.7

Design an algorithm for computing the n-th Fibonacci number that requires less than $O(n)$ time. *Hint:* You probably want to use the result from problem 2.6. However, computing ϕ^n naively still requires $O(n)$ time because each multiplication is a single operation.

Problem 2.8

Propose a more realistic model of rabbit life (and death) that limits the life span of rabbits by k years. For example, if $k = 2.5$, then the corresponding sequence $1, 1, 2, 3, 4$ grows more slowly than the Fibonacci seqence. Write a recurrence relation and pseudocode to compute the number of rabbits under this model. Will the number of rabbits ever exceed the number of atoms in the universe (under these assumptions)?

Problem 2.9

Write an iterative (i.e., nonrecursive) algorithm to solve the Hanoi Tower problem.

Problem 2.10

Prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Problem 2.11

Prove that $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$ and that $\sum_{i=1}^{n} 2^{-i} = 1 - 2^{-n}$.

Problem 2.12

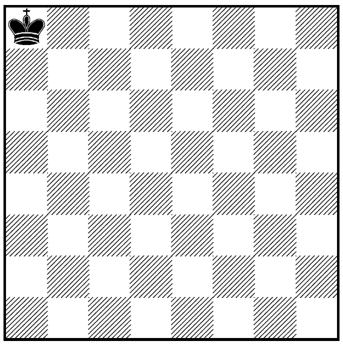
We saw that BETTERCHANGE is an incorrect algorithm for the set of denominations $(25, 20, 10, 5, 1)$. Add a new denomination to this set such that BETTERCHANGE will return the correct change combination for any value of M.

Problem 2.13

Design an algorithm that computes the average number of coins returned by the program USCHANGE(M) as M varies from 1 to 100.

Problem 2.14

Given a set of arbitrary denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, write an algorithm that can decide whether BETTERCHANGE is a correct algorithm when run on $\mathbf{c}$.



Problem 2.15

A king stands on the upper left square of the chessboard. Two players make turns moving the king either one square to the right or one square downward or one square along a diagonal in the southeast direction. The player who can place the king on the lower right square of the chessboard wins. Who will win? Describe the winning strategy.

Problem 2.16

Bob and Alice are bored one Saturday afternoon so they invent the following game. Initially, there are n rocks in a single pile. At each turn, one of the two players can split any pile of rocks that has more than 1 rock into two piles of arbitrary size such that the size of each of the two new piles must add up to the size of the original big pile. No player can split a pile that has only a single rock, and the last person to move wins. Does one of the two players, first or second, have an advantage? Explain which player will win for each value of n.

Problem 2.17

There are n bacteria and 1 virus in a Petri dish. Within the first minute, the virus kills one bacterium and produces another copy of itself, and all of the remaining bacteria reproduce, making 2 viruses and $2 \cdot (n-1)$ bacteria. In the second minute, each of the viruses kills a bacterium and produces a new copy of itself (resulting in 4 viruses and $2(2(n-1)-2) = 4n-8$ bacteria; again, the remaining bacteria reproduce. This process continues every minute. Will the viruses eventually kill all the bacteria? If so, design an algorithm that computes how many steps it will take. How does the running time of your algorithm depend on n?

Problem 2.18

A very large bioinformatics department at a prominent university has a mix of 100 professors: some are honest and hard-working, while others are deceitful and do not like students. The honest professors always tell the truth, but the deceitful ones sometimes tell the truth and sometimes lie. You can ask any professors the following question about any other professor: "Professor Y, is Professor X honest?" Professor Y will answer with either "yes" or "no." Design an algorithm that, with no more than 198 questions, would allow you to figure out which of the 100 professors are honest (thus identifying possible research advisors). It is known that there are more honest than dishonest professors.

Problem 2.19

You are given an 8×8 table of natural numbers. In any one step, you can either double each of the numbers in any one row, or subtract 1 from each of the numbers in any one column. Devise an algorithm that transforms the original table into a table of all zeros. What is the running time of your algorithm?

Problem 2.20

There are n black, m green, and k brown chameleons on a deserted island. When two chameleons of different colors meet they both change their color to the third one (e.g., black and green chameleons become brown). For each choice of $n, m,$ and k decide whether it is possible that after some time all the chameleons on the island are the same color (if you think that it is always possible, check the case $n = 1$, $m = 3$, and $k = 5$).

6 *Dynamic Programming Algorithms*

We introduced dynamic programming in chapter 2 with the Rocks problem. While the Rocks problem does not appear to be related to bioinformatics, the algorithm that we described is a computational twin of a popular alignment algorithm for sequence comparison. Dynamic programming provides a framework for understanding DNA sequence comparison algorithms, many of which have been used by biologists to make important inferences about gene function and evolutionary history. We will also apply dynamic programming to gene finding and other bioinformatics problems.

6.1 The Power of DNA Sequence Comparison

After a new gene is found, biologists usually have no idea about its function. A common approach to inferring a newly sequenced gene's function is to find similarities with genes of known function. A striking example of such a biological discovery made through a similarity search happened in 1984 when scientists used a simple computational technique to compare the newly discovered cancer-causing ν-sis oncogene with all (at the time) known genes. To their astonishment, the cancer-causing gene matched a normal gene involved in growth and development called platelet-derived growth factor (PDGF).[1] After discovering this similarity, scientists became suspicious that cancer might be caused by a normal growth gene being switched on at the wrong time—in essence, a good gene doing the right thing at the wrong time.

1. *Oncogenes* are genes in viruses that cause a cancer-like transformation of infected cells. Oncogene ν-sis in the *simian sarcoma virus* causes uncontrolled cell growth and leads to cancer in monkeys. The seemingly unrelated *growth factor* PDGF is a protein that stimulates cell growth.

Another example of a successful similarity search was the discovery of the cystic fibrosis gene. Cystic fibrosis is a fatal disease associated with abnormal secretions, and is diagnosed in children at a rate of 1 in 3900. A defective gene causes the body to produce abnormally thick mucus that clogs the lungs and leads to lifethreatening lung infections. More than 10 million Americans are unknowing and symptomless carriers of the defective cystic fibrosis gene; each time two carriers have a child, there is a 25% chance that the child will have cystic fibrosis.

In 1989 the search for the cystic fibrosis gene was narrowed to a region of 1 million nucleotides on the chromosome 7, but the exact location of the gene remained unknown. When the area around the cystic fibrosis gene was sequenced, biologists compared the region against a database of all known genes, and discovered similarities between some segment within this region and a gene that had already been discovered, and was known to code for *adenosine triphosphate (ATP) binding proteins*.[2] These proteins span the cell membrane multiple times as part of the ion transport channel; this seemed a plausible function for a cystic fibrosis gene, given the fact that the disease involves sweat secretions with abnormally high sodium content. As a result, the similarity analysis shed light on a damaged mechanism in faulty cystic fibrosis genes.

Establishing a link between cancer-causing genes and normal growth genes and elucidating the nature of cystic fibrosis were only the first success stories in sequence comparison. Many applications of sequence comparison algorithms quickly followed, and today bioinformatics approaches are among the dominant techniques for the discovery of gene function.

This chapter describes algorithms that allow biologists to reveal the similarity between different DNA sequences. However, we will first show how dynamic programming can yield a faster algorithm to solve the Change problem.

6.2 The Change Problem Revisited

We introduced the Change problem in chapter 2 as the problem of changing an amount of money M into the smallest number of coins from denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$. We showed that the naive greedy solution used by cashiers everywhere is not actually a correct solution to this problem, and ended with a correct—though slow—brute force algorithm. We will con-

2. ATP binding proteins provide energy for many reactions in the cell.

sider a slightly modified version of the Change problem, in which we do not concern ourselves with the actual combination of coins that make up the optimal change solution. Instead, we only calculate the smallest number of coins needed (it is easy to modify this algorithm to also return the coin combination that achieves that number).

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents. The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;

- the best combination for $77 - 3 = 74$ cents, plus a 3-cent coin;

- the best combination for $77 - 7 = 70$ cents, plus a 7-cent coin.

For 77 cents, the best combination would be the smallest of the above three choices. The same logic applies to 76 cents (best of 75, 73, or 69 cents), and so on (fig. 6.1). If $bestNumCoins_M$ is the smallest number of coins needed to change M cents, then the following recurrence relation holds:

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-1} + 1 \\ bestNumCoins_{M-3} + 1 \\ bestNumCoins_{M-7} + 1 \end{cases}$$

In the more general case of d denominations $\mathbf{c} = (c_1, \ldots, c_d)$:

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-c_1} + 1 \\ bestNumCoins_{M-c_2} + 1 \\ \vdots \\ bestNumCoins_{M-c_d} + 1 \end{cases}$$

This recurrence motivates the following algorithm:

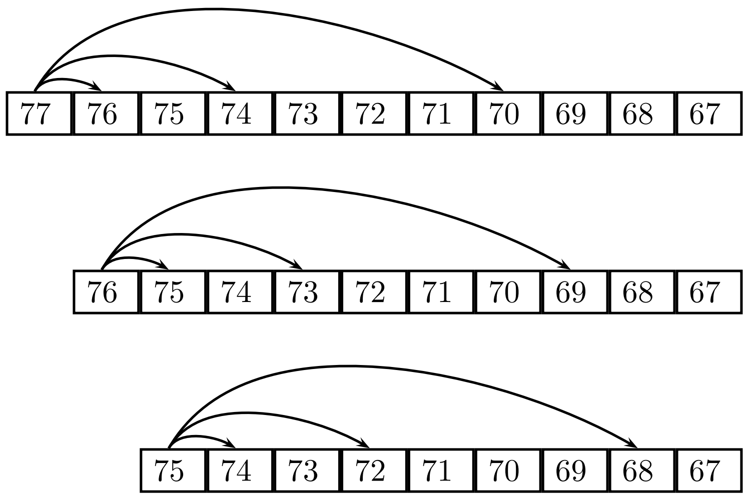


Figure 6.1 The relationships between optimal solutions in the Change problem. The smallest number of coins for 77 cents depends on the smallest number of coins for 76, 74, and 70 cents; the smallest number of coins for 76 cents depends on the smallest number of coins for 75, 73, and 69 cents, and so on.

```
RECURSIVECHANGE($M, \mathbf{c}, d$)
1 if $M = 0$
2     return 0
3 $bestNumCoins \leftarrow \infty$
4 for $i \leftarrow 1$ to $d$
5     if $M \geq c_i$
6         $numCoins \leftarrow$ RECURSIVECHANGE($M - c_i, \mathbf{c}, d$)
7         if $numCoins + 1 < bestNumCoins$
8             $bestNumCoins \leftarrow numCoins + 1$
9 return $bestNumCoins$
```

The sequence of calls that RECURSIVECHANGE makes has a feature in common with the sequence of calls made by RECURSIVEFIBONACCI, namely, that RECURSIVECHANGE recalculates the optimal coin combination for a given amount of money repeatedly. For example, the optimal coin combination for 70 cents is recomputed repeatedly nine times over and over as $(77-7)$, $(77-3-3-1)$, $(77-3-1-3)$, $(77-1-3-3)$, $(77-3-1-1-1-1)$, $(77-1-3-1-1-1)$, $(77-1-1-3-1-1)$, $(77-1-1-1-3-1)$, $(77-1-1-1-1-3)$, and $(77-1-1-1-1-1-1-1)$. The optimal coin combination for 20 cents will be recomputed billions of times rendering RECURSIVECHANGE impractical.

To improve RECURSIVECHANGE, we can use the same strategy as we did for the Fibonacci problem—all we really need to do is use the fact that the solution for M relies on solutions for $M - c_1$, $M - c_2$, and so on, and then reverse the order in which we solve the problem. This allows us to leverage previously computed solutions to form solutions to larger problems and avoid all this recomputation.

Instead of trying to find the minimum number of coins to change M cents, we attempt the superficially harder task of doing this for *each* amount of money, m, from 0 to M. This appears to require more work, but in fact, it simplifies matters. The following algorithm with running time $O(Md)$ calculates $bestNumCoins_m$ for increasing values of m. This works because the best number of coins for some value m depends only on values less than m.

```
DPCHANGE($M, \mathbf{c}, d$)
1 $bestNumCoins_0 \leftarrow 0$
2 **for** $m \leftarrow 1$ **to** $M$
3     $bestNumCoins_m \leftarrow \infty$
4     **for** $i \leftarrow 1$ **to** $d$
5         **if** $m \geq c_i$
6             **if** $bestNumCoins_{m-c_i} + 1 < bestNumCoins_m$
7                 $bestNumCoins_m \leftarrow bestNumCoins_{m-c_i} + 1$
8 **return** $bestNumCoins_M$
```

The key difference between RECURSIVECHANGE and DPCHANGE is that the first makes d recursive calls to compute the best change for M (and each of these calls requires a lot of work!), while the second analyzes the d already precomputed values to almost instantly compute the new one. As surprising as it may sound, simply reversing the order of computations in figure 6.1 makes a dramatic difference in efficiency (fig. 6.2).

We stress again the difference between the complexity of a problem and the complexity of an algorithm. In particular, we initially showed an $O(M^d)$ algorithm to solve the Change problem, and there did not appear to be any easy way to remedy this situation. Yet the DPCHANGE algorithm provides a simple $O(Md)$ solution. Conversely, a minor modification of the Change problem renders the problem very difficult. Suppose you had a limited number of each denomination and needed to change M cents using no more than the provided supply of each coin. Since you have fewer possible choices in

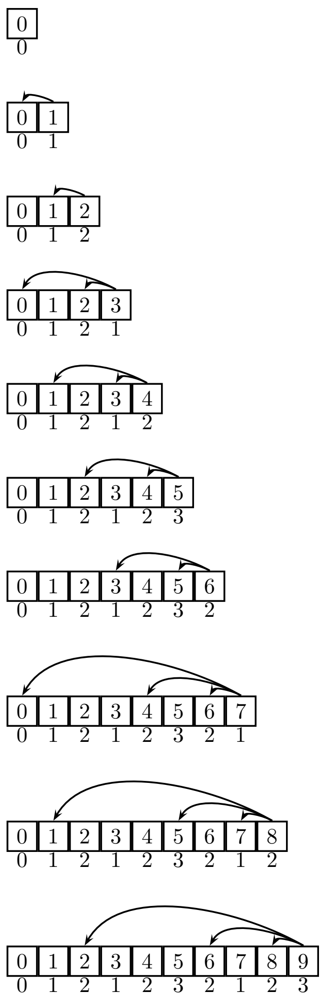


Figure 6.2 The solution for 9 cents ($bestNumCoins_9$) depends on 8 cents, 6 cents and 2 cent, but the smallest number of coins can be obtained by computing $bestNumCoins_m$ for $0 \leq m \leq 9$.

this new problem, it would seem to require even less time than the original Change problem, and that a minor modification to DPCHANGE would work. However, this is not the case and this problem turns out to be very difficult.

6.3 The Manhattan Tourist Problem

We will further illustrate dynamic programming with a surprisingly useful toy problem, called the Manhattan Tourist problem, and then build on this intuition to describe DNA sequence alignment.

Imagine a sightseeing tour in the borough of Manhattan in New York City, where a group of tourists are determined to walk from the corner of 59th Street and 8th Avenue to the Chrysler Building at 42nd Street and Lexington Avenue. There are many attractions along the way, but assume for the moment that the tourists want to see as many attractions as possible. The tourists are allowed to move either to the south or to the east, but even so, they can choose from many different paths (exactly how many is left as a problem at the end of the chapter). The upper path in figure 6.3 will take the tourists to the Museum of Modern Art, but they will have to miss Times Square; the bottom path will allow the tourists to see Times Square, but they will have to miss the Museum of Modern Art.

The map above can also be represented as a gridlike structure (figure 6.4) with the numbers next to each line (called *weights*) showing the number of attractions on every block. The tourists must decide among the many possible paths between the northwesternmost point (called the *source vertex*) and the southeasternmost point (called the *sink vertex*). The weight of a path from the source to the sink is simply the sum of weights of its edges, or the overall number of attractions. We will refer to this kind of construct as a *graph*, the intersections of streets we will call *vertices*, and the streets themselves will be *edges* and have a weight associated with them. We assume that horizontal edges in the graph are oriented to the east like $\rightarrow$ while vertical edges are oriented to the south like $\downarrow$. A *path* is a continuous sequence of edges, and the *length of a path* is the sum of the edge weights in the path.[3] A more detailed discussion of graphs can be found in chapter 8.

Although the upper path in figure 6.3 is better than the bottom one, in the sense that the tourists will see more attractions, it is not immediately clear if there is an even better path in the grid. The Manhattan Tourist problem is to find the path with the maximum number of attractions,[4] that is, a *longest path*

3. We emphasize that the length of paths in the graph represent the overall number of attractions on this path and has nothing to do with the real length of the path (in miles), that is, the distance the tourists travel.

4. There are many interesting museums and architectural landmarks in Manhattan. However, it is impossible to please everyone, so one can change the relative importance of the types of attractions by modulating the weights on the edges in the graph. This flexibility in assigning weights will become important when we discuss *scoring matrices* for sequence comparison.

(a path of maximum overall weight) in the grid.

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

Input: A weighted grid G with two distinguished vertices: a *source* and a *sink*.

Output: A longest path in G from *source* to *sink*.

Note that, since the tourists only move south and east, any grid positions west or north of the source are unusable. Similarly, any grid positions south or east of the sink are unusable, so we can simply say that the source vertex is at $(0, 0)$ and that the sink vertex at (n, m) defines the southeasternmost corner of the grid. In figure 6.4 $n = m = 4$, but n does not always have to equal m. We will use the grid shown in figure 6.4, rather than the one corresponding to the map of Manhattan in figure 6.3 so that you can see a nontrivial example of this problem.

The brute force approach to the Manhattan Tourist problem is to search among all paths in the grid for the longest path, but this is not an option for even a moderately large grid. Inspired by the previous chapter you may be tempted to use a greedy strategy. For example, a sensible greedy strategy would be to choose between two possible directions (south or east) by comparing how many attractions tourists would see if they moved one block south instead of moving one block east. This greedy strategy may provide rewarding sightseeing experience in the beginning but, a few blocks later, may bring you to an area of Manhattan you really do not want to be in. In fact, no known greedy strategy for the Manhattan Tourist problem provides an optimal solution to the problem. Had we followed the (obvious) greedy algorithm, we would have chosen the following path, corresponding to twenty three attractions.[5]

5. We will show that the optimal number is, in fact, thirty-four.

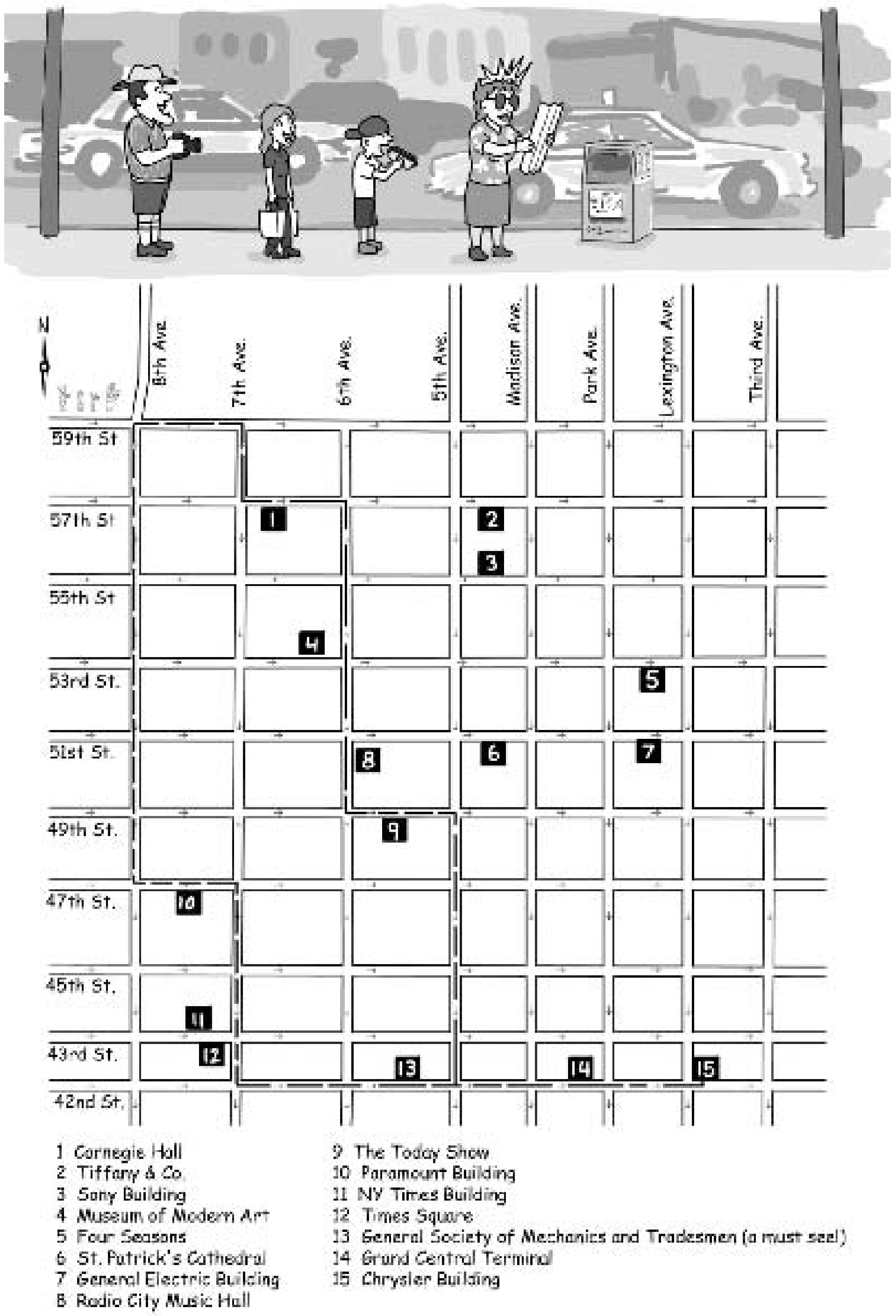


Figure 6.3 A city somewhat like Manhattan, laid out on a grid with one-way streets. You may travel only to the east or to the south, and you are currently at the northwesternmost point (source) and need to travel to the southeasternmost point (sink). Your goal is to visit as many attractions as possible.

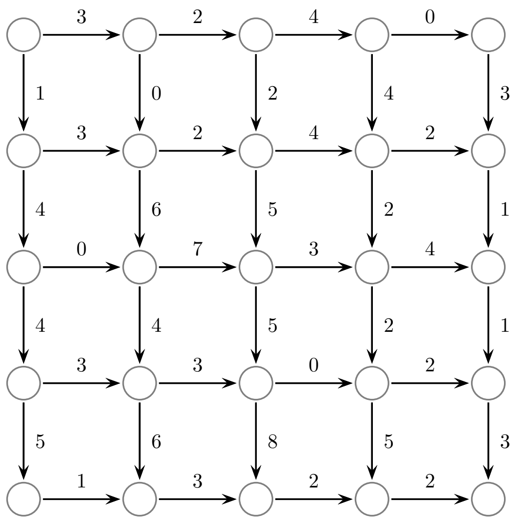
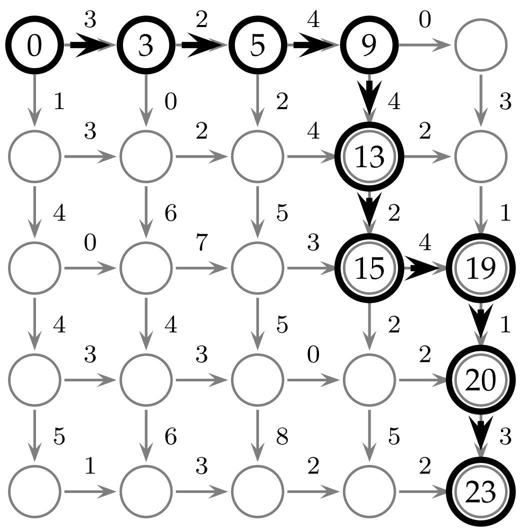


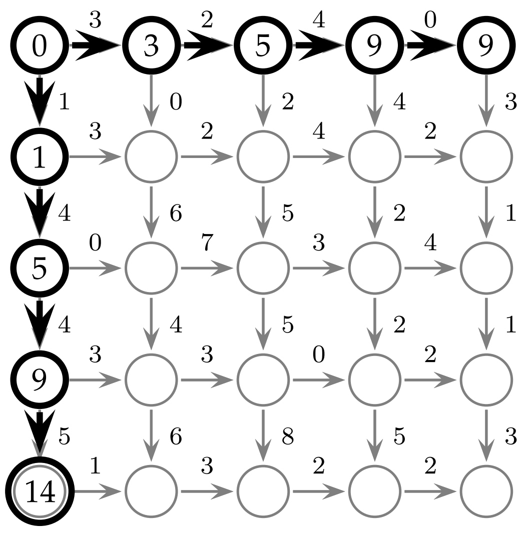
Figure 6.4 Manhattan represented as a graph with weighted edges.



Instead of solving the Manhattan Tourist problem directly, that is, finding the longest path from *source* $(0, 0)$ to *sink* (n, m), we solve a more general problem: find the longest path from *source* to an arbitrary vertex (i, j) with $0 \leq i \leq n$, $0 \leq j \leq m$. We will denote the length of such a best path as $s_{i,j}$, noticing that $s_{n,m}$ is the weight of the path that represents the solution to the

Manhattan Tourist problem. If we only care about the longest path between $(0, 0)$ and (n, m)—the Manhattan Tourist problem—then we have to answer one question, namely, what is the best way to get from *source* to *sink*. If we solve the general problem, then we have to answer $n \times m$ questions: what is the best way to get from *source* to anywhere. At first glance it looks like we have just created $n \times m$ different problems (computing (i, j) with $0 \leq i \leq n$ and $0 \leq j \leq m$) instead of a single one (computing $s_{n,m}$), but the fact that solving the more general problem is as easy as solving the Manhattan Tourist problem is the basis of dynamic programming. Note that DPCHANGE also generalized the problems that it solves by finding the optimal number of coins for *all* values less than or equal to M.

Finding $s_{0,j}$ (for $0 \leq j \leq m$) is not hard, since in this case the tourists do not have any flexibility in their choice of path. By moving strictly to the east, the weight of the path $s_{0,j}$ is the sum of weights of the first j city blocks. Similarly, $s_{i,0}$ is also easy to compute for $0 \leq i \leq n$, since the tourists move only to the south.

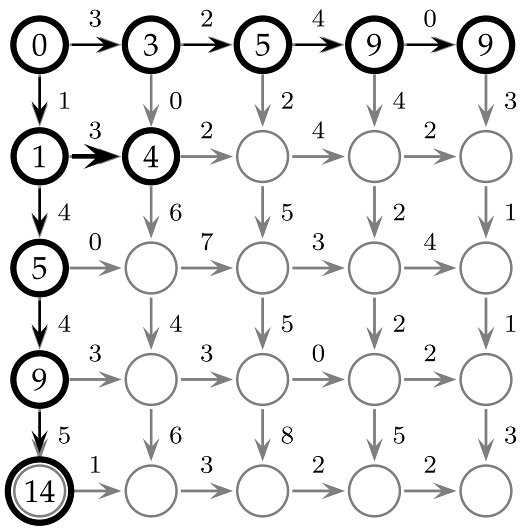


Now that we have figured out how to compute $s_{0,1}$ and $s_{1,0}$, we can compute $s_{1,1}$. The tourists can arrive at $(1, 1)$ in only two ways: either by traveling south from $(0, 1)$ or east from $(1, 0)$. The weight of each of these paths is

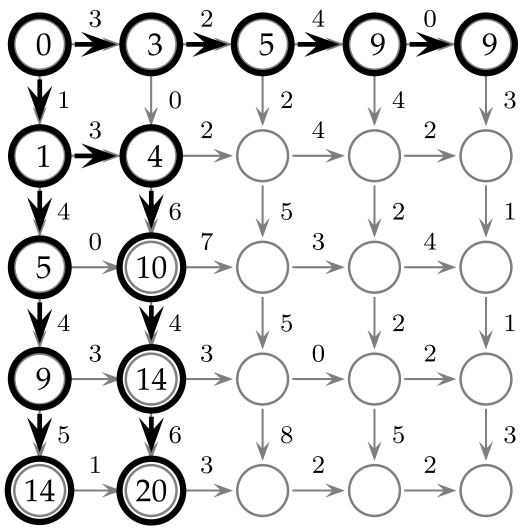
- $s_{0,1} +$ weight of the edge (block) between (0,1) and (1,1);

- $s_{1,0}+$ weight of the edge (block) between (1,0) and (1,1).

Since the goal is to find the longest path to, in this case, $(1,1)$, we choose the larger of the above two quantities: $3+0$ and $1+3$. Note that since there are no other ways to get to grid position $(1,1)$, we have found the longest path from $(0,0)$ to $(1,1)$.

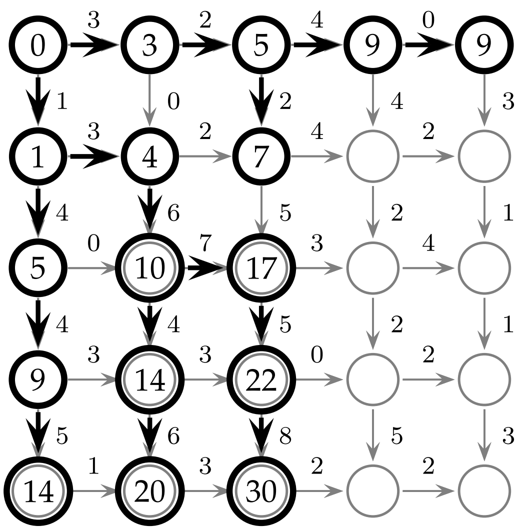


We have just found $s_{1,1}$. Similar logic applies to $s_{2,1}$, and then to $s_{3,1}$, and so on; once we have calculated $s_{i,0}$ for all i, we can calculate $s_{i,1}$ for all i.



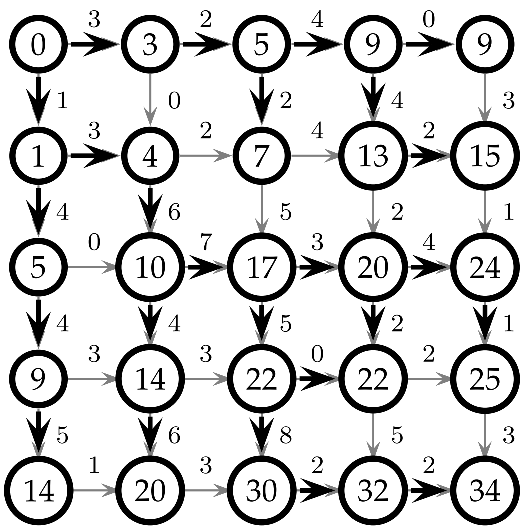
Once we have calculated $s_{i,1}$ for all i, we can use the same idea to calculate $s_{i,2}$ for all i, and so on. For example, we can calculate $s_{1,2}$ as follows.

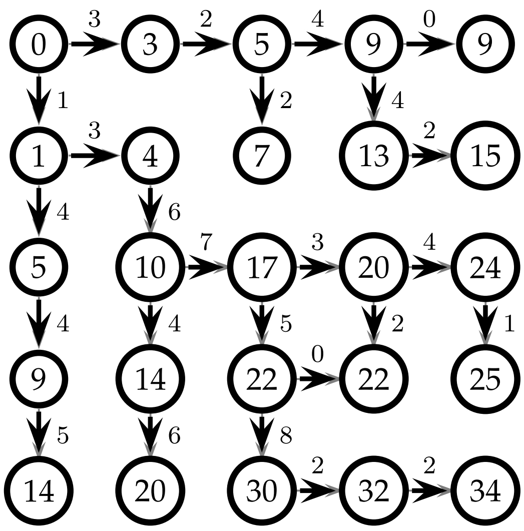
$$s_{1,2} = \max \begin{cases} s_{1,1} + \text{ weight of the edge between (1,1) and (1,2)} \\ s_{0,2} + \text{ weight of the edge between (0,2) and (1,2)} \end{cases}$$



In general, having the entire column $s_{*,j}$ allows us to compute the next whole column $s_{*,j+1}$. The observation that the only way to get to the intersection at (i,j) is either by moving south from intersection $(i-1,j)$ or by moving east from the intersection $(i,j-1)$ leads to the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{ weight of the edge between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{ weight of the edge between } (i,j-1) \text{ and } (i,j) \end{cases}$$





This recurrence allows us to compute every score $s_{i,j}$ in a single sweep of the grid. The algorithm MANHATTANTOURIST implements this procedure. Here, $\overset{\downarrow}{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run north to south, and $\overset{\rightarrow}{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run west to east. That is, $\overset{\downarrow}{w}_{i,j}$ is the weight of the edge between $(i, j-1)$ and (i, j); and $\vec{w}_{i,j}$ is the weight of the edge between $(i, j-1)$ and (i, j).

```
MANHATTANTOURIST($\overset{\downarrow}{\mathbf{w}}, \overset{\rightarrow}{\mathbf{w}}, n, m$)
1  $s_{0,0} \leftarrow 0$
2  for $i \leftarrow 1$ to $n$
3      $s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
4  for $j \leftarrow 1$ to $m$
5      $s_{0,j} \leftarrow s_{0,j-1} + \overset{\rightarrow}{w}_{0,j}$
6  for $i \leftarrow 1$ to $n$
7      for $j \leftarrow 1$ to $m$
8          $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} \\ s_{i,j-1} + \overset{\rightarrow}{w}_{i,j} \end{cases}$
9  return $s_{n,m}$
```

Lines 1 through 5 set up the *initial conditions* on the matrix s, and line 8 corresponds to the *recurrence* that allows us to fill in later table entries based on earlier ones. Most of the dynamic programming algorithms we will develop in the context of DNA sequence comparison will look just like MANHATTANTOURIST with only minor changes. We will generally just arrive at a recurrence like line 8 and call it an algorithm, with the understanding that the actual implementation will be similar to MANHATTANTOURIST.[6]

Many problems in bioinformatics can be solved efficiently by the application of the dynamic programming technique, once they are cast as traveling in a Manhattan-like grid. For example, development of new sequence comparison algorithms often amounts to building an appropriate "Manhattan" that adequately models the specifics of a particular biological problem, and by defining the block weights that reflect the costs of mutations from one DNA sequence to another.

6. MANHATTANTOURIST computes the length of the longest path in the grid, but does not give the path itself. In section 6.5 we will describe a minor modification to the algorithm that returns not only the optimal length, but also the optimal path.

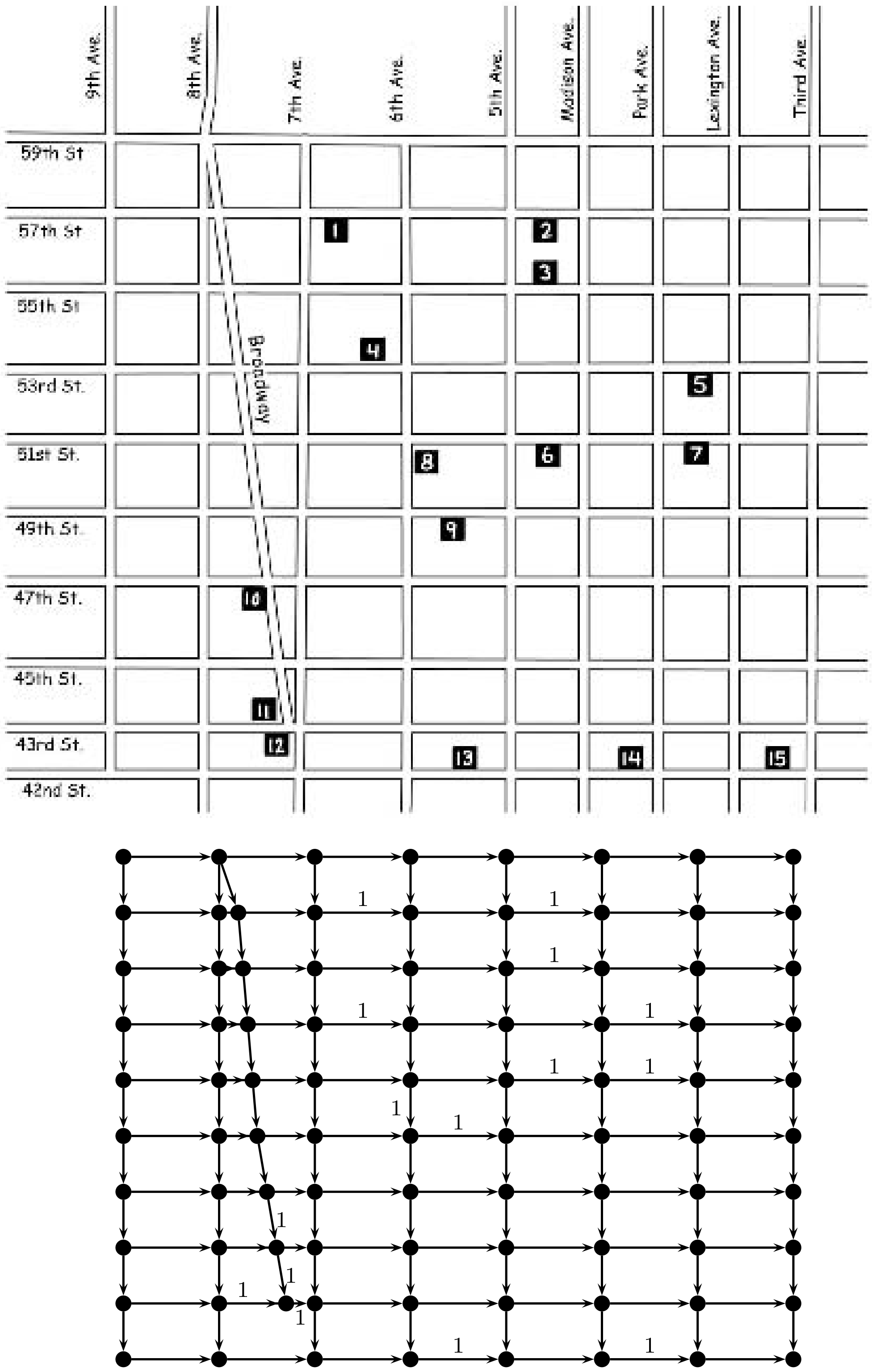


Figure 6.5 A city somewhat more like Manhattan than figure 6.4 with the complicating issue of a street that runs diagonally across the grid. Broadway cuts across several blocks. In the case of the Manhattan Tourist problem, it changes the optimal path (the optimal path in this new city has six attractions instead of five).

Unfortunately, Manhattan is not a perfectly regular grid. Broadway cuts across the borough (figure 6.5). We would like to solve a generalization of the Manhattan Tourist problem for the case in which the street map is not a regular rectangular grid. In this case, one can model any city map as a graph with vertices corresponding to the intersections of streets, and edges corresponding to the intervals of streets between the intersections. For the sake of simplicity we assume that the city blocks correspond to *directed* edges, so that the tourist can move only in the direction of the edge and that the resulting graph has no *directed cycles*.[7] Such graphs are called *directed acyclic graphs*, or *DAGs*. We assume that every edge has an associated weight (e.g., the number of attractions) and represent a graph G as a pair of two sets, V for vertices and E for edges: $G = (V, E)$. We number vertices from 1 to $|V|$ with a single integer, rather than a row-column pair as in the Manhattan problem. This does not change the generic dynamic programming algorithm other than in notation, but it allows us to represent imperfect grids. An edge from E can be specified in terms of its origin vertex u and its destination vertex v as (u, v). The following problem is simply a generalization of the Manhattan Tourist problem that is able to deal with arbitrary DAGs rather than with perfect grids.

Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

Output: A longest path in G from *source* to *sink*.

Not surprisingly, the Longest Path in a DAG problem can also be solved by dynamic programming. At every vertex, there may be multiple edges that "flow in" and multiple edges that "flow out." In the city analogy, any intersection may have multiple one-way streets leading in, and some other number of one-way streets exiting. We will call the number of edges entering a vertex (i.e., the number of inbound streets) the *indegree* of the vertex (i.e., intersection), and the number of edges leaving a vertex (i.e., the number of outbound streets) the *outdegree* of the vertex.

In the nicely regular case of the Manhattan problem, most vertices had

7. A directed cycle is a path from a vertex back to itself that respects the directions of edges. If the resulting graph contained a cycle, a tourist could start walking along this cycle revisiting the same attractions many times. In this case there is no "best" solution since a tourist may increase the number of visited attractions indefinitely.

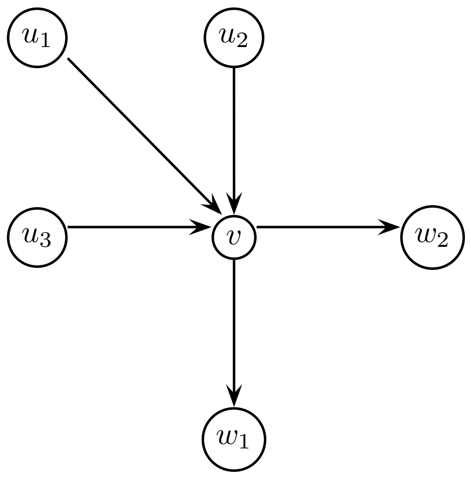


Figure 6.6 A graph with six vertices. The vertex v has indegree 3 and outdegree 2. The vertices u_1, u_2 and u_3 are all predecessors of v, and w_1 and w_2 are successors of v.

indegree 2 and outdegree 2, except for the vertices along the boundaries of the grid. In the more general DAG problem, a vertex can have an arbitrary indegree and outdegree. We will call u a *predecessor* to vertex v if $(u, v) \in E$—in other words, a predecessor of a vertex is any vertex that can be reached by traveling backwards along an inbound edge. Clearly, if v has indegree k, it has k predecessors.

Suppose a vertex v has indegree 3, and the set of predecessors of v is $\{u_1, u_2, u_3\}$ (figure 6.6). The longest path to v can be computed as follows:

$$s_v = \max \begin{cases} s_{u_1} + \text{ weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{ weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{ weight of edge from } u_3 \text{ to } v \end{cases}$$

In general, one can imagine a rather hectic city plan, but the recurrence relation remains simple, with the score s_v of the vertex v defined as follows.

$$s_v = \max_{u \in Predecessors(v)} (s_u + \text{ weight of edge from } u \text{ to } v)$$

Here, $Predecessors(v)$ is the set of all vertices u such that u is a predecessor of v. Since every edge participates in only a single recurrence, the running

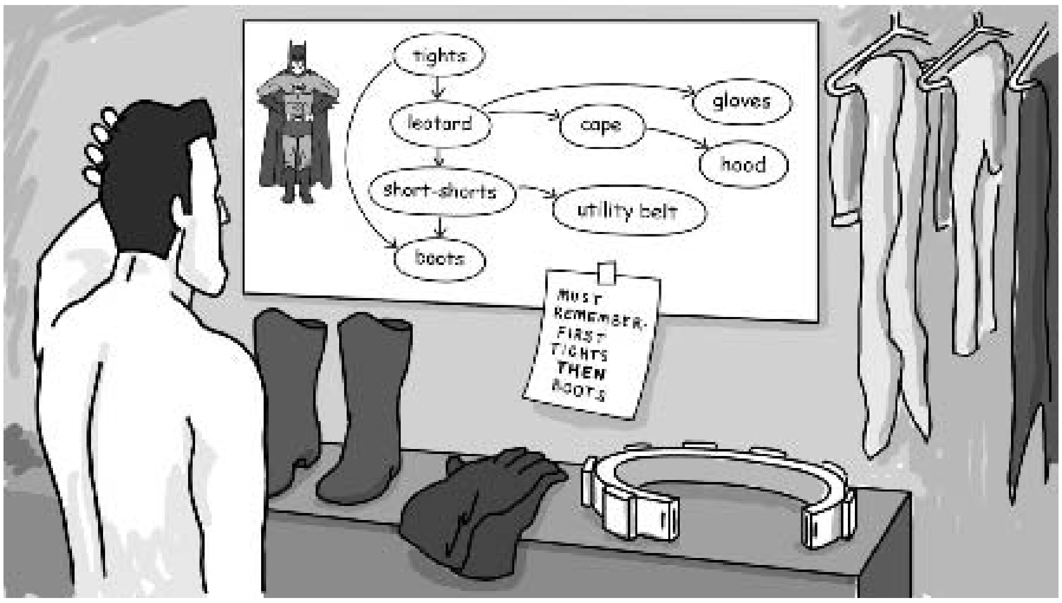


Figure 6.7 The "Dressing in the Morning problem" represented by a DAG. Some of us have more trouble than others.

time of the algorithm is defined by the number of edges in the graph.[8] The one hitch to this plan for solving the Longest Path problem in a DAG is that one must decide on the order in which to visit the vertices while computing **s**. This ordering is important, since by the time vertex v is analyzed, the values s_u for *all* its predecessors must have been computed. Three popular strategies for exploring the perfect grid are displayed in figure 6.9, column by column, row by row, and diagonal by diagonal. These exploration strategies correspond to different *topological orderings* of the DAG corresponding to the perfect grid. An ordering of vertices $v_1, \ldots, v_n$ of a DAG is called *topological* if every edge (v_i, v_j) of the DAG connects a vertex with a smaller index to a vertex with a larger index, that is, $i < j$. Figure 6.7 represents a DAG that corresponds to a problem that we each face every morning. Every DAG has a topological ordering (fig. 6.8); a problem at the end of this chapter asks you to prove this fact.

8. A graph with vertex set V can have at most $|V|^2$ edges, but graphs arising in sequence comparison are usually sparse, with many fewer edges.

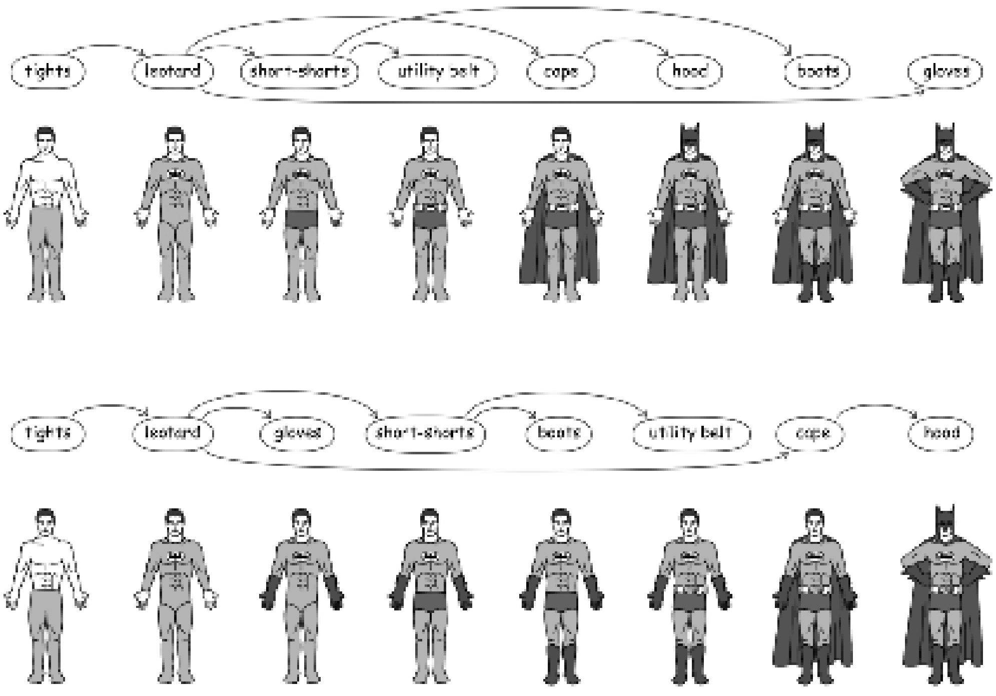


Figure 6.8 Two different ways of getting dressed in the morning corresponding to two different topological orderings of the graph in figure 6.7.

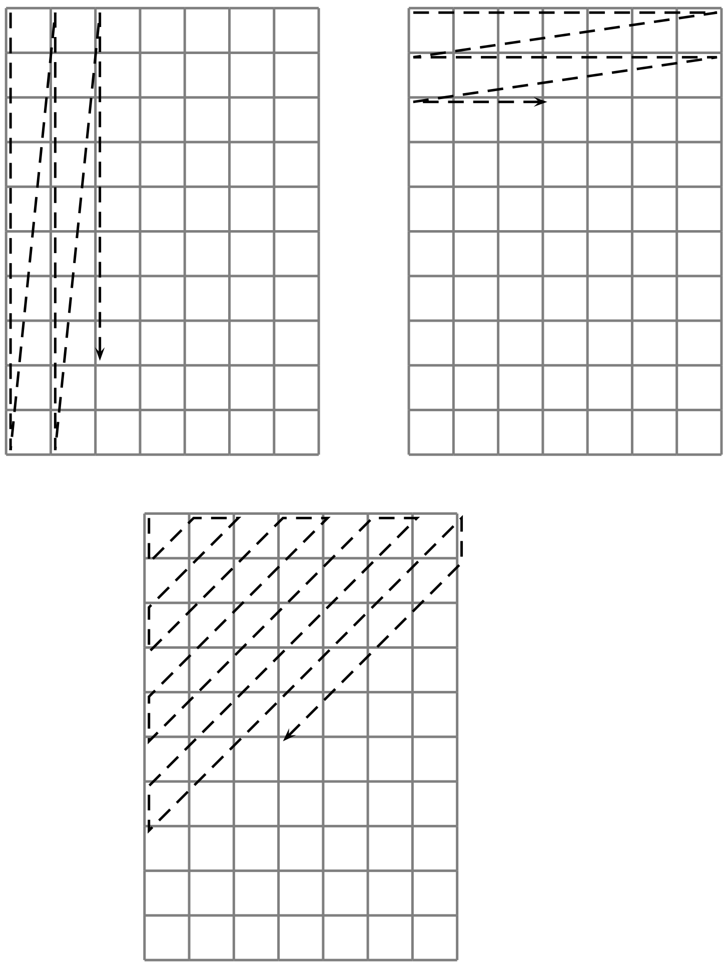


Figure 6.9 Three different strategies for filling in a dynamic programming array. The first fills in the array column by column: earlier columns are filled in before later ones. The second fills in the array row by row. The third method fills array entries along the diagonals and is useful in parallel computation.

6.4 Edit Distance and Alignments

So far, we have been vague about what we mean by "sequence similarity" or "distance" between DNA sequences. Hamming distance (introduced in chapter 4), while important in computer science, is not typically used to compare DNA or protein sequences. The Hamming distance calculation rigidly assumes that the ith symbol of one sequence is already *aligned* against the ith symbol of the other. However, it is often the case that the ith symbol in one sequence corresponds to a symbol at a different—and unknown—position in the other. For example, mutation in DNA is an evolutionary process: DNA replication errors cause substitutions, insertions, and deletions of nucleotides, leading to "edited" DNA texts. Since DNA sequences are subject to insertions and deletions, biologists rarely have the luxury of knowing in advance whether the ith symbol in one DNA sequence corresponds to the ith symbol in the other.

As figure 6.10 (a) shows, while strings ATATATAT and TATATATA are very different from the perspective of Hamming distance, they become very similar if one simply moves the second string over one place to align the $(i+1)$-st letter in ATATATAT against the ith letter in TATATATA for $1 \leq i \leq 7$. Strings ATATATAT and TATAAT present another example with more subtle similarities. Figure 6.10 (b) reveals these similarities by aligning position 2 in ATATATAT against position 1 in TATAAT. Other pairs of aligned positions are 3 against 2, 4 against 3, 5 against 4, 7 against 5, and 8 against 6 (positions 1 and 6 in ATATATAT remain unaligned).

In 1966, Vladimir Levenshtein introduced the notion of the *edit distance* between two strings as the minimum number of editing operations needed to transform one string into another, where the edit operations are insertion of a symbol, deletion of a symbol, and substitution of one symbol for another. For example, TGCATAT can be transformed into ATCCGAT with five editing operations, shown in figure 6.11. This implies that the edit distance between TGCATAT and ATCCGAT is at most 5. Actually, the edit distance between them is 4 because you can transform one to the other with one move fewer, as in figure 6.12.

Unlike Hamming distance, edit distance allows one to compare strings of different lengths. Oddly, Levenshtein introduced the definition of edit distance but never described an algorithm for actually finding the edit distance between two strings. This algorithm has been discovered and rediscovered many times in applications ranging from automated speech recognition to, obviously, molecular biology. Although the details of the algorithms are

```
A T A T A T A T -
  : : : : : : :
- T A T A T A T A
```

(a) Alignment of ATATATAT against TATATATA.

```
A T A T A T A T
  : : : :   : :
- T A T A - A T
```

(b) Alignment of ATATATAT against TATAAT.

Figure 6.10 Alignment of ATATATAT against TATATATA and of ATATATAT against TATAAT.

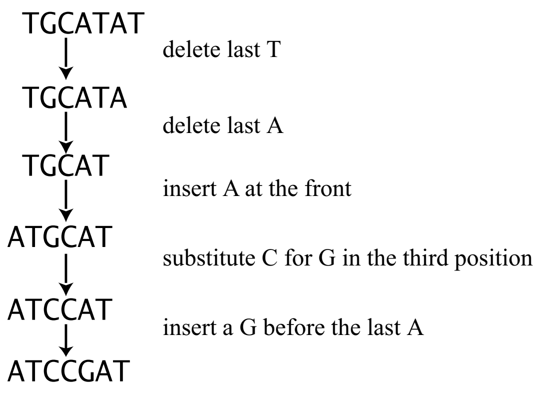


Figure 6.11 Five edit operations can take TGCATAT into ATCCGAT.

slightly different across the various applications, they are all based on dynamic programming.

The *alignment* of the strings **v** (of n characters) and **w** (of m characters, with m not necessarily the same as n) is a two-row matrix such that the first row contains the characters of **v** in order while the second row contains the characters of **w** in order, where spaces may be interspersed throughout the strings in different places. As a result, the characters in each string appear in order, though not necessarily adjacently. We also assume that no column

$$\begin{tikzcd}
\text{TGCATAT} \arrow[d, "\text{insert A at the front}"] \\
\text{ATGCATAT} \arrow[d, "\text{delete T in the sixth position}"] \\
\text{ATGCAAT} \arrow[d, "\text{substitute G for A in the fifth position}"] \\
\text{ATGCGAT} \arrow[d, "\text{substitute C for G in the third position}"] \\
\text{ATCCGAT}
\end{tikzcd}$$

Figure 6.12 Four edit operations can also take TGCATAT into ATCCGAT.

of the alignment matrix contains spaces in both rows, so that the alignment may have at most $n + m$ columns.

A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Columns that contain the same letter in both rows are called *matches*, while columns containing different letters are called *mismatches*. The columns of the alignment containing one space are called *indels*, with the columns containing a space in the top row called *insertions* and the columns with a space in the bottom row *deletions*. The alignment shown in figure 6.13 (top) has five matches, zero mismatches, and four indels. The number of matches plus the number of mismatches plus the number of indels is equal to the length of the alignment matrix and must be smaller than $n + m$.

Each of the two rows in the alignment matrix is represented as a string interspersed by space symbols "$-$"; for example AT$-$GTTAT$-$ is a representation of the row corresponding to $\mathbf{v}$ = ATGTTAT, while ATCGT$-$A$-$C is a representation of the row corresponding to $\mathbf{w}$ = ATCGTAC. Another way to represent the row AT$-$GTTAT$-$ is $1\,2\,2\,3\,4\,5\,6\,7\,7$, which shows the number of symbols of $\mathbf{v}$ present up to a given position. Similarly, ATCGT$-$A$-$C is represented as $1\,2\,3\,4\,5\,5\,6\,6\,7$. When both rows of an alignment are represented in this way (fig. 6.13, top), the resulting matrix is

$$\begin{pmatrix}0\\0\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix}\begin{pmatrix}2\\2\end{pmatrix}\begin{pmatrix}2\\3\end{pmatrix}\begin{pmatrix}3\\4\end{pmatrix}\begin{pmatrix}4\\5\end{pmatrix}\begin{pmatrix}5\\5\end{pmatrix}\begin{pmatrix}6\\6\end{pmatrix}\begin{pmatrix}7\\6\end{pmatrix}\begin{pmatrix}7\\7\end{pmatrix}$$

Each column in this matrix is a coordinate in a two-dimensional $n \times m$ grid;

the entire alignment is simply a path

$$(0,0) \to (1,1) \to (2,2) \to (2,3) \to (3,4) \to (4,5) \to (5,5) \to (6,6) \to (7,6) \to (7,7)$$

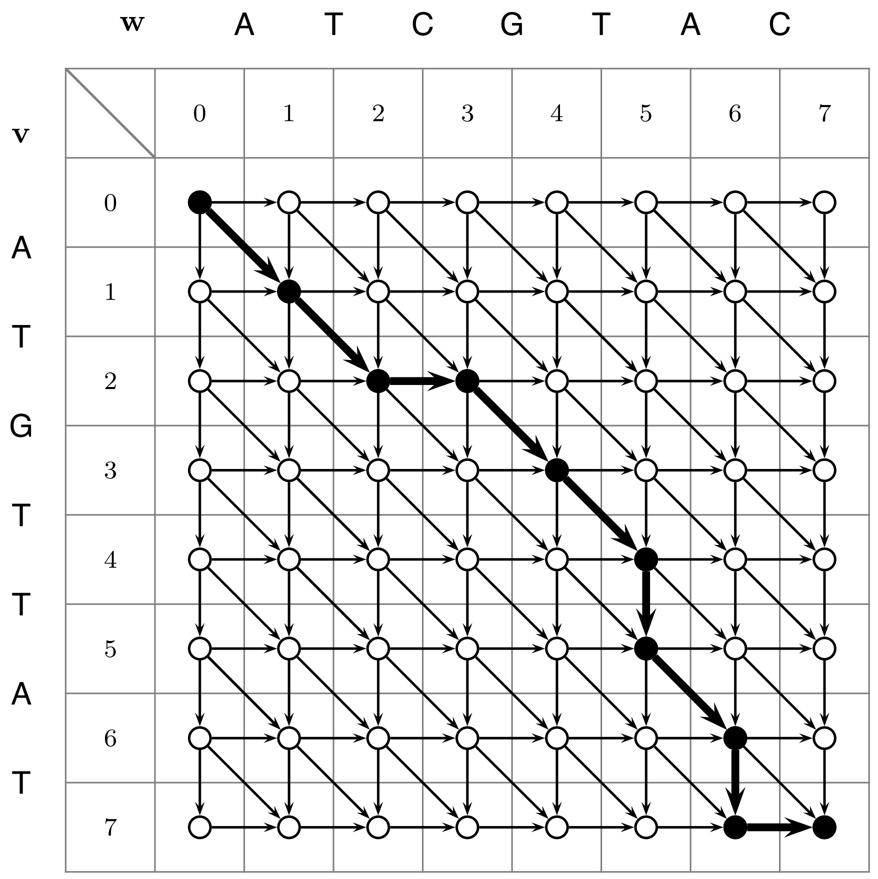
from $(0,0)$ to (n,m) in that grid (again, see figure 6.13). This grid is similar to the Manhattan grid that we introduced earlier, where each entry in the grid looks like a city block. The main difference is that here we can move along the diagonal. We can construct a graph, this time called the *edit graph*, by introducing a vertex for every intersection of streets in the grid, shown in figure 6.13. The edit graph will aid us in calculating the edit distance.

Every alignment corresponds to a path in the edit graph, and every path in the edit graph corresponds to an alignment where every edge in the path corresponds to one column in the alignment (fig. 6.13). Diagonal edges in the path that end at vertex (i,j) in the graph correspond to the column $\begin{pmatrix} v_i \\ w_j \end{pmatrix}$, horizontal edges correspond to $\begin{pmatrix} - \\ w_j \end{pmatrix}$, and vertical edges correspond to $\begin{pmatrix} v_i \\ - \end{pmatrix}$. The alignment above can be drawn as follows.

$$\begin{array}{cccccccccc} & \text{A} & \text{T} & - & \text{G} & \text{T} & \text{T} & \text{A} & \text{T} & - \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 2 \\ 2 \end{pmatrix} & \begin{pmatrix} 2 \\ 3 \end{pmatrix} & \begin{pmatrix} 3 \\ 4 \end{pmatrix} & \begin{pmatrix} 4 \\ 5 \end{pmatrix} & \begin{pmatrix} 5 \\ 5 \end{pmatrix} & \begin{pmatrix} 6 \\ 6 \end{pmatrix} & \begin{pmatrix} 7 \\ 6 \end{pmatrix} & \begin{pmatrix} 7 \\ 7 \end{pmatrix} \\ & \text{A} & \text{T} & \text{G} & \text{C} & \text{T} & - & \text{A} & - & \text{C} \end{array}$$

Analyzing the merit of an alignment is equivalent to analyzing the merit of the corresponding path in the edit graph. Given any two strings, there are a large number of different alignment matrices and corresponding paths in the edit graph. Some of these have a surplus of mismatches and indels and a small number of matches, while others have many matches and few indels and mismatches. To determine the relative merits of one alignment over another, we rely on the notion of a scoring function, which takes as input an alignment matrix (or, equivalently, a path in the edit graph) and produces a score that determines the “goodness” of the alignment. There are a variety of scoring functions that we could use, but we want one that gives higher scores to alignments with more matches. The simplest functions score a column as a positive number if both letters are the same, and as a negative number if the two letters are different. The score for the whole alignment is the sum of the individual column scores. This scoring scheme amounts to

$$\begin{array}{cccccccccccc}
 & & 0 & 1 & 2 & 2 & 3 & 4 & 5 & 6 & 7 & 7 \\
\mathbf{v} & = & & \mathsf{A} & \mathsf{T} & - & \mathsf{G} & \mathsf{T} & \mathsf{T} & \mathsf{A} & \mathsf{T} & - \\
 & & & | & | & & | & | & & | & & \\
\mathbf{w} & = & & \mathsf{A} & \mathsf{T} & \mathsf{C} & \mathsf{G} & \mathsf{T} & - & \mathsf{A} & - & \mathsf{C} \\
 & & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 6 & 7
\end{array}$$



$$\begin{array}{ccccccccc}
\searrow & \searrow & \rightarrow & \searrow & \searrow & \downarrow & \searrow & \downarrow & \rightarrow \\
\mathrm{A} & \mathrm{T} & - & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} & \mathrm{T} & - \\
\mathrm{A} & \mathrm{T} & \mathrm{C} & \mathrm{G} & \mathrm{T} & - & \mathrm{A} & - & \mathrm{C}
\end{array}$$

Figure 6.13 An alignment grid for $\mathbf{v} = \mathsf{ATGTTAT}$ and $\mathbf{w} = \mathsf{ATCGTAC}$. Every alignment corresponds to a path in the alignment grid from $(0, 0)$ to (n, m), and every path from $(0, 0)$ to (n, m) in the alignment grid corresponds to an alignment.

assigning weights to the edges in the edit graph.

By choosing different scoring functions, we can solve different string comparison problems. If we choose the very simple scoring function of "$+1$ for a match, 0 otherwise," then the problem becomes that of finding the longest common subsequence between two strings, which is discussed below. Before describing how to calculate Levenshtein's edit distance, we develop the Longest Common Subsequence problem as a warm-up.

6.5 Longest Common Subsequences

The simplest form of a sequence similarity analysis is the Longest Common Subsequence (LCS) problem, where we eliminate the operation of substitution and allow only insertions and deletions. A *subsequence* of a string $\mathbf{v}$ is simply an (ordered) sequence of characters (not necessarily consecutive) from $\mathbf{v}$. For example, if $\mathbf{v}$ = ATTGCTA, then AGCA and ATTA are subsequences of $\mathbf{v}$ whereas TGTT and TCG are not.[9] A *common* subsequence of two strings is a subsequence of both of them. Formally, we define the *common subsequence* of strings $\mathbf{v} = v_1 \ldots v_n$ and $\mathbf{w} = w_1 \ldots w_m$ as a sequence of positions in $\mathbf{v}$,

$$1 \leq i_1 < i_2 < \cdots < i_k \leq n$$

and a sequence of positions in $\mathbf{w}$,

$$1 \leq j_1 < j_2 < \cdots < j_k \leq m$$

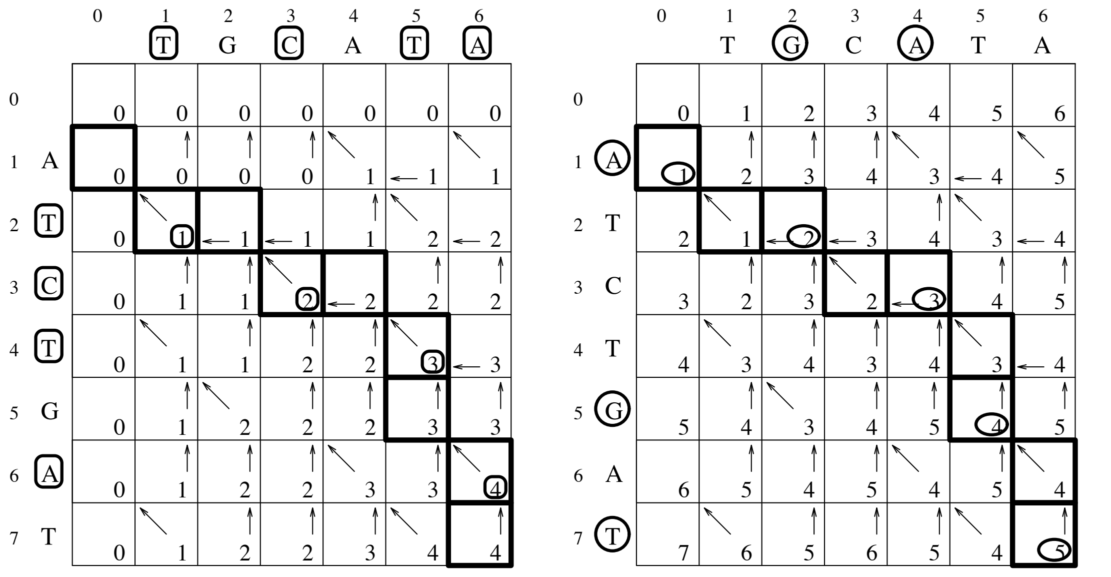
such that the symbols at the corresponding positions in $\mathbf{v}$ and $\mathbf{w}$ coincide:

$$v_{i_t} = w_{j_t} \text{ for } 1 \leq t \leq k.$$

For example, TCTA is a common to both ATCTGAT and TGCATA.

Although there are typically many common subsequences between two strings $\mathbf{v}$ and $\mathbf{w}$, some of which are longer than others, it is not immediately obvious how to find the longest one. If we let $s(\mathbf{v}, \mathbf{w})$ be the length of the longest common subsequence of $\mathbf{v}$ and $\mathbf{w}$, then the edit distance between $\mathbf{v}$ and $\mathbf{w}$—under the assumption that only insertions and deletions are allowed—is $d(\mathbf{v}, \mathbf{w}) = n + m - 2s(\mathbf{v}, \mathbf{w})$, and corresponds to the mini-

9. The difference between a sub*sequence* and a sub*string* is that a substring consists only of consecutive characters from $\mathbf{v}$, while a subsequence may pick and choose characters from $\mathbf{v}$ as long as their ordering is preserved.


Computing similarity s(V,W)=4
V and W have a subsequence TCTA in common
Computing distance d(V,W)=5
V can be transformed into W by deleting A,G,T and inserting G,A

Alignment:

```
A T - C - T G A T
- T G C A T - A -
```

Figure 6.14 Dynamic programming algorithm for computing the longest common subsequence.

mum number of insertions and deletions needed to transform **v** into **w**. Figure 6.14 (bottom) presents an LCS of length 4 for the strings **v** = ATCTGAT and **w** = TGCATA and a shortest sequence of two insertions and three deletions transforming **v** into **w** (shown by "-" in the figure). The LCS problem follows.

Longest Common Subsequence Problem:
Find the longest subsequence common to two strings.

Input: Two strings, **v** and **w**.

Output: The longest common subsequence of **v** and **w**.

What do the LCS problem and the Manhattan Tourist problem have in common? Every common subsequence corresponds to an alignment with no

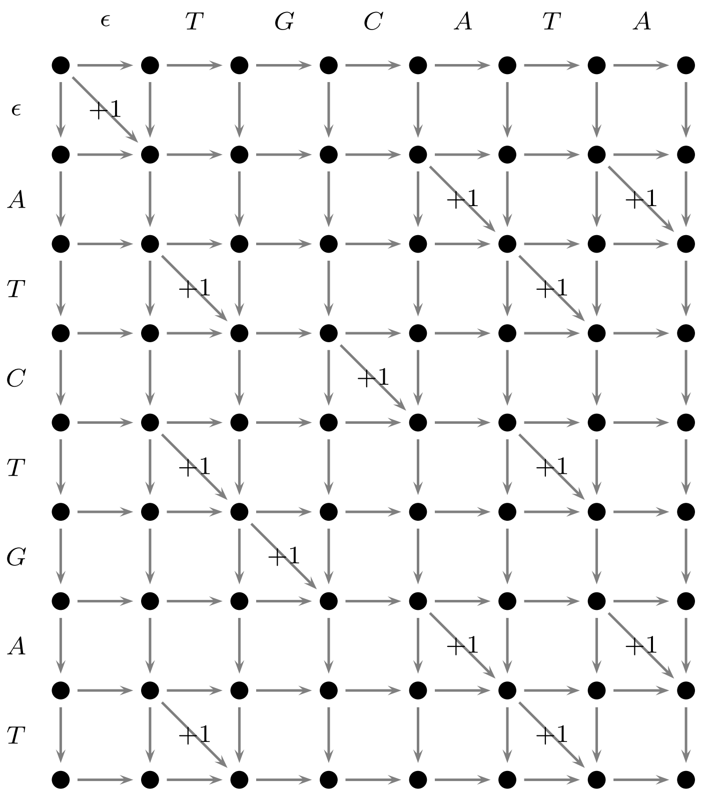


Figure 6.15 An LCS edit graph.

mismatches. This can be obtained simply by removing all diagonal edges from the edit graph whose characters do not match, thus transforming it into a graph like that shown in figure 6.15. We further illustrate the relationship between the Manhattan Tourist problem and the LCS Problem by showing that these two problems lead to very similar recurrences.

Define $s_{i,j}$ to be the length of an LCS between $v_1 \ldots v_i$, the i-prefix of $\mathbf{v}$ and $w_1 \ldots w_j$, the j-prefix of $\mathbf{w}$. Clearly, $s_{i,0} = s_{0,j} = 0$ for all $1 \le i \le n$ and

$1 \leq j \leq m$. One can see that $s_{i,j}$ satisfies the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

The first term corresponds to the case when v_i is not present in the LCS of the i-prefix of $\mathbf{v}$ and j-prefix of $\mathbf{w}$ (this is a deletion of v_i); the second term corresponds to the case when w_j is not present in this LCS (this is an insertion of w_j); and the third term corresponds to the case when both v_i and w_j are present in the LCS (v_i *matches* w_j). Note that one can "rewrite" these recurrences by adding some zeros here and there as

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

This recurrence for the LCS computation is like the recurrence given at the end of the section 6.3, if we were to build a particularly gnarly version of Manhattan and gave horizontal and vertical edges weights of 0, and set the weights of diagonal (matching) edges equal to $+1$ as in figure 6.15.

In the following, we use $\mathbf{s}$ to represent our dynamic programming table, the data structure that we use to fill in the dynamic programming recurrence. The length of an LCS between $\mathbf{v}$ and $\mathbf{w}$ can be read from the element (n, m) of the dynamic programming table, but to reconstruct the LCS from the dynamic programming table, one must keep some additional information about which of the three quantities, $s_{i-1,j}$, $s_{i,j-1}$, or $s_{i-1,j-1} + 1$, corresponds to the maximum in the recurrence for $s_{i,j}$. The following algorithm achieves this goal by introducing *backtracking pointers* that take one of the three values $\leftarrow$, $\uparrow$, or $\nwarrow$. These specify which of the above three cases holds, and are stored in a two-dimensional array $\mathbf{b}$ (see figure 6.14).

```
LCS($\mathbf{v}, \mathbf{w}$)
1  for $i \leftarrow 0$ to $n$
2      $s_{i,0} \leftarrow 0$
3  for $j \leftarrow 1$ to $m$
4      $s_{0,j} \leftarrow 0$
5  for $i \leftarrow 1$ to $n$
6      for $j \leftarrow 1$ to $m$
7          $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$
8          $b_{i,j} \leftarrow \begin{cases} \text{“}\uparrow\text{''} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{''} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{''}, & \text{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$
9  return $(s_{n,m}, \mathbf{b})$
```

The following recursive program prints out the longest common subsequence using the information stored in $\mathbf{b}$. The initial invocation that prints the solution to the problem is PRINTLCS($\mathbf{b}, \mathbf{v}, n, m$).

```
PRINTLCS($\mathbf{b}, \mathbf{v}, i, j$)
 1  if $i = 0$ or $j = 0$
 2      return
 3  if $b_{i,j} =$ “$\nwarrow$''
 4      PRINTLCS($\mathbf{b}, \mathbf{v}, i - 1, j - 1$)
 5      print $v_i$
 6  else
 7      if $b_{i,j} =$ “$\uparrow$''
 8          PRINTLCS($\mathbf{b}, \mathbf{v}, i - 1, j$)
 9      else
10          PRINTLCS($\mathbf{b}, \mathbf{v}, i, j - 1$)
```

The dynamic programming table in figure 6.14 (left) presents the computation of the similarity score $s(\mathbf{v}, \mathbf{w})$ between $\mathbf{v}$ and $\mathbf{w}$, while the table on the right presents the computation of the edit distance between $\mathbf{v}$ and $\mathbf{w}$ under the assumption that insertions and deletions are the only allowed operations. The edit distance $d(\mathbf{v}, \mathbf{w})$ is computed according to the initial conditions $d_{i,0} = i$, $d_{0,j} = j$ for all $1 \le i \le n$ and $1 \le j \le m$ and the following recurrence:

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \\ d_{i-1,j-1}, & \text{if } v_i = w_j \end{cases}$$

6.6 Global Sequence Alignment

The LCS problem corresponds to a rather restrictive scoring that awards 1 for matches and does not penalize indels. To generalize scoring, we extend the k-letter alphabet $\mathcal{A}$ to include the gap character "$-$", and consider an arbitrary $(k+1) \times (k+1)$ *scoring matrix* δ, where k is typically 4 or 20 depending on the type of sequences (DNA or protein) one is analyzing. The score of the column $\binom{x}{y}$ in the alignment is $\delta(x, y)$ and the alignment score is defined as the sum of the scores of the columns. In this way we can take into account scoring of mismatches and indels in the alignment. Rather than choosing a particular scoring matrix and then resolving a restated alignment problem, we will pose a general Global Alignment problem that takes the scoring matrix as input.

Global Alignment Problem:
Find the best alignment between two strings under a given scoring matrix.

Input: Strings $\mathbf{v}$, $\mathbf{w}$ and a scoring matrix δ.

Output: An alignment of $\mathbf{v}$ and $\mathbf{w}$ whose score (as defined by the matrix δ) is maximal among all possible alignments of $\mathbf{v}$ and $\mathbf{w}$.

The corresponding recurrence for the score $s_{i,j}$ of an optimal alignment between the i-prefix of $\mathbf{v}$ and j-prefix of $\mathbf{w}$ is as follows:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

When mismatches are penalized by some constant $-\mu$, indels are penalized by some other constant $-\sigma$, and matches are rewarded with $+1$, the resulting score is

$$\#matches - \mu \cdot \#mismatches - \sigma \cdot \#indels$$

The corresponding recurrence can be rewritten as

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} - \mu, \text{ if } v_i \neq w_j \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$$

We can again store similar "backtracking pointer" information while calculating the dynamic programming table, and from this reconstruct the alignment. We remark that the LCS problem is the Global Alignment problem with the parameters $\mu = 0$, $\sigma = 0$ (or, equivalently, $\mu = \infty$, $\sigma = 0$).

6.7 Scoring Alignments

While the scoring matrices for DNA sequence comparison are usually defined only by the parameters μ (mismatch penalty) and σ (indel penalty), scoring matrices for sequences in the amino acid alphabet of proteins are quite involved. The common matrices for protein sequence comparison, *point accepted mutations (PAM)* and *block substitution (BLOSUM)*, reflect the frequency with which amino acid x replaces amino acid y in evolutionarily related sequences.

Random mutations of the nucleotide sequence within a gene may change the amino acid sequence of the corresponding protein. Some of these mutations do not drastically alter the protein's structure, but others do and impair the protein's ability to function. While the former mutations usually do not affect the fitness of the organism, the latter often do. Therefore some amino acid substitutions are commonly found throughout the process of molecular evolution and others are rare: `Asn, Asp, Glu`, and `Ser` are the most "mutable" amino acids while `Cys` and `Trp` are the least mutable. For example, the probability that `Ser` mutates into `Phe` is roughly three times greater than the probability that `Trp` mutates into `Phe`. Knowledge of the types of changes that are most and least common in molecular evolution allows biologists to construct the amino acid scoring matrices and to produce biologically adequate sequence alignments. As a result, in contrast to nucleotide sequence comparison, the optimal alignments of amino acid sequences may have very few matches (if any) but still represent biologically adequate alignments. The entry of amino acid scoring matrix $\delta(i, j)$ usually reflects how often the amino acid i substitutes the amino acid j in the alignments of related protein sequences. If one is provided with a large set of alignments of related sequences, then computing $\delta(i,j)$ simply amounts to counting how many times the amino acid i is aligned with amino acid j. A "minor" complication is that to build this set of biologically adequate alignments one needs to know the scoring matrix! Fortunately, in many cases the alignment of very similar sequences is so obvious that it can be constructed even without a scoring matrix, thus resolving this predicament. For example, if proteins are 90% identical, even a naive scoring matrix (e.g., a matrix that gives premium $+1$ for matches and penalties -1 for mismatches and indels) would do the job. After these "obvious" alignments are constructed they can be used to compute a scoring matrix δ that can be used iteratively to construct less obvious alignments.

This simplified description hides subtle details that are important in the construction of scoring matrices. The probability of `Ser` mutating into `Phe` in proteins that diverged 15 million years ago (e.g., related proteins in mouse and rat) is smaller than the probability of the `Ser` $\rightarrow$ `Phe` mutation in proteins that diverged 80 million years ago (e.g., related proteins in mouse and human). This observation implies that the best scoring matrices to compare two proteins depends on how similar these organisms are.

Biologists get around this problem by first analyzing extremely similar proteins, for example, proteins that have, on average, only one mutation per 100 amino acids. Many proteins in human and chimpanzee fulfill this requirement. Such sequences are defined as being *one PAM unit diverged* and to a first approximation one can think of a PAM unit as the amount of time in which an "average" protein mutates 1% of its amino acids. The *PAM 1* scoring matrix is defined from many alignments of extremely similar proteins as follows.

Given a set of base alignments, define $f(i,j)$ as the total number of times amino acids i and j are aligned against each other, divided by the total number of aligned positions. We also define $g(i,j)$ as $\frac{f(i,j)}{f(i)}$, where $f(i)$ is the frequency of amino acid i in all proteins from the data set. $g(i,j)$ defines the probability that an amino acid i mutates into amino acid j within 1 PAM unit. The (i,j) entry of the *PAM 1* matrix is defined as $\delta(i,j) = \log \frac{f(i,j)}{f(i)\cdot f(j)} = \log \frac{g(i,j)}{f(j)}$ ($f(i) \cdot f(j)$ stands for the frequency of aligning amino acid i against amino acid j that one expects simply by chance). The *PAM n* matrix can be defined as the result of applying the PAM 1 matrix n times. If $\mathbf{g}$ is the 20×20 matrix of frequencies $g(i,j)$, then $\mathbf{g}^n$ (multiplying this matrix by itself n times) gives the probability that amino acid i mutates into amino acid j during n PAM units. The (i,j) entry of the PAM n matrix is defined as

$\log \frac{g^n_{i,j}}{f(j)}$.

For large n, the resulting PAM matrices often allow one to find related proteins even when there are practically no matches in the alignment. In this case, the underlying nucleotide sequences are so diverged that their comparison usually fails to find any statistically significant similarities. For example, the similarity between the cancer-causing ν-sis oncogene and the growth factor PDGF would probably have remained undetected had Russell Doolittle and colleagues not transformed the nucleotide sequences into amino acid sequences prior to performing the comparison.

6.8 Local Sequence Alignment

The Global Alignment problem seeks similarities between two entire strings. This is useful when the similarity between the strings extends over their entire length, for example, in protein sequences from the same protein family. These protein sequences are often very conserved and have almost the same length in organisms ranging from fruit flies to humans. However, in many biological applications, the score of an alignment between two substrings of $\mathbf{v}$ and $\mathbf{w}$ might actually be larger than the score of an alignment between the entireties of $\mathbf{v}$ and $\mathbf{w}$.

For example, *homeobox* genes, which regulate embryonic development, are present in a large variety of species. Although homeobox genes are very different in different species, one region in each gene—called the *homeodomain*—is highly conserved. The question arises how to find this conserved area and ignore the areas that show little similarity. In 1981 Temple Smith and Michael Waterman proposed a clever modification of the global sequence alignment dynamic programming algorithm that solves the Local Alignment problem.

Figure 6.16 presents the comparison of two hypothetical genes $\mathbf{v}$ and $\mathbf{w}$ of the same length with a conserved domain present at the beginning of $\mathbf{v}$ and at the end of $\mathbf{w}$. For simplicity, we will assume that the conserved domains in these two genes are identical and cover one third of the entire length, n, of these genes. In this case, the path from $source$ to $sink$ capturing the similarity between the homeodomains will include approximately $\frac{2}{3}n$ horizontal edges, $\frac{1}{3}n$ diagonal match edges (corresponding to homeodomains), and $\frac{2}{3}n$ vertical edges. Therefore, the score of this path is

$$-\frac{2}{3}n\sigma + \frac{1}{3}n - \frac{2}{3}n\sigma = n\left(\frac{1}{3} - \frac{4}{3}\sigma\right)$$

However, this path contains so many indels that it is unlikely to be the highest scoring alignment. In fact, biologically irrelevant diagonal paths from the source to the sink will likely have a higher score than the biologically relevant alignment, since mismatches are usually penalized less than indels. The expected score of such a diagonal path is $n(\frac{1}{4} - \frac{3}{4}\mu)$ since every diagonal edge corresponds to a match with probability $\frac{1}{4}$ and mismatch with probability $\frac{3}{4}$. Since $(\frac{1}{3} - \frac{4}{3}\sigma) < (\frac{1}{4} - \frac{3}{4}\mu)$ for many settings of indel and mismatch penalties, the global alignment algorithm will miss the correct solution of the real biological problem, and is likely to output a biologically irrelevant near-diagonal path. Indeed, figure 6.16 bears exactly this observation.

When biologically significant similarities are present in certain parts of DNA fragments and are not present in others, biologists attempt to maximize the alignment score $s(v_i \ldots v_{i'}, w_j \ldots w_{j'})$, over all substrings $v_i \ldots v_{i'}$ of $\mathbf{v}$ and $w_j \ldots w_{j'}$ of $\mathbf{w}$. This is called the Local Alignment problem since the alignment does not necessarily extend over the entire string length as it does in the Global Alignment problem.

Local Alignment Problem:
Find the best local alignment between two strings.

Input: Strings $\mathbf{v}$ and $\mathbf{w}$ and a scoring matrix δ.

Output: Substrings of $\mathbf{v}$ and $\mathbf{w}$ whose global alignment, as defined by δ, is maximal among all global alignments of all substrings of $\mathbf{v}$ and $\mathbf{w}$.

The solution to this seemingly harder problem lies in the realization that the Global Alignment problem corresponds to finding the longest local path between vertices $(0, 0)$ and (n, m) in the edit graph, while the Local Alignment problem corresponds to finding the longest path among paths between *arbitrary vertices* (i, j) and (i', j') in the edit graph. A straightforward and inefficient approach to this problem is to find the longest path between every pair of vertices (i, j) and (i', j'), and then to select the longest of these computed paths.[10] Instead of finding the longest path from every vertex (i, j) to every other vertex (i', j'), the Local Alignment problem can be reduced to finding the longest paths from the *source* (0,0) to every other vertex by

10. This will result in a very slow algorithm with $O(n^4)$ running time: there are roughly n^2 pairs of vertices (i, j) and computing local alignments starting at each of them typically takes $O(n^2)$ time.

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
```

```
                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

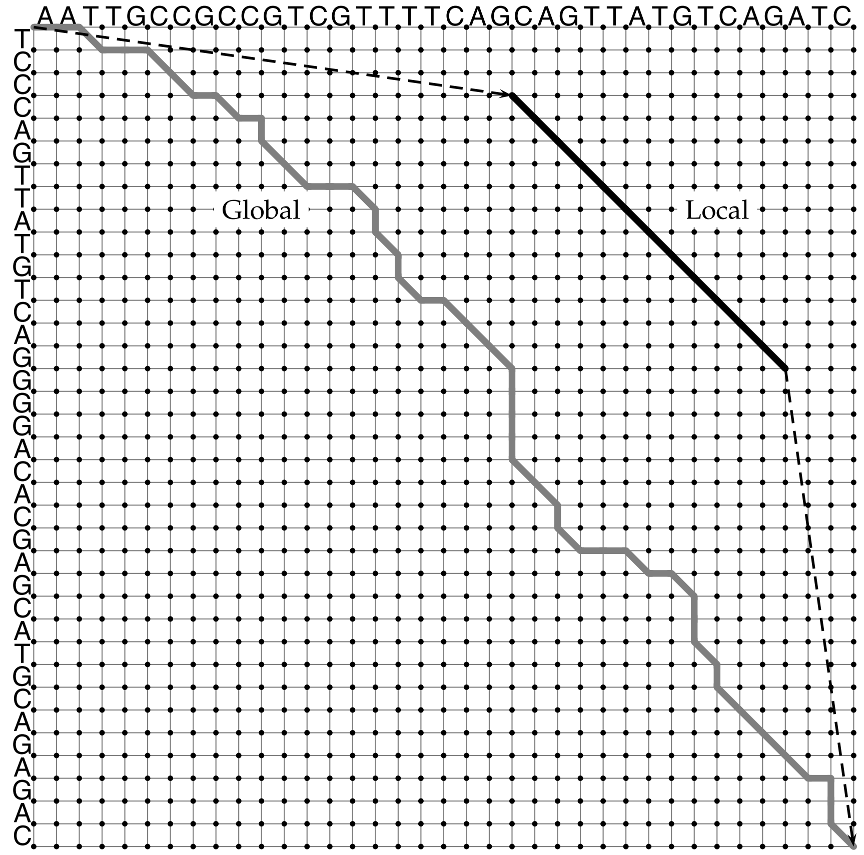

AATTGCCGCCGTCGTTTTCAGCAGTTATGTCAGATC
TCCCAGTTATGTCAGGGGACACGAGCATGCAGAGAC
Global
Local

Figure 6.16 (a) Global and (b) local alignments of two hypothetical genes that each have a conserved domain. The local alignment has a much worse score according to the global scoring scheme, but it correctly locates the conserved domain.

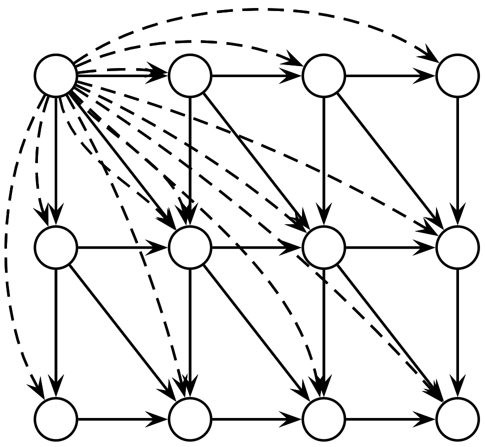


Figure 6.17 The Smith-Waterman local alignment algorithm introduces edges of weight 0 (here shown with dashed lines) from the source vertex $(0, 0)$ to every other vertex in the edit graph.

adding edges of weight 0 in the edit graph. These edges make the source vertex (0,0) a predecessor of every vertex in the graph and provide a "free ride" from the source to any other vertex (i, j). A small difference in the following recurrence reflects this transformation of the edit graph (shown in figure 6.17):

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

The largest value of $s_{i,j}$ over the whole edit graph represents the score of the best local alignment of $\mathbf{v}$ and $\mathbf{w}$; recall that in the Global Alignment problem, we simply looked at $s_{n,m}$. The difference between local and global alignment is illustrated in figure 6.16 (top).

Optimal local alignment reports only the longest path in the edit graph. At the same time, several local alignments may have biological significance and methods have been developed to find the k best nonoverlapping local alignments. These methods are particularly important for comparison of multidomain proteins that share similar blocks that have been shuffled in one protein compared to another. In this case, a single local alignment representing all significant similarities may not exist.

6.9 Alignment with Gap Penalties

Mutations are usually caused by errors in DNA replication. Nature frequently deletes or inserts entire substrings as a unit, as opposed to deleting or inserting individual nucleotides. A *gap* in an alignment is defined as a contiguous sequence of spaces in one of the rows. Since insertions and deletions of substrings are common evolutionary events, penalizing a gap of length x as $-\sigma x$ is cruel and unusual punishment. Many practical alignment algorithms use a softer approach to gap penalties and penalize a gap of x spaces by a function that grows slower than the sum of penalties for x indels.

To this end, we define *affine gap penalties* to be a linearly weighted score for large gaps. We can set the score for a gap of length x to be $-(\rho + \sigma x)$, where $\rho > 0$ is the penalty for the introduction of the gap and $\sigma > 0$ is the penalty for each symbol in the gap (ρ is typically large while σ is typically small). Though this may seem to be complicating our alignment approach, it turns out that the edit graph representation of the problem is robust enough to accommodate it.

Affine gap penalties can be accommodated by adding "long" vertical and horizontal edges in the edit graph (e.g., an edge from (i, j) to $(i + x, j)$ of length $-(\rho + \sigma x)$ and an edge from (i, j) to $(i, j + x)$ of the same length) from each vertex to every other vertex that is either east or south of it. We can then apply the same algorithm as before to compute the longest path in this graph. Since the number of edges in the edit graph for affine gap penalties increases, at first glance it looks as though the running time for the alignment algorithm also increases from $O(n^2)$ to $O(n^3)$, where n is the longer of the two string lengths.[11] However, the following three recurrences keep the running time down:

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho + \sigma) \end{cases}$$

$$\vec{s}_{i,j} = \max \begin{cases} \vec{s}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

11. The complexity of the corresponding Longest Path in a DAG problem is defined by the number of edges in the graph. Adding long horizontal and vertical edges imposed by affine gap penalties increases the number of edges by a factor of n.

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \overset{\downarrow}{s}_{i,j} \\ \vec{s}_{i,j} \end{cases}$$

The variable $\overset{\downarrow}{s}_{i,j}$ computes the score for alignment between the i-prefix of $\mathbf{v}$ and the j-prefix of $\mathbf{w}$ ending with a deletion (i.e., a gap in $\mathbf{w}$), while the variable $\vec{s}_{i,j}$ computes the score for alignment ending with an insertion (i.e., a gap in $\mathbf{v}$). The first term in the recurrences for $\overset{\downarrow}{s}_{i,j}$ and $\vec{s}_{i,j}$ corresponds to extending the gap, while the second term corresponds to initiating the gap. Essentially, $\overset{\downarrow}{s}_{i,j}$ and $\vec{s}_{i,j}$ are the scores of optimal paths that arrive at vertex (i, j) via vertical and horizontal edges correspondingly.

Figure 6.18 further explains how alignment with affine gap penalties can be reduced to the Manhattan Tourist problem in the appropriate city grid. In this case the city is built on three levels: the bottom level built solely with vertical $\downarrow$ edges with weight $-\sigma$; the middle level built with diagonal edges of weight $\delta(v_i, w_j)$; and the upper level, which is built from horizontal edges $\rightarrow$ with weight $-\sigma$. The lower level corresponds to gaps in sequence $\mathbf{w}$, the middle level corresponds to matches and mismatches, and the upper level corresponds to gaps in sequence $\mathbf{v}$. Also, in this graph there are two edges from each vertex $(i, j)_{middle}$ at the middle level that connect this vertex with vertex $(i + 1, j)_{lower}$ at the lower level and with vertex $(i, j + 1)_{upper}$ at the upper level. These edges model a start of the gap and have weight $-(\rho + \sigma)$. Finally, one has to introduce zero-weight edges connecting vertices $(i, j)_{lower}$ and $(i, j)_{upper}$ with vertex $(i, j)_{middle}$ at the middle level (these edges model the end of the gap). In effect, we have created a rather complicated graph, but the same algorithm works with it.

We have now introduced a number of pairwise sequence comparison problems and shown that they can all be solved by what is essentially the same dynamic programming algorithm applied to a suitably built Manhattan-style city. We will now consider other applications of dynamic programming in bioinformatics.

6.10 Multiple Alignment

The goal of protein sequence comparison is to discover structural or functional similarities among proteins. Biologically similar proteins may not exhibit a strong sequence similarity, but we would still like to recognize resem-

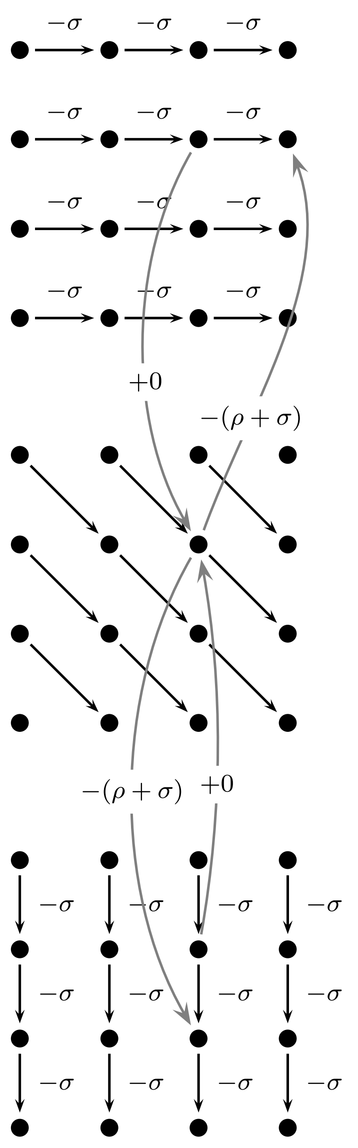
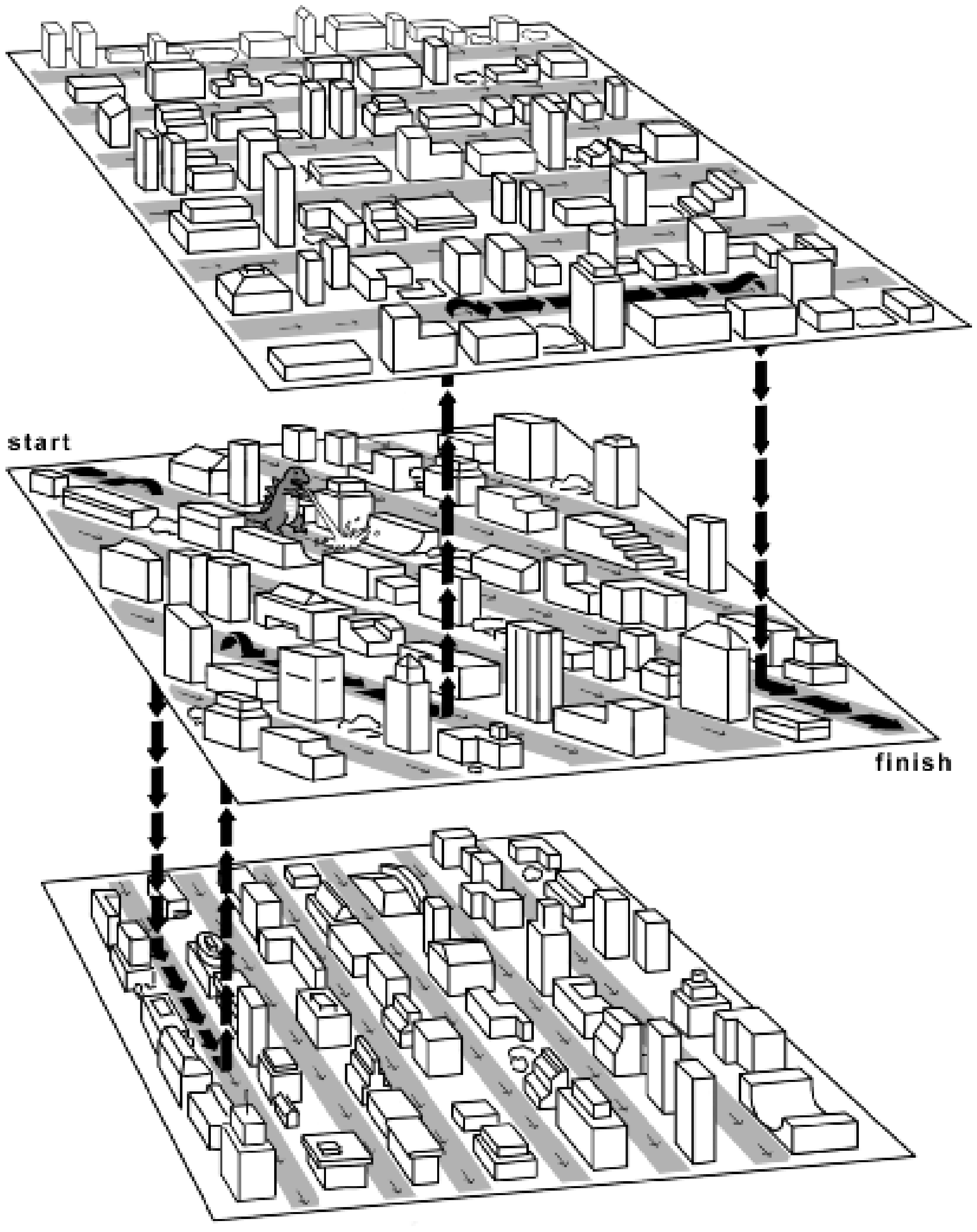


Figure 6.18 A three-level edit graph for alignment with affine gap penalties. Every vertex (i, j) in the middle level has one outgoing edge to the upper level, one outgoing edge to the lower level, and one incoming edge each from the upper and lower levels.


start
finish

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
 ||||| |   X|||| |          || XXX|||  | ||| |  |
-ATTGC-G--ATTCGTAT------GGGACA-TGGATGCATGCAG-TGAC
```

Figure 6.19 Multiple alignment of three sequences.

blance even when the sequences share only weak similarities.[12] If sequence similarity is weak, pairwise alignment can fail to identify biologically related sequences because weak pairwise similarities may fail statistical tests for significance. However, simultaneous comparison of many sequences often allows one to find similarities that are invisible in pairwise sequence comparison.

Let $\mathbf{v}_1, \ldots, \mathbf{v}_k$ be k strings of length $n_1, \ldots, n_k$ over an alphabet $\mathcal{A}$. Let $\mathcal{A}'$ denote the extended alphabet $\mathcal{A} \bigcup \{-\}$, where '$-$' denotes the space character (reserved for insertions and deletions). A *multiple alignment* of strings $\mathbf{v_1}, \ldots, \mathbf{v_k}$ is specified by a $k \times n$ matrix A, where $n \geq \max_{1 \leq i \leq k} n_i$. Each element of the matrix is a member of $\mathcal{A}'$, and each row i contains the characters of $\mathbf{v_i}$ in order, interspersed with $n - n_i$ spaces (figure 6.19). We also assume that every column of the multiple alignment matrix contains at least one symbol from $\mathcal{A}$, that is, no column in a multiple alignment contains only spaces. The multiple alignment matrix we have constructed is a generalization of the pairwise alignment matrix to $k > 2$ sequences. The score of a multiple alignment is defined to be the sum of scores of the columns, with the optimal alignment being the one that maximizes the score. Just as it was in section 4.5, the consensus of an alignment is a string of the most common characters in each column of the multiple alignment. At this point, we will use a very general scoring function that is defined by a k-dimensional matrix δ of size $|\mathcal{A}'| \times \ldots \times |\mathcal{A}'|$ that describes the scores of all possible combinations of k symbols from $\mathcal{A}'$.[13]

A straightforward dynamic programming algorithm in the k-dimensional edit graph formed from k strings solves the Multiple Alignment problem.

12. Sequences that code for proteins that perform the same function are likely to be somehow related but it may be difficult to decide whether this similarity is significant or happens just by chance.

13. This is a k-dimensional scoring matrix rather than the two-dimensional $|\mathcal{A}'| \times |\mathcal{A}'|$ matrix for pairwise alignment (which is a multiple alignment with $k = 2$).

For example, suppose that we have three sequences $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, and that we want to find the "best" alignment of all three. Every multiple alignment of three sequences corresponds to a path in the three-dimensional Manhattan-like edit graph. In this case, one can apply the same logic as we did for two dimensions to arrive at a dynamic programming recurrence, this time with more terms to consider. To get to vertex (i, j, k) in a three-dimensional edit graph, you could come from any of the following predecessors (note that $\delta(x, y, z)$ denotes the score of a column with letters x, y, and z, as in figure 6.20):

1. $(i-1, j, k)$ for score $\delta(u_i, -, -)$
2. $(i, j-1, k)$ for score $\delta(-, v_j, -)$
3. $(i, j, k-1)$ for score $\delta(-, -, w_k)$
4. $(i-1, j-1, k)$ for score $\delta(u_i, v_j, -)$
5. $(i-1, j, k-1)$ for score $\delta(u_i, -, w_k)$
6. $(i, j-1, k-1)$ for score $\delta(-, v_j, w_k)$
7. $(i-1, j-1, k-1)$ for score $\delta(u_i, v_j, w_k)$

We create a three-dimensional dynamic programming array s and it is easy to see that the recurrence for $s_{i,j,k}$ in the three-dimensional case is similar to the recurrence in the two-dimensional case (fig. 6.21). Namely,

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j,k} & +\delta(v_i, -, -) \\ s_{i,j-1,k} & +\delta(-, w_j, -) \\ s_{i,j,k-1} & +\delta(-, -, u_k) \\ s_{i-1,j-1,k} & +\delta(v_i, w_j, -) \\ s_{i-1,j,k-1} & +\delta(v_i, -, u_k) \\ s_{i,j-1,k-1} & +\delta(-, w_j, u_k) \\ s_{i-1,j-1,k-1} & +\delta(v_i, w_j, u_k) \end{cases}$$

Unfortunately, in the case of k sequences, the running time of this approach is $O((2n)^k)$, so some improvements of the exact algorithm, and many heuristics for suboptimal multiple alignments, have been proposed. A good heuristic would be to compute all $\binom{k}{2}$ optimal pairwise alignments between every pair of strings and then combine them together in such a way that pairwise alignments induced by the multiple alignment are close to the optimal

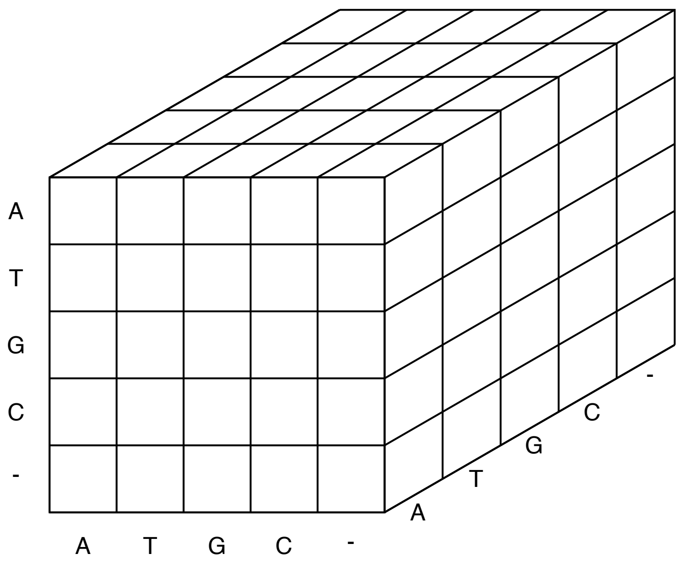


Figure 6.20 The scoring matrix, δ, used in a three-sequence alignment.

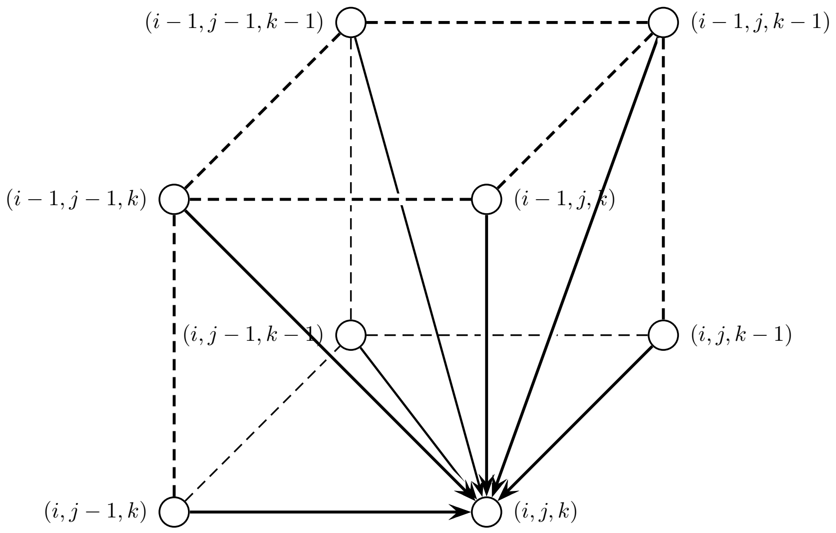


Figure 6.21 A cell in the alignment graph between three sequences.

ones. Unfortunately, it is not always possible to combine optimal pairwise alignments into a multiple alignment since some pairwise alignments may be incompatible. For example, figure 6.22 (a) shows three sequences whose optimal pairwise alignment can be combined into a multiple alignment, whereas (b) shows three sequences that cannot be combined. As a result, some multiple alignment algorithms attempt to combine some compatible subset of optimal pairwise alignments into a multiple alignment.

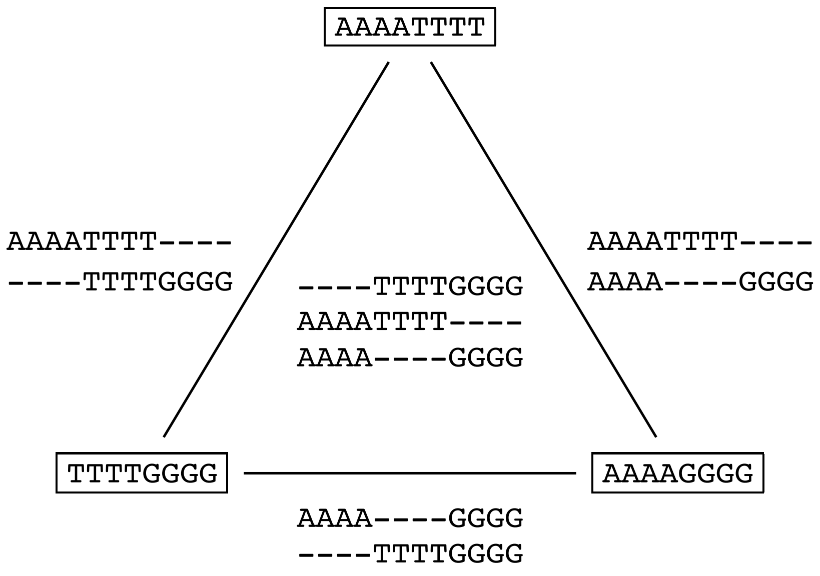
Another approach to do this uses one particularly strong pairwise alignment as a building block for the multiple k-way alignment, and iteratively adds one string to the growing multiple alignment. This greedy *progressive multiple alignment* heuristic selects the pair of strings with greatest similarity and merges them together into a new string following the principle "once a gap, always a gap."[14] As a result, the multiple alignment of k sequences is reduced to the multiple alignment of $k-1$ sequences. The motivation for the choice of the closest strings at the early steps of the algorithm is that close strings often provide the most reliable information about a real alignment. Many popular iterative multiple alignment algorithms, including the tool CLUSTAL, use similar strategies.

Although progressive multiple alignment algorithms work well for very close sequences, there are no performance guarantees for this approach. The problem with progressive multiple alignment algorithms like CLUSTAL is that they may be misled by some spuriously strong pairwise alignment, in effect, a bad seed. If the very first two sequences picked for building multiple alignment are aligned in a way that is incompatible with the optimal multiple alignment, the error in this initial pairwise alignment will propagate all the way through to the whole multiple alignment. Many multiple alignment algorithms have been proposed, and even with systematic deficiencies such as the above they remain quite useful in computational biology.

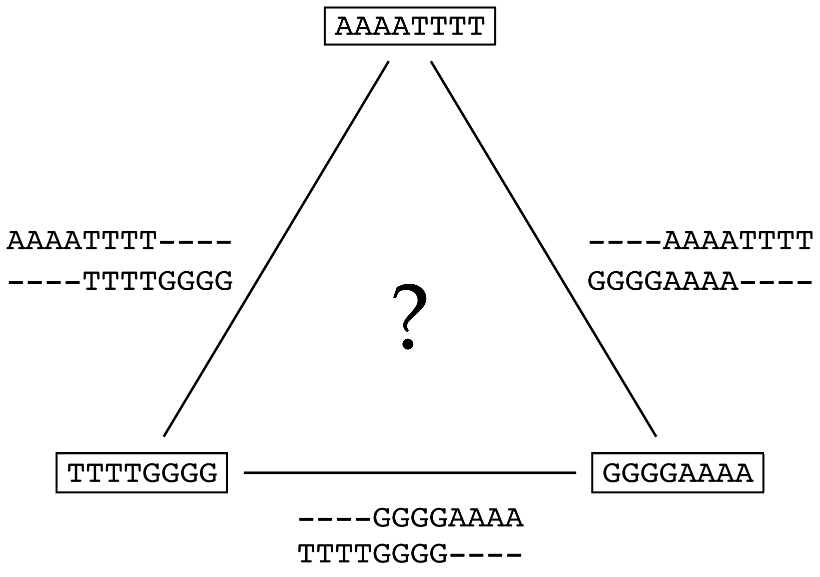
We have described multiple alignment for k sequences as a generalization of the Pairwise Alignment problem, which assumed the existence of a k-dimensional scoring matrix δ. Since such k-dimensional scoring matrices are not very practical, we briefly describe two other scoring approaches that are more biologically relevant. The choice of the scoring function can drastically affect the quality of the resulting alignment, and no single scoring approach is perfect in all circumstances.

The columns of a multiple alignment of k sequences describe a path of

14. Essentially, this principle states that once a gap has been introduced into the alignment it will never close, even if that would lead to a better overall score.



(a) Compatible pairwise alignments



(b) Incompatible pairwise alignments

Figure 6.22 Given three sequences, it might be possible to combine their pairwise alignment into a multiple alignment (a), but it might not be (b).

edges in a k-dimensional version of the Manhattan gridlike edit graph. The weights of these edges are determined by the scoring function δ. Intuitively, we want to assign higher scores to the columns with a low variation in letters, such that high scores correspond to highly conserved sequences. For example, in the *Multiple Longest Common Subsequence* problem, the score of a column is set to 1 if all the characters in the column are the same, and 0 if even one character disagrees.

In the more statistically motivated *entropy* approach, the score of a multiple alignment is defined as the sum of the entropies of the columns, which are defined to be[15]

$$\sum_{x \in \mathcal{A}'} p_x \log p_x$$

where p_x is the frequency of letter $x \in \mathcal{A}'$ in a given column. In this case, the more conserved the column, the larger the entropy score. For example, a column that has each of the 4 nucleotides present $\frac{k}{4}$ times will have an entropy score of $4\frac{1}{4}\log\frac{1}{4} = -2$, while a completely conserved column (as in the multiple LCS problem) would have entropy 0. Finding the longest path in the k-dimensional edit graph corresponds to finding the multiple alignment with the largest entropy score.

While entropy captures some statistical notion of a good alignment, it can be hard to design efficient algorithms that optimize this scoring function. Another popular scoring approach is the *Sum-of-Pairs score (SP-score)*. Any multiple alignment A of k sequences $\mathbf{v_1}, \ldots, \mathbf{v_k}$ forces a pairwise alignment between any two sequences $\mathbf{v_i}$ and $\mathbf{v_j}$ of score $s_A(\mathbf{v_i}, \mathbf{v_j})$.[16] The SP-score for a multiple alignment A is given by $\sum_{1 \le i < j \le k} s_A(\mathbf{v_i}, \mathbf{v_j})$. In this definition, the score of an alignment A is built from the scores of all pairs of strings in the alignment.

6.11 Gene Prediction

In 1961 Sydney Brenner and Francis Crick demonstrated that every triplet of nucleotides (codon) in a gene codes for one amino acid in the corresponding protein. They were able to introduce deletions in DNA and observed that deletion of a single nucleotide or two consecutive nucleotides in a gene dramatically alters its protein product. Paradoxically, deleting three consecutive

15. The correct way to define entropy is to take the negative of this expression, but the definition above allows us to deal with a maximization rather than a minimization problem.
16. We remark that the resulting "forced" alignment is not necessarily optimal.

nucleotides results in minor changes in the protein. For example, the phrase `THE SLY FOX AND THE SHY DOG` (written in triplets) turns into gibberish after deleting one letter (`THE SYF OXA NDT HES HYD OG`) or two letters (`THE SFO XAN DTH ESH YDO G`), but makes some sense after deleting three nucleotides `THE SOX AND THE SHY DOG`. Inspired by this experiment Charles Yanofsky proved that a gene and its protein product are collinear, that is, the first codon in the gene codes for the first amino acid in the protein, the second codon codes for the second amino acid (rather than, say, the seventeenth), and so on. Yanofsky's ingenious experiment was so influential that nobody even questioned whether codons are represented by continuous stretches in DNA, and for the subsequent fifteen years biologists believed that a protein was encoded by a long string of contiguous triplets. However, the discovery of split human genes in 1977 proved that genes are often represented by a *collection* of substrings, and raised the computational problem of predicting the locations of genes in a genome given only the genomic DNA sequence.

The human genome is larger and more complex than bacterial genomes. This is not particularly surprising since one would expect to find more genes in humans than in bacteria. However, the genome size of many eukaryotes does not appear to be related to an organism's genetic complexity; for example, the salamander genome is ten times larger than the human genome. This apparent paradox was resolved by the discovery that many organisms contain not only genes but also large amounts of so-called *junk DNA* that does not code for proteins at all. In particular, most human genes are broken into pieces called *exons* that are separated by this junk DNA. The difference in the sizes of the salamander and human genomes thus presumably reflects larger amounts of junk DNA and repeats in the salamander genome.

Split genes are analogous to a magazine article that begins on page 1, continues on page 13, then takes up again on pages 43, 51, 74, 80, and 91, with pages of advertising appearing in between. We do not understand why these jumps occur. and a significant portion of the human genome is this junk "advertising" that separates exons.

More confusing is that the jumps between different parts of split genes are inconsistent from species to species. A gene in an insect edition of the genome will be organized differently than the related gene in a worm genome. The number of parts (exons) may be different: the information that appears in one part in the human edition may be broken up into two in the mouse version, or vice versa. While the genes themselves are related, they may be quite different in terms of the parts' structure.

Split genes were first discovered in 1977 in the laboratories of Phillip Sharp and Richard Roberts during studies of the adenovirus. The discovery was such a surprise that the paper by Roberts's group had an unusually catchy title for the journal *Cell*: "An Amazing Sequence Arrangement at the 5′ End of Adenovirus 2 Messenger RNA." Sharp's group focused their experiments on an mRNA[17] that encodes a viral protein known as *hexon*. To map the hexon mRNA in the viral genome, mRNA was hybridized to adenovirus DNA and the hybrid molecules were analyzed by electron microscopy. Strikingly, the mRNA-DNA hybrids formed in this experiment displayed three loop structures, rather than the continuous duplex segment suggested by the classic continuous gene model (figure 6.23). Further hybridization experiments revealed that the hexon mRNA is built from four separate fragments of the adenovirus genome. These four continuous segments (called *exons*) in the adenovirus genome are separated by three "junk" fragments called *introns*.

Gene prediction is the problem of locating genes in a genomic sequence. Human genes constitute only 3% of the human genome, and no existing in silico gene recognition algorithm provides completely reliable gene recognition. The intron-exon model of a gene seems to prevail in eukaryotic organisms; prokaryotic organisms (like bacteria) do not have broken genes. As a result, gene prediction algorithms for prokaryotes tend to be somewhat simpler than those for eukaryotes.[18]

There are roughly two categories of approaches that researchers have used for predicting gene location. The statistical approach to gene prediction is to look for features that appear frequently in genes and infrequently elsewhere. Many researchers have attempted to recognize the locations of *splicing signals* at exon-intron junctions.[19] For example, the dinucleotides AG and GT on the left- and right-hand sides of an exon are highly conserved (figure 6.24). In addition, there are other less conserved positions on both sides of the exons. The simplest way to represent such binding sites is by a profile describing the propensities of different nucleotides to occur at different positions. Unfortu-

17. At that time, messenger RNA (mRNA) was viewed as a copy of a gene translated into the RNA alphabet. It is used to transfer information from the nuclear genome to the ribosomes to direct protein synthesis.
18. This is not to say that bacterial gene prediction is a trivial task but rather to indicate that eukaryotic gene finding is very difficult.
19. If genes are separated into exons interspersed with introns, then the RNA that is transcribed from DNA (i.e., the complementary copy of a gene) should be longer than the mRNA that is used as a template for protein synthesis. Therefore, some biological process needs to remove the introns in the pre-mRNA and concatenate the exons into a single mRNA string. This process is known as *splicing*, and the resulting mRNA is used as a template for protein synthesis in cytoplasm.

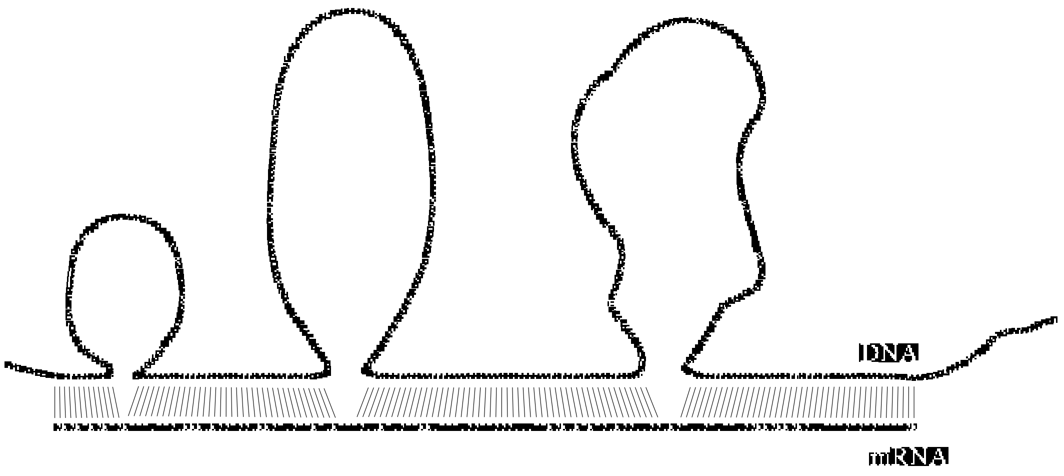


Figure 6.23 An electron microscopy experiment led to the discovery of split genes. When mRNA (below) is hybridized against the DNA that generated it, three distinct loops can be seen (above). Because the loops are present in the DNA and are not present in mRNA, certain parts (introns) must be removed during the process of mRNA formation called splicing.

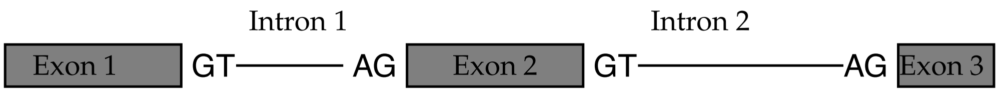


Figure 6.24 Exons typically are flanked by the dinucleotides AG and GT.

nately, using profiles to detect splice sites has met with limited success since these profiles are quite weak and tend to match frequently in the genome at nonsplice sites. Attempts to improve the accuracy of gene prediction led to the second category of approaches for gene finding: those based on similarity.

The similarity-based approach to gene prediction relies on the observation that a newly sequenced gene has a good chance of being related to one that is already known. For example, 99% of mouse genes have human analogs. However, one cannot simply look for a similar sequence in one organism's genome based on the genes known in another, for the reasons outlined above: both the exon sequence and the exon structure of the related gene in different species are different. The commonality between the related genes in both organisms is that they produce similar proteins. Accordingly, instead of employing a statistical analysis of exons, similarity-based methods attempt to solve a combinatorial puzzle: find a set of substrings (putative exons) in a genomic sequence (say, mouse) whose concatenation fits a known human protein. In this scenario, we suppose we know a human protein, and we want to discover the exon structure of the related gene in the mouse genome. The more sequence data we collect, the more accurate and reliable similarity-based methods become. Consequently, the trend in gene prediction has recently shifted from statistically motivated approaches to similarity-based algorithms.

6.12 Statistical Approaches to Gene Prediction

As mentioned above, statistical approaches to finding genes rely on detecting subtle statistical variations between coding (exons) and non-coding regions. The simplest way to detect potential coding regions is to look at *open reading frames*, or *ORFs*. One can represent a genome of length n as a sequence of $\frac{n}{3}$ codons.[20] The three "stop" codons, (TAA, TAG, and TGA) break this sequence into segments, one between every two consecutive stop codons. The subsegments of these that start from a start codon, ATG, are ORFs. ORFs within a single genomic sequence may overlap since there are six possible "reading frames": three on one strand starting at positions 1, 2, and 3, and three on the reverse strand, as shown in figure 6.25.

One would expect to find frequent stop codons in noncoding DNA, since the average number of codons between two consecutive stop codons in "random" DNA should be $\frac{64}{3} \approx 21$.[21] This is much smaller than the number of codons in an average protein, which is roughly 300. Therefore, ORFs longer than some threshold length indicate potential genes. However, gene prediction algorithms based on selecting significantly long ORFs may fail to detect short genes or genes with short exons.

20. In fact, there are three such representations for each DNA strand: one starting at position 1, another at 2 (ignoring the first base), and the third one at 3 (ignoring the first two bases).
21. There are $4^3 = 64$ codons, and three of them are Stop codons.

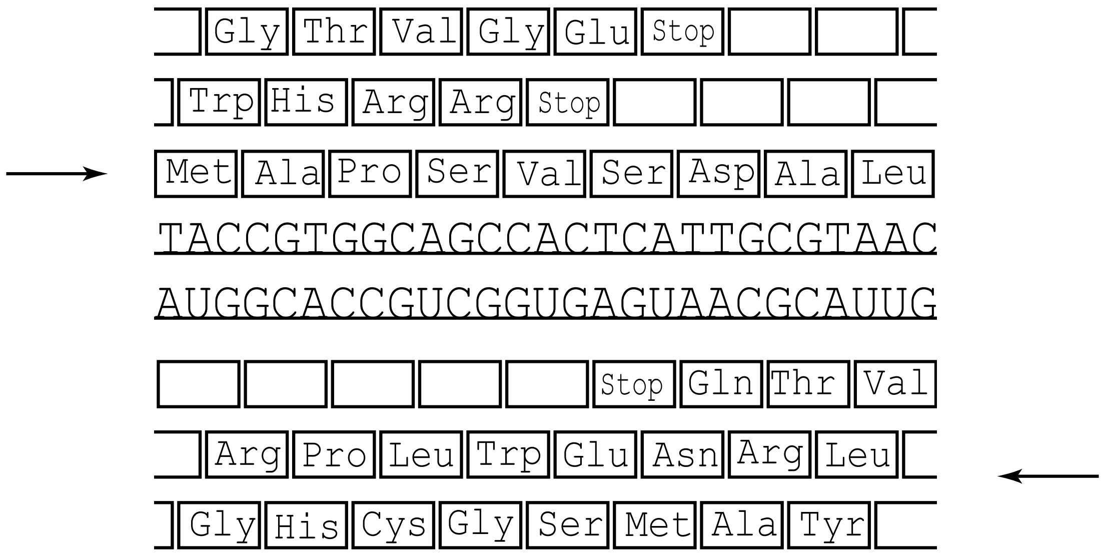

Gly Thr Val Gly Glu Stop
Trp His Arg Arg Stop
Met Ala Pro Ser Val Ser Asp Ala Leu
TACCGTGGCAGCCACTCATTGCGTAAC
AUGGCACCGUCGGUGAGUAACGCAUUG
Stop Gln Thr Val
Arg Pro Leu Trp Glu Asn Arg Leu
Gly His Cys Gly Ser Met Ala Tyr

Figure 6.25 The six reading frames for the sequence ATGCTTAGTCTG. The string may be read forward or backward, and there are three frame shifts in each direction.

Many statistical gene prediction algorithms rely on statistical features in protein-coding regions, such as biases in *codon usage*. We can enter the frequency of occurrence of each codon within a given sequence into a 64-element *codon usage array*, as in table 6.1. The codon usage arrays for coding regions are different than the codon usage arrays for non-coding regions, enabling one to use them for gene prediction. For example, in human genes codons CGC and AGG code for the same amino acid (Arg) but have very different frequencies: CGC is 12 times more likely to be used in genes than AGG (table 6.1). Therefore, an ORF that "prefers" CGC over AGG while coding for Arg is a likely candidate gene. One can use a likelihood ratio approach[22] to compute the conditional probabilities of the DNA sequence in a window, under the hypothesis that the window contains a coding sequence, and under the hypothesis that the window contains a noncoding sequence. If we slide this window along the genomic DNA sequence (and calculate the likelihood

22. The *likelihood ratio* technique allows one to test the applicability of two distinct hypotheses; when the likelihood ratio is large, the first hypothesis is more likely to be true than the second one.

Table 6.1 The genetic code and codon usage in *Homo sapiens*. The codon for methionine, or AUG, also acts as a start codon; all proteins begin with Met. The numbers next to each codon reflects the frequency of that codon's occurrence while coding for an amino acid. For example, among all lysine (Lys) residues in all the proteins in a genome, the codon AAG generates 25% of them while the codon AAG generates 75%. These frequencies differ across species.

	U	C	A	G
U	UUU `Phe` 57	UCU `Ser` 16	UAU `Tyr` 58	UGU `Cys` 45
	UUC `Phe` 43	UCC `Ser` 15	UAC `Tyr` 42	UGC `Cys` 55
	UUA `Leu` 13	UCA `Ser` 13	UAA `Stp` 62	UGA `Stp` 30
	UUG `Leu` 13	UCG `Ser` 15	UAG `Stp` 8	UGG `Trp` 100
C	CUU `Leu` 11	CCU `Pro` 17	CAU `His` 57	CGU `Arg` 37
	CUC `Leu` 10	CCC `Pro` 17	CAC `His` 43	CGC `Arg` 38
	CUA `Leu` 4	CCA `Pro` 20	CAA `Gln` 45	CGA `Arg` 7
	CUG `Leu` 49	CCG `Pro` 51	CAG `Gln` 66	CGG `Arg` 10
A	AUU `Ile` 50	ACU `Thr` 18	AAU `Asn` 46	AGU `Ser` 15
	AUC `Ile` 41	ACC `Thr` 42	AAC `Asn` 54	AGC `Ser` 26
	AUA `Ile` 9	ACA `Thr` 15	AAA `Lys` 75	AGA `Arg` 5
	AUG `Met` 100	ACG `Thr` 26	AAG `Lys` 25	AGG `Arg` 3
G	GUU `Val` 27	GCU `Ala` 17	GAU `Asp` 63	GGU `Gly` 34
	GUC `Val` 21	GCC `Ala` 27	GAC `Asp` 37	GGC `Gly` 39
	GUA `Val` 16	GCA `Ala` 22	GAA `Glu` 68	GGA `Gly` 12
	GUG `Val` 36	GCG `Ala` 34	GAG `Glu` 32	GGG `Gly` 15

ratio at each point), genes are often revealed as peaks in the likelihood ratio plots.

An even better coding sensor is the *in-frame hexamer count*[23] proposed by Mark Borodovsky and colleagues. Gene prediction in bacterial genomes also takes advantage of several conserved sequence motifs often found in the regions around the start of transcription. Unfortunately, such sequence motifs are more elusive in eukaryotes.

While the described approaches are successful in prokaryotes, their application to eukaryotes is complicated by the exon-intron structure. The average length of exons in vertebrates is 130 nucleotides, and exons of this length are too short to produce reliable peaks in the likelihood ratio plot while analyzing ORFs because they do not differ enough from random fluctuations to be detectable. Moreover, codon usage and other statistical parameters proba-

23. The in-frame hexamer count reflects frequencies of pairs of consecutive codons.

bly have nothing in common with the way the splicing machinery actually recognizes exons. Many researchers have used a more biologically oriented approach and have attempted to recognize the locations of splicing signals at exon-intron junctions. There exists a (weakly) conserved sequence of eight nucleotides at the boundary of an exon and an intron (*donor* splice site) and a sequence of four nucleotides at the boundary of an intron and exon (*acceptor* splice site). Since profiles for splice sites are weak, these approaches have had limited success and have been supplanted by hidden Markov model (HMM) approaches[24] that capture statistical dependencies between sites. A popular example of this latter approach is GENSCAN, which was developed in 1997 by Chris Burge and Samuel Karlin. GENSCAN combines coding region and splicing signal predictions into a single framework. For example, a splice site prediction is more believable if signs of a coding region appear on one side of the site but not on the other. Many such statistics are used in the HMM framework of GENSCAN that merges splicing site statistics, coding region statistics, and motifs near the start of the gene, among others. However, the accuracy of GENSCAN decreases for genes with many short exons or with unusual codon usage.

6.13 Similarity-Based Approaches to Gene Prediction

A similarity-based approach to gene prediction uses previously sequenced genes and their protein products as a template for the recognition of unknown genes in newly sequenced DNA fragments. Instead of employing statistical properties of exons, this method attempts to solve the following combinatorial puzzle: given a known target protein and a genomic sequence, find a set of substrings (candidate exons) of the genomic sequence whose concatenation (splicing) best fits the target.

A naive brute force approach to the spliced alignment problem is to find all local similarities between the genomic sequence and the target protein sequence. Each substring from the genomic sequence that exhibits sufficient similarity to the target protein could be considered a *putative exon*.[25] The putative exons so chosen may lack the canonical exon-flanking dinucleotides AG and GT but we can extend or shorten them slightly to make sure that they are flanked by AG and GT. The resulting set may contain overlapping

24. Hidden Markov models are described in chapter 11.
25. Putative here means that the sequence *might* be an exon, even though we have no proof of this.

substrings, and the problem is to choose the best subset of nonoverlapping substrings as a putative exon structure.[26]

We will model a putative exon with a *weighted interval* in the genomic sequence, which is described by three parameters (l, r, w), as in figure 6.26. Here, l is the left-hand position, r is the right-hand position, and w is the weight of the putative exon. The weight w may reflect the local alignment score for the genomic interval against the target protein sequence, or the strength of flanking acceptor and donor sites, or any combination of these and other measures; it reflects the likelihood that this interval is an exon. A *chain* is any set of nonoverlapping weighted intervals. The total weight of a chain is the sum of the weights of the intervals in the chain. A *maximum chain* is a chain with maximum total weight among all possible chains. Below we assume that the weights of all intervals are positive $(w > 0)$.

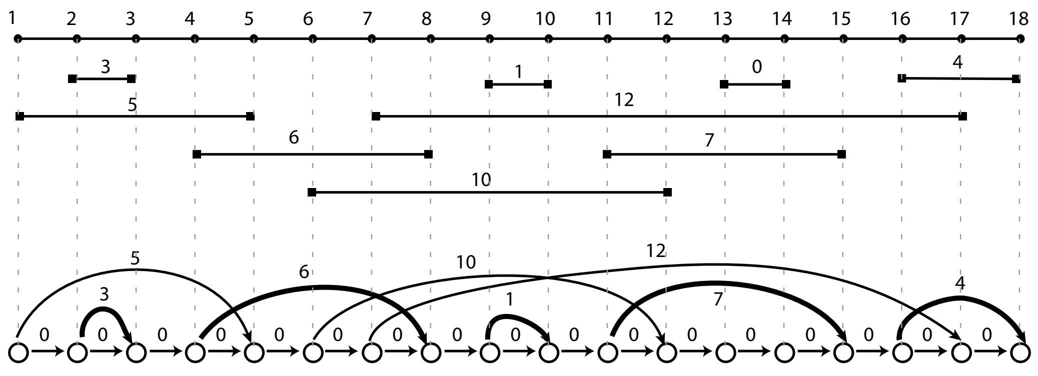
Exon Chaining Problem:
Given a set of putative exons, find a maximum set of nonoverlapping putative exons.

Input: A set of weighted intervals (putative exons).

Output: A maximum chain of intervals from this set.

The Exon Chaining problem for n intervals can be solved by dynamic programming in a graph G on $2n$ vertices, n of which represent starting (left) positions of intervals and n of which represent ending (right) positions of intervals, as in figure 6.26. We assume that the set of left and right interval ends is sorted into increasing order and that all positions are distinct, forming an ordered array of vertices $(v_1, \dots v_{2n})$ in graph G.[27] There are $3n - 1$ edges in this graph: there is an edge between each l_i and r_i of weight w_i for i from 1 to n, and $2n - 1$ additional edges of weight 0 which simply connect adjacent vertices (v_i, v_{i+1}) forming a path in the graph from v_1 to v_{2n}. In the algorithm below, s_i represents the length of the longest path in the graph ending at vertex v_i. Thus, s_{2n} is the solution to the Exon Chaining problem.

26. We choose nonoverlapping substrings because exons in real genes do not overlap.
27. In particular, we are assuming that no interval starts exactly where another ends.



0	0	3	3	5	5	5	9	9	10	10	15	15	15	17	17	17	20

Figure 6.26 A short "genomic" sequence, a set of nine weighted intervals, and the graph used for the dynamic programming solution to the Exon Chaining problem. Five weighted intervals, $(2, 3, 3)$, $(4, 8, 6)$, $(9, 10, 1)$, $(11, 15, 7)$, and $(16, 18, 4)$, shown by bold edges, form an optimal solution to the Exon Chaining problem. The array at the bottom shows the values $s_1, s_2, \ldots, s_{2n}$ generated by the EXONCHANING algorithm.

```
EXONCHAINING($G, n$)
1  for $i \leftarrow 1$ to $2n$
2      $s_i \leftarrow 0$
3  for $i \leftarrow 1$ to $2n$
4      if vertex $v_i$ in $G$ corresponds to the right end of an interval $I$
5          $j \leftarrow$ index of vertex for left end of the interval $I$
6          $w \leftarrow$ weight of the interval $I$
7          $s_i \leftarrow \max\{s_j + w, s_{i-1}\}$
8      else
9          $s_i \leftarrow s_{i-1}$
10 return $s_{2n}$
```

One shortcoming of this approach is that the endpoints of putative exons are not very well defined, and this assembly method does not allow for any flexibility at these points. More importantly, the optimal chain of intervals may not correspond to any valid alignment. For example, the first interval in the optimal chain may be similar to a suffix of the protein, while the second interval in the optimal chain may be similar to a prefix. In this case, the putative exons corresponding to the valid chain of these two intervals cannot

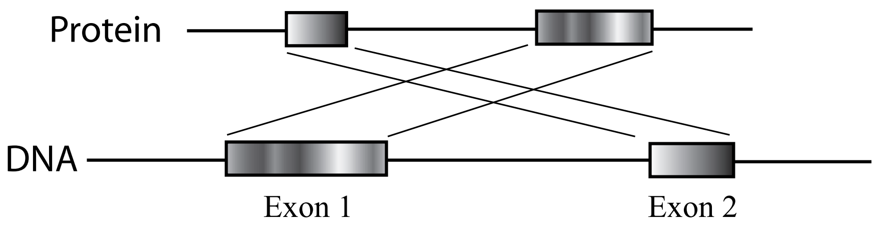


Figure 6.27 An infeasible chain that might have a maximal score. The first exon corresponds to a region at the end of the target protein, while the second exon corresponds to a region at the beginning of the target protein. These exons cannot be combined into a valid global DNA-protein alignment.

be combined into a valid alignment (figure 6.27).

6.14 Spliced Alignment

In 1996, Mikhail Gelfand and colleagues proposed the *spliced alignment* approach to find genes in eukaryotes: use a related protein within one genome to reconstruct the exon-intron structure of a gene in another genome. The spliced alignment begins by selecting either all putative exons between potential acceptor and donor sites (e.g., between AG and GT dinucleotides), or by finding all substrings similar to the target protein, as in the Exon Chaining problem. By filtering this set in a way that attempts not to lose true exons, one is left with a set of candidate exons that may contains many false exons, but definitely contains all the true ones. While it is difficult to distinguish the good (true exons) from the bad (false exons) by a statistical procedure alone, we can use the alignment with the target protein to aid us in our search. In theory, only the true exons will form a coherent representation of a protein.

Given the set of candidate exons and a target protein sequence, we explore all possible chains (assemblies) of the candidate exon set to find the assembly with the highest similarity score to the target protein. The number of different assemblies may be huge, but the spliced alignment algorithm is able to find the best assembly among all of them in polynomial time. For simplicity we will assume that the protein sequence is expressed in the same alphabet as the geneome. Of course, this is not the case in nature, and a problem at the end of this chapter asks you to modify the recurrence relations accordingly.

Let $G = g_1 \ldots g_n$ be the genomic sequence, $T = t_1 \ldots t_m$ be the target sequence, and $\mathcal{B}$ be the set of candidate exons (blocks). As above, a chain Γ is any sequence of nonoverlapping blocks, and the string formed by a chain is just the concatenation of all the blocks in the chain. We will use Γ^* to denote the string formed by the chain Γ. The chain that we are searching for is the one whose concatenation forms the string with the highest similarity to the target sequence.[28]

Spliced Alignment Problem:
Find a chain of candidate exons in a genomic sequence that best fits a target sequence.

Input: Genomic sequence G, target sequence T, and a set of candidate exons (blocks) $\mathcal{B}$.

Output: A chain of candidate exons Γ such that the global alignment score $s(\Gamma^*, T)$ is maximum among all chains of candidate exons from $\mathcal{B}$.

As an example, consider the "genomic" sequence "It was brilliant thrilling morning and the slimy, hellish, lithe doves gyrated and gambled nimbly in the waves" with the set of blocks shown in figure 6.28 (top) by overlapping rectangles. If our target is the famous Lewis Carroll line "'twas brillig, and the slithy toves did gyre and gimble in the wabe" then figure 6.28 illustrates the spliced alignment problem of choosing the best "exons" (or blocks, in this case) that can be assembled into the target.

The spliced alignment problem can be cast as finding a path in a directed acyclic graph [fig. 6.28 (middle)]. Vertices in this graph (shown as rectangles) correspond to blocks (candidate exons), and directed edges connect nonoverlapping blocks. A vertex corresponding to a block B is labeled by a string represented by this block. Therefore, every path in the spliced alignment graph spells out the string obtained by concatenation of labels of its vertices. The weight of a path in this graph is defined as the score of the optimal alignment between the concatenated blocks of this path and the target sequence. Note that we have defined the weight of an entire path in the graph, but we have not defined weights for individual edges. This makes the Spliced Alignment problem different from the standard Longest Path problem. Nevertheless, we can leverage dynamic programming to solve the problem.

28. We emphasize the difference between the scoring functions for the Exon Chaining problem and the Spliced Alignment problem. In contrast to the Spliced Alignment problem, the set of nonoverlapping substrings representing the solution of the Exon Chaining problem does not necessarily correspond to a valid alignment between the genomic sequence and the target protein sequence.

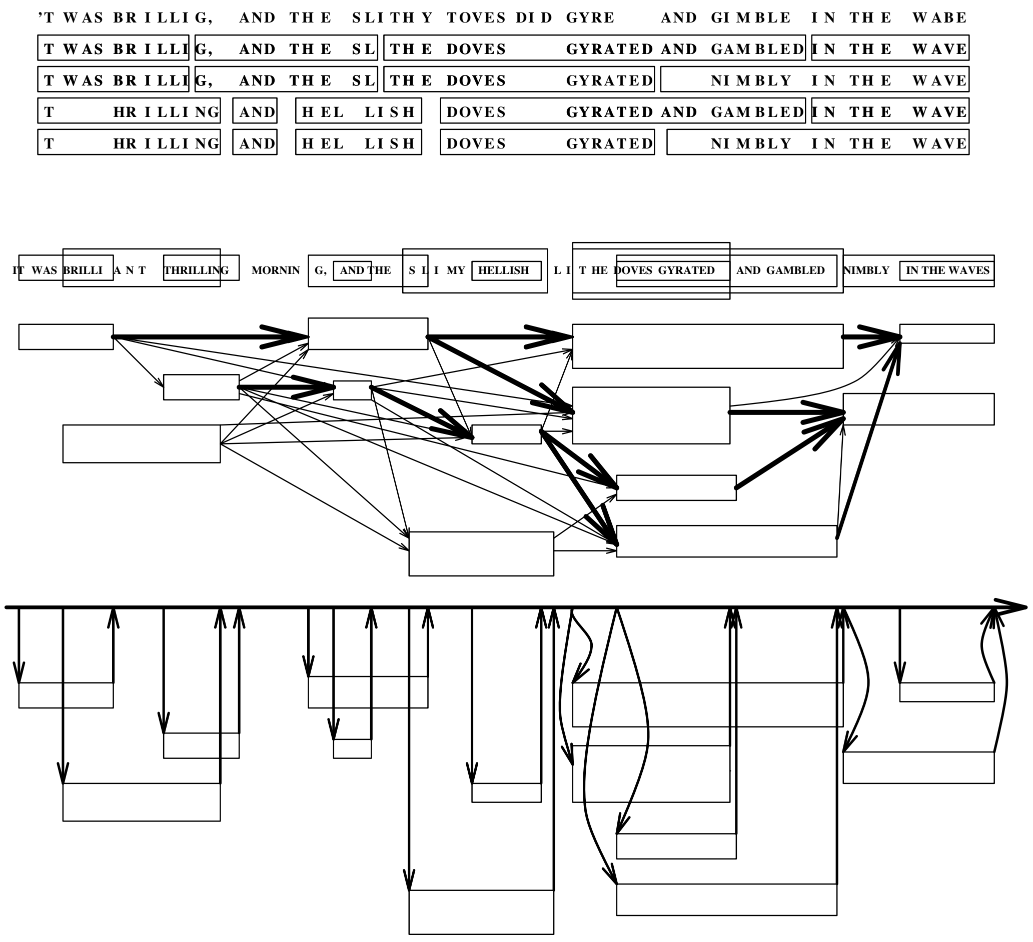

'T WAS BRILLIG, AND THE SLITHY TOVES DID GYRE AND GIMBLE IN THE WABE
T WAS BRILLI G, AND THE SL THE DOVES GYRATED AND GAMBLED IN THE WAVE
T WAS BRILLI G, AND THE SL THE DOVES GYRATED NIMBLY IN THE WAVE
T HRILLING AND HEL LISH DOVES GYRATED AND GAMBLED IN THE WAVE
T HRILLING AND HEL LISH DOVES GYRATED NIMBLY IN THE WAVE
IT WAS BRILLI A N T THRILLING MORNIN G, AND THE S L I MY HELLISH L I T HE DOVES GYRATED AND GAMBLED NIMBLY IN THE WAVES

Figure 6.28 The Spliced Alignment problem: four different block assemblies with the best fit to Lewis Carroll's line (top), the corresponding spliced alignment graph (middle), and the transformation of the spliced alignment graph that helps reduce the running time (bottom).

To describe the dynamic programming recurrence for the Spliced Alignment problem, we first need to define the similarity score between the i-prefix of the *spliced alignment graph* in figure 6.28 and the j-prefix of the target sequence T. The difficulty is that there are typically many different i-prefixes of the graph, since there are multiple blocks containing position i.[29]

Let $B = g_{\mathit{left}} \ldots g_i \ldots g_{\mathit{right}}$ be a candidate exon containing position i in genomic sequence G. Define the *i-prefix* of B as $B(i) = g_{\mathit{left}} \ldots g_i$ and *end*$(B) =$

29. For example, there are two i-prefixes ending with E within HELLISH, and four i-prefixes ending in R within GYRATED in figure 6.28.

right (the words *left* and *right* are used here as indices). If the chain $\Gamma = (B_1, B_2, \ldots, B)$ ends at block B, define $\Gamma^*(i)$ to be the concatenation of all candidate exons in the chain up to (and excluding) B, plus all the characters in B up to i. That is, $\Gamma^*(i) = B_1 \circ B_2 \circ \cdots \circ B(i)$.[30] Finally, let

$$S(i,j,B) = \max_{\text{all chains } \Gamma \text{ ending in } B} s(\Gamma^*(i), T(j)).$$

That is, given i, j, and a candidate exon B that covers position i, $S(i,j,B)$ is the score of the optimal spliced alignment between the i-prefix of G and the j-prefix of T under the assumption that this alignment ends in block B.

The following recurrence allows us to efficiently compute $S(i,j,B)$. For the sake of simplicity we consider sequence alignment with linear gap penalties for insertion or deletion equal to $-\sigma$, and use the scoring matrix δ for matches and mismatches.

The dynamic programming recurrence for the Spliced Alignment problem is broken into two cases depending on whether i is the starting vertex of block B or not. In the latter case, the recurrence is similar to the canonical sequence alignment:

$$S(i,j,B) = \max \begin{cases} S(i-1,j,B) - \sigma \\ S(i,j-1,B) - \sigma \\ S(i-1,j-1,B) + \delta(g_i, t_j) \end{cases}$$

On the other hand, if i is the starting position of block B, then

$$S(i,j,B) = \max \begin{cases} S(i,j-1,B) - \sigma \\ \max_{\text{all blocks } B' \text{ preceding } B} S(end(B'), j-1, B') + \delta(g_i, t_j), \\ \max_{\text{all blocks } B' \text{ preceding } B} S(end(B'), j, B') - \sigma, \end{cases}$$

After computing this three-dimensional table $S(i,j,B)$, the score of the optimal spliced alignment is

$$\max_B S(end(B), m, B),$$

where the maximum is taken over all possible blocks. One can further reduce the number of edges in the spliced alignment graph by making a transfor-

30. The notation $x \circ y$ denotes concatenation of strings x and y.

mation of the graph in figure 6.28 (middle) into a graph shown in figure 6.28 (bottom). The details of the corresponding recurrences are left as a problem at the end of this chapter.

The above description hides some important details of block generation. The simplest approach to the construction of the candidate exon set is to generate all fragments between potential acceptor sites represented by AG and potential donor sites represented by GT, removing possible exons with stop codons in all three frames. However, this approach does not work well since it generates many short blocks. Experiments with the spliced alignment algorithm have shown that incorrect predictions are frequently associated with the *mosaic effect* caused by very short potential exons. The difficulty is that these short exons can be easily combined to fit any target protein, simply because it is easier to construct a given sentence from a thousand random short strings than from the same number of random long strings. For example, with high probability, the phrase "filtration of candidate exons" can be made up from a sample of a thousand random two-letter strings ("fi," "lt," "ra," etc. are likely to be present in this sample). The probability that the same phrase can be made up from a sample of the same number of random five-letter strings is close to zero (even finding the string "filtr" in this sample is unlikely). This observation explains the mosaic effect: if the number of short blocks is high, chains of these blocks can replace actual exons in spliced alignments, thus leading to predictions with an unusually large number of short exons. To avoid the mosaic effect, the candidate exons should be subjected to some filtering procedure.

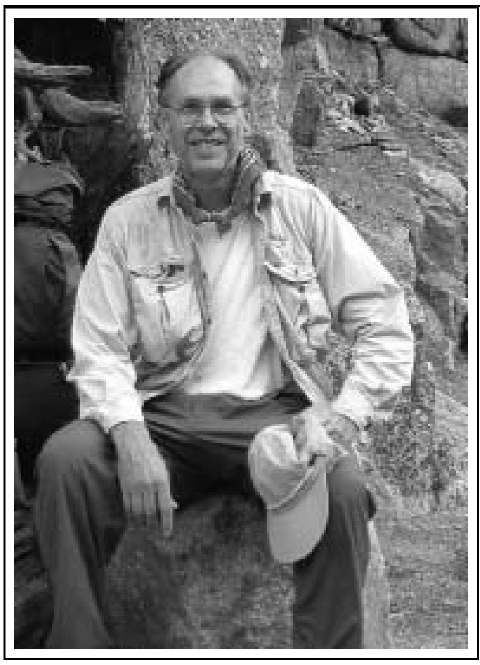
6.15 Notes

Although the first dynamic programming algorithm for DNA sequence comparison was published as early as 1970 by Saul Needleman and Christian Wunsch (79), Russell Doolittle and colleagues used heuristic algorithms to establish the similarity between cancer-causing genes and the PDGF gene in 1983 (28). When Needleman and Wunsch published their paper in 1970, they did not know that a very similar algorithm had been published two years earlier in a pioneering paper on automatic speech recognition (105) (though the details of the algorithms are slightly different, they are both variations of dynamic programming). Earlier still, Vladimir Levenshtein introduced the notion of edit distance in 1966 (64), albeit without an algorithm for computing it. The local alignment algorithm introduced by Temple Smith and

Michael Waterman in 1981 (96) quickly became the most popular alignment tool in computational biology. Later Michael Waterman and Mark Eggert developed an algorithm for finding the k best nonoverlapping local alignments (109). The algorithm for alignment with affine gap penalties was the work of Osamu Gotoh (42).

The progressive multiple alignment approach, initially explored by Da-Fei Feng and Russell Doolittle [Feng and Doolittle, 1987 (36)], resulted in many practical algorithms, with CLUSTAL (48) one of the most popular.

The cellular process of splicing was discovered in the laboratories of Phillip Sharp (12) and Richard Roberts (21). Applications of both Markov models and in-frame hexamer count statistics for gene prediction were proposed by Borodovsky and McInnich (14). Chris Burge and Samuel Karlin developed an HMM approach to gene prediction that resulted in the popular GENSCAN algorithm in 1997 (20). In 1995 Snyder and Stormo (97) developed a similarity-based gene prediction algorithm that amounts to the solution of a problem that is similar to the Exon Chaining problem. The spliced alignment algorithm was developed by Mikhail Gelfand and colleagues in 1996 (41).



Michael Waterman (born 1942 in Oregon) currently holds an Endowed Associates Chair at the University of Southern California. His BS in Mathematics is from Oregon State University, and his PhD in Statistics and Probability is from Michigan State University. He was named a Guggenheim Fellow in 1995 and was elected to the National Academy of Sciences in 2001. In 2002 he received a Gairdner Foundation Award. He is one of the founding fathers of bioinformatics whose fundamental contributions to the area go back to the 1970s when he worked at Los Alamos National Laboratories. Waterman says:

I went to college to escape what I considered to be a dull and dreary existence of raising livestock on pasture land in western Oregon where my family has lived since 1911. My goal was to find an occupation with a steady income where I could look forward to going to work; this eliminated ranching and logging (which was how I spent my college summers). Research and teaching didn't seem possible or even desirable, but I went on for a PhD because such a job did not appear.

In graduate school at Michigan State I found a wonderful advisor in John Kinney from whom I learned ergodic and information theory. John aimed me at a branch of number theory for a thesis. We were doing statistical properties of the iteration of deterministic functions long before that became a fad. I began using computers to explore iteration, something which puzzled certain of my professors who felt I was wasting time I could be spending proving theorems. After graduation and taking a nonresearch job at a small school in Idaho, my work in iteration led to my first summer visit to Los Alamos National labs. Later I met Temple Smith there in 1973 and was drawn into problems from biology. Later I wrote in my book of New Mexico essays *Skiing the Sun* (107):

> I was an innocent mathematician until the summer of 1974. It was then than I met Temple Ferris Smith and for two months was cooped up with him in an office at Los Alamos National Laboratories. That experience transformed my research, my life, and perhaps my sanity. Soon after we met, he pulled out a little blackboard and started lecturing me about biology: what it was, what was important, what was going on. Somewhere in there by implication was what we should work on, but the truth be told he didn't know what that was either. I was totally confused: amino acids, nucleosides, beta sheets. What were these things? Where was the mathematics?

I knew no modern biology, but studying alignment and evolution was quite attractive to me. The most fun was formulating problems, and in my opinion that remains the most important aspect of our subject. Temple and I spent days and weeks trying to puzzle out what we should be working on. Charles DeLisi, a biophysicist who went on to play a key role in jump-starting the Human Genome Project, was in T-10 (theoretical biology) at the lab. When he saw the progress we had made on alignment problems, he came to me and said there was another problem which should interest me. This was the RNA folding problem which was almost untouched. Tinoco had published the idea of making a base-pair matrix for a sequence and that was it. By the fall of 1974 I had seen the neat connection between alignment and folding, and the following summer I wrote a long manuscript that defined the objects of study, established some of their properties, explicitly stated the basic problem of folding (which included free energies for all structural components), and finally gave algorithms for its solution. I had previously wondered what such a discovery might feel like, and it was wonderfully satisfying. However it felt entirely like exploration and not a grand triumph of creation as I had expected. In fact I had always wanted to be an explorer and regretted the end of the American frontier; wandering about this new RNA landscape was a great joy, just as I had thought when I was a child trying to transport myself by daydreams out of my family's fields into some new and unsettled country.

6.16 Problems

In 1879, Lewis Carroll proposed the following puzzle to the readers of *Vanity Fair*: transform one English word into another by going through a series of intermediate English words, where each word in the sequence differs from the next by only one substitution. To transform $head$ into $tail$ one can use four intermediates: $head \rightarrow heal \rightarrow teal \rightarrow tell \rightarrow tall \rightarrow tail$. We say that two words $\mathbf{v}$ and $\mathbf{w}$ are equivalent if $\mathbf{v}$ can be transformed into $\mathbf{w}$ by substituting individual letters in such a way that all intermediate words are English words present in an English dictionary.

Problem 6.1
Find an algorithm to solve the following *Equivalent Words problem*.

Equivalent Words Problem:
Given two words and a dictionary, find out whether the words are equivalent.

Input: The dictionary, $\mathcal{D}$ (a set of words), and two words $\mathbf{v}$ and $\mathbf{w}$ from the dictionary.

Output: A transformation of $\mathbf{v}$ into $\mathbf{w}$ by substitutions such that all intermediate words belong to $\mathcal{D}$. If no transformation is possible, output "$\mathbf{v}$ and $\mathbf{w}$ are not equivalent."

Given a dictionary $\mathcal{D}$, the *Lewis Carroll distance*, $d_{LC}(\mathbf{v}, \mathbf{w})$, between words $\mathbf{v}$ and $\mathbf{w}$ is defined as the smallest number of substitutions needed to transform $\mathbf{v}$ into $\mathbf{w}$ in such a way that all intermediate words in the transformation are in the dictionary $\mathcal{D}$. We define $d_{LC}(\mathbf{v}, \mathbf{w}) = \infty$ if $\mathbf{v}$ and $\mathbf{w}$ are not equivalent.

Problem 6.2
Find an algorithm to solve the following *Lewis Carroll problem*.

Lewis Carroll Problem:
Given two words and a dictionary, find the Lewis Carroll distance between these words.

Input: The dictionary $\mathcal{D}$, and two words $\mathbf{v}$ and $\mathbf{w}$ from the dictionary.

Output: $d_{LC}(\mathbf{v}, \mathbf{w})$

Problem 6.3
Find an algorithm to solve a generalization of the Lewis Carroll problem when insertions, deletions, and substitutions are allowed (rather than only substitutions).

Problem 6.4

Modify DPCHANGE to return not only the smallest *number* of coins but also the correct combination of coins.

Problem 6.5

Let $s(\mathbf{v}, \mathbf{w})$ be the length of a longest common subsequence of the strings $\mathbf{v}$ and $\mathbf{w}$ and $d(\mathbf{v}, \mathbf{w})$ be the edit distance between $\mathbf{v}$ and $\mathbf{w}$ under the assumption that insertions and deletions are the only allowed operations. Prove that $d(\mathbf{v}, \mathbf{w}) = n+m-2s(\mathbf{v}, \mathbf{w})$, where n is the length of $\mathbf{v}$ and m is the length of $\mathbf{w}$.

Problem 6.6

Find the number of different paths from *source* $(0, 0)$ to *sink* (n, m) in an $n \times m$ rectangular grid.

Problem 6.7

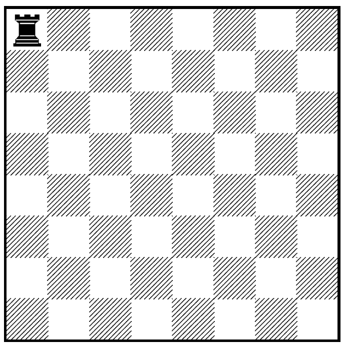
Can you find an approximation ratio of the greedy algorithm for the Manhattan Tourist problem?

Problem 6.8

Let $\mathbf{v} = v_1 v_2 \cdots v_n$ be a string, and let P be a $4 \times m$ profile. Generalize the sequence alignment algorithm for aligning a sequence against a profile. Write the corresponding recurrence (in lieu of pseudocode), and estimate the amount of time that your algorithm will take with respect to n and m.

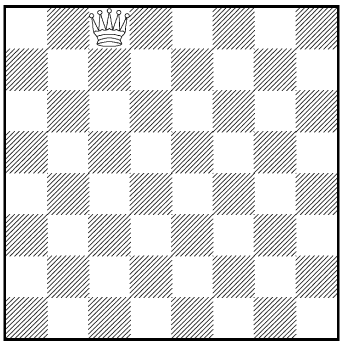
Problem 6.9

There are only two buttons inside an elevator in a building with 50 floors. The elevator goes 11 floors up if the first button is pressed and 6 floors down if the second button is pressed. Is it possible to get from floor 32 to floor 33? What is the minimum number of buttons one has to press to do so? What is the shortest time one needs to get from floor 32 to floor 33 (time is proportional to the number of floors that are passed on the way)?



Problem 6.10

A rook stands on the upper left square of a chessboard. Two players make turns moving the rook either horizontally to the right or vertically downward (as many squares as they want). The player who can place the rook on the lower right square of the chessboard wins. Who will win? Describe the winning strategy.



Problem 6.11

A queen stands on the third square of the uppermost row of a chessboard. Two players take turns moving the queen either horizontally to the right or vertically downward or diagonally in the southeast direction (as many squares as they want). The player who can place the queen on the lower right square of the chessboard wins. Who will win? Describe the winning strategy.

Problem 6.12

Two players play the following game with two "chromosomes" of length n and m nucleotides. At every turn a player can destroy one of the chromosomes and break another one into two nonempty parts. For example, the first player can destroy a chromosome of length n and break another chromosome into two chromosomes of length $\frac{m}{3}$ and $m - \frac{m}{3}$. The player left with two single-nucleotide chromosomes loses. Who will win? Describe the winning strategy for each n and m.

Problem 6.13

Two players play the following game with two sequences of length n and m nucleotides. At every turn a player can either delete an arbitrary number of nucleotides from one sequence or an equal (but still arbitrary) number of nucleotides from both sequences. The player who deletes the last nucleotide wins. Who will win? Describe the winning strategy for each n and m.

Problem 6.14

Two players play the following game with two sequences of length n and m nucleotides. At every turn a player must delete two nucleotides from one sequence (either the first or the second) and one nucleotide from the other. The player who cannot move loses. Who will win? Describe the winning strategy for each n and m.

Problem 6.15

Two players play the following game with a nucleotide sequence of length n. At every turn a player may delete either one or two nucleotides from the sequence. The player who deletes the last letter wins. Who will win? Describe the winning strategy for each n.

Problem 6.16

Two players play the following game with a nucleotide sequence of length $n = n_A + n_T + n_C + n_G$, where n_A, n_T, n_C, and n_G are the number of A,T,C, and G in the sequence. At every turn a player may delete either one or two nucleotides from the sequence. The player who is left with a uni-nucleotide sequence of an arbitrary length (i.e., the sequence containing only one of 4 possible nucleotides) loses. Who will win? Describe the winning strategy for each n_A, n_T, n_C, and n_G.

Problem 6.17

What is the optimal global alignment for APPLE and HAPPE? Show all optimal alignments and the corresponding paths under the match premium $+1$, mismatch penalty -1, and indel penalty -1.

Problem 6.18

What is the optimal global alignment for MOAT and BOAST? Show all optimal alignments and the corresponding paths under the scoring matrix below and indel penalty -1.

	A	B	M	O	S	T
A	1	-1	-1	-2	-2	-3
B		1	-1	-1	-2	-2
M			2	-1	-1	-2
O				1	-1	-1
S					1	-1
T						2

Problem 6.19

Fill the global alignment dynamic programming matrix for strings AT and AAGT with affine scoring function defined by match premium 0, mismatch penalty -1, gap opening penalty -1, and gap extension penalty -1. Find all optimal global alignments.

Problem 6.20

Consider the sequences $\mathbf{v}$ = TACGGGTAT and $\mathbf{w}$ = GGACGTACG. Assume that the match premium is $+1$ and that the mismatch and indel penalties are -1.

- Fill out the dynamic programming table for a global alignment between $\mathbf{v}$ and $\mathbf{w}$. Draw arrows in the cells to store the backtrack information. What is the score of the optimal global alignment and what alignment does this score correspond to?
- Fill out the dynamic programming table for a local alignment between $\mathbf{v}$ and $\mathbf{w}$. Draw arrows in the cells to store the backtrack information. What is the score of the optimal local alignment in this case and what alignment achieves this score?
- Suppose we use an affine gap penalty where it costs -20 to open a gap, and -1 to extend it. Scores of matches and mismatches are unchanged. What is the optimal global alignment in this case and what score does it achieve?

Problem 6.21

For a pair of strings $\mathbf{v} = v_1 \ldots v_n$ and $\mathbf{w} = w_1 \ldots w_m$, define $M(\mathbf{v}, \mathbf{w})$ to be the matrix whose (i, j)th entry is the score of the optimal global alignment which aligns the character v_i with the character w_j. Give an $O(nm)$ algorithm which computes $M(\mathbf{v}, \mathbf{w})$.

Define an *overlap alignment* between two sequences $\mathbf{v} = v_1 \ldots v_n$ and $\mathbf{w} = w_1 \ldots w_m$ to be an alignment between a suffix of $\mathbf{v}$ and a prefix of $\mathbf{w}$. For example, if $\mathbf{v} =$ TATATA and $\mathbf{w} =$ AAATTT, then a (not necessarily optimal) overlap alignment between $\mathbf{v}$ and $\mathbf{w}$ is

```
ATA
AAA
```

Optimal overlap alignment is an alignment that maximizes the global alignment score between $v_i, \ldots, v_n$ and $w_1, \ldots w_j$, where the maximum is taken over all suffixes $v_i, \ldots, v_n$ of $\mathbf{v}$ and all prefixes $w_1, \ldots w_j$ of $\mathbf{w}$.

Problem 6.22

Give an algorithm which computes the optimal overlap alignment, and runs in time $O(nm)$.

Suppose that we have sequences $\mathbf{v} = v_1 \ldots v_n$ and $\mathbf{w} = w_1 \ldots w_m$, where $\mathbf{v}$ is longer than $\mathbf{w}$. We wish to find a substring of $\mathbf{v}$ which best matches *all* of $\mathbf{w}$. Global alignment won't work because it would try to align all of $\mathbf{v}$. Local alignment won't work because it may not align all of $\mathbf{w}$. Therefore this is a distinct problem which we call the *Fitting problem*. *Fitting* a sequence $\mathbf{w}$ into a sequence $\mathbf{v}$ is a problem of finding a substring $\mathbf{v}'$ of $\mathbf{v}$ that maximizes the score of alignment $s(\mathbf{v}', \mathbf{w})$ among all substrings of $\mathbf{v}$. For example, if $\mathbf{v} =$ GTAGGCTTAAGGTTA and $\mathbf{w} =$ TAGATA, the best alignments might be

	global	local	fitting
$\mathbf{v}$	`GTAGGCTTAAGGTTA`	`TAG`	`TAGGCTTA`
$\mathbf{w}$	`-TAG----A---T-A`	`TAG`	`TAGA--TA`
score	-3	3	2

The scores are computed as 1 for match, -1 for mismatch or indel. Note that the optimal local alignment is not a valid fitting alignment. On the other hand, the optimal global alignment contains a valid fitting alignment, but it achieves a suboptimal score among all fitting alignments.

Problem 6.23

Give an algorithm which computes the optimal fitting alignment. Explain how to fill in the first row and column of the dynamic programming table and give a recurrence to fill in the rest of the table. Give a method to find the best alignment once the table is filled in. The algorithm should run in time $O(nm)$.

We have studied two approaches to sequence alignment: global and local alignment. There is a middle ground: an approach known as *semiglobal* alignment. In semiglobal alignment, the entire sequences are aligned (as in global alignment). What makes it semiglobal is that the "internal gaps" of the alignment are counted, but the "gaps on the end" are not. For example, consider the following two alternative alignments:

```
Sequence 1:   CAGCA-CTTGGATTCTCGG
Sequence 2:   ---CAGCGTGG--------
```

```
Sequence 1:   CAGCACTTGGATTCTCGG
Sequence 2:   CAGC-----G-T----GG
```

The first alignment has 6 matches, 1 mismatch, and 12 gaps. The second alignment has 8 matches, no mismatches, and 10 gaps. Using the simplest scoring scheme ($+1$ match, -1 mismatch, -1 gap), the score for the first alignment is -7, and the score for the second alignment is -2, so we would prefer the second alignment. However, the first alignment is more biologically realistic. To get an algorithm which prefers the first alignment to the second, we can not count the gaps "on the ends."

Under this new ("semiglobal") approach, the first alignment would have 6 matches, 1 mismatch, and 1 gap, while the second alignment would still have 8 matches, no mismatches, and 10 gaps. Now the first alignment would have a score of 4, and the second alignment would have a score of -2, so the first alignment would have a better score.

Note the similarities and the differences between the Fitting problem and the Semiglobal Alignment problem as illustrated by the semiglobal—but not fitting—alignment of ACGTCAT against TCATGCA:

```
Sequence 1:   ACGTCAT---
Sequence 2:   ---TCATGCA
```

Problem 6.24

Devise an efficient algorithm for the Semiglobal Alignment problem and illustrate its work on the sequences ACAGATA and AGT. For scoring, use the match premium $+1$, mismatch penalty -1, and indel penalty -1.

Define a *NoDeletion* global alignment to be an alignment between two sequences $\mathbf{v} = v_1 v_2 \ldots v_n$ and $\mathbf{w} = w_1 w_2 \ldots w_m$, where only matches, mismatches, and insertions are allowed. That is, there can be no deletions from $\mathbf{v}$ to $\mathbf{w}$ (i.e., all letters of $\mathbf{w}$ occur in the alignment with no spaces). Clearly we must have $m \geq n$ and let $k = m - n$.

Problem 6.25

Give an $O(nk)$ algorithm to find the optimal *NoDeletion* global alignment (note the improvement over the $O(nm)$ algorithm when k is small).

Problem 6.26

Substrings $v_i, \ldots, v_{i+k}$ and $v_{i'}, \ldots, v_{i'+k}$ of the string $v_1, \ldots, v_n$ form a *substring pair* if $i' - i + k > MinGap$, where $MinGap$ is a parameter. Define the substring pair score as the (global) alignment score of $v_i, \ldots, v_{i+k}$ and $v_{i'}, \ldots, v_{i'+k}$. Design an algorithm that finds a substring pair with maximum score.

Problem 6.27

For a parameter k, compute the global alignment between two strings, subject to the constraint that the alignment contains at most k gaps (blocks of consecutive indels).

Nucleotide sequences are sometimes written in an alphabet with five characters: A, T, G, C, and N, where N stands for an unspecified nucleotide (in essence, a wild-card). Biologists may use N when sequencing does not allow one to unambiguously infer the identity of a nucleotide at a specific position. A sequence with an N is referred to as a *degenerate* string; for example, ATTNG may correspond to four different interpretations: ATTAG, ATTTG, ATTGG, and ATTCG. In general, a sequence with k unspecified nucleotides N will have 4^k different interpretations.

Problem 6.28

Given a non-degenerate string, $\mathbf{v}$, and a degenerate string $\mathbf{w}$ that contains k Ns, devise a method to find the best interpretation of $\mathbf{w}$ according to $\mathbf{v}$. That is, out of all 4^k possible interpretations of $\mathbf{w}$, find $\mathbf{w}'$ with the minimum alignment score $s(\mathbf{w}', \mathbf{v})$.

Problem 6.29

Given a non-degenerate string, $\mathbf{v}$, and a degenerate string $\mathbf{w}$ that contains k Ns, devise a method to find the worst interpretation of $\mathbf{w}$ according to $\mathbf{v}$. That is, out of all 4^k possible interpretations of $\mathbf{w}$, find $\mathbf{w}'$ with the minimum alignment score $s(\mathbf{w}', \mathbf{v})$.

Problem 6.30

Given two strings $\mathbf{v}_1$ and $\mathbf{v}_2$, explain how to construct a string $\mathbf{w}$ minimizing

$$|d(\mathbf{v}_1, \mathbf{w}) - d(\mathbf{v}_2, \mathbf{w})|$$

such that

$$d(\mathbf{v}_1, \mathbf{w}) + d(\mathbf{v}_2, \mathbf{w}) = d(\mathbf{v}_1, \mathbf{v_2}).$$

$d(\cdot, \cdot)$ is the edit distance between two strings.

Problem 6.31

Given two strings $\mathbf{v}_1$ and $\mathbf{v}_2$ and a text $\mathbf{w}$, find whether there is an occurrence of $\mathbf{v}_1$ and $\mathbf{v}_2$ interwoven (without spaces) in $\mathbf{w}$. For example, the strings **abac** and bbc occur interwoven in **ca**b**ba**b**c**cdw. Give an efficient algorithm for this problem.

A string $\mathbf{x}$ is called a *supersequence* of a string $\mathbf{v}$ if $\mathbf{v}$ is a subsequence of $\mathbf{x}$. For example, ABLUE is a supersequence for BLUE and ABLE.

Problem 6.32

Given strings $\mathbf{v}$ and $\mathbf{w}$, devise an algorithm to find the shortest supersequence for both $\mathbf{v}$ and $\mathbf{w}$.

A *tandem repeat* P^k of a pattern $P = p_1 \ldots p_n$ is a pattern of length $n \cdot k$ formed by concatenation of k copies of P. Let P be a pattern and T be a text of length m. The *Tandem Repeat* problem is to find a best local alignment of T with some tandem repeat of P. This amounts to aligning P^k against T and the standard local alignment algorithm solves this problem in $O(km^2)$ time.

Problem 6.33

Devise a faster algorithm for solving the tandem repeat problem.

An alignment of circular strings is defined as an alignment of linear strings formed by cutting (linearizing) these circular strings at arbitrary positions. The following problem asks to find the cut points of two circular strings that maximize the alignment of the resulting linear strings.

Problem 6.34

Devise an efficient algorithm to find an optimal alignment (local and global) of circular strings.

The early graphical method for comparing nucleotide sequences—dot matrices—still yields one of the best visual representations of sequence similarities. The axes in a dot matrix correspond to the two sequences $\mathbf{v} = v_1 \ldots v_n$ and $\mathbf{w} = w_1 \ldots w_m$. A dot is placed at coordinates (i, j) if the substrings $s_i \ldots s_{i+k}$ and $t_j \ldots t_{j+k}$ are sufficiently similar. Two such substrings are considered to be sufficiently similar if the Hamming distance between them is at most d.

When the sequences are very long, it is not necessary to show exact coordinates; figure 6.29 is based on the sequences corresponding to the β-globin gene in human and mouse. In these plots each axis is on the order of 1000 base pairs long, $k = 10$ and $d = 2$.

Problem 6.35

Use figure 6.29 to answer the following questions:

- How many exons are in the human β-globulin gene?
- The dot matrix in figure 6.29 (top) is between the mouse and human genes (i.e., all introns and exons are present). Do you think the number of exons in the β-globulin gene is different in the human genome as compared to the mouse genome?
- Label segments of the axes of the human and mouse genes in figure 6.29 to show where the introns and exons would be located.

A local alignment between two different strings $\mathbf{v}$ and $\mathbf{w}$ finds a pair of substrings, one in $\mathbf{v}$ and the other in $\mathbf{w}$, with maximum similarity. Suppose that we want to find a pair of (nonoverlapping) substrings *within* string $\mathbf{v}$ with maximum similarity (*Optimal Inexact Repeat problem*). Computing an optimal local alignment between $\mathbf{v}$ and $\mathbf{v}$ does not solve the problem, since the resulting alignment may correspond to overlapping substrings.

Problem 6.36

Devise an algorithm for the Optimal Inexact Repeat problem.

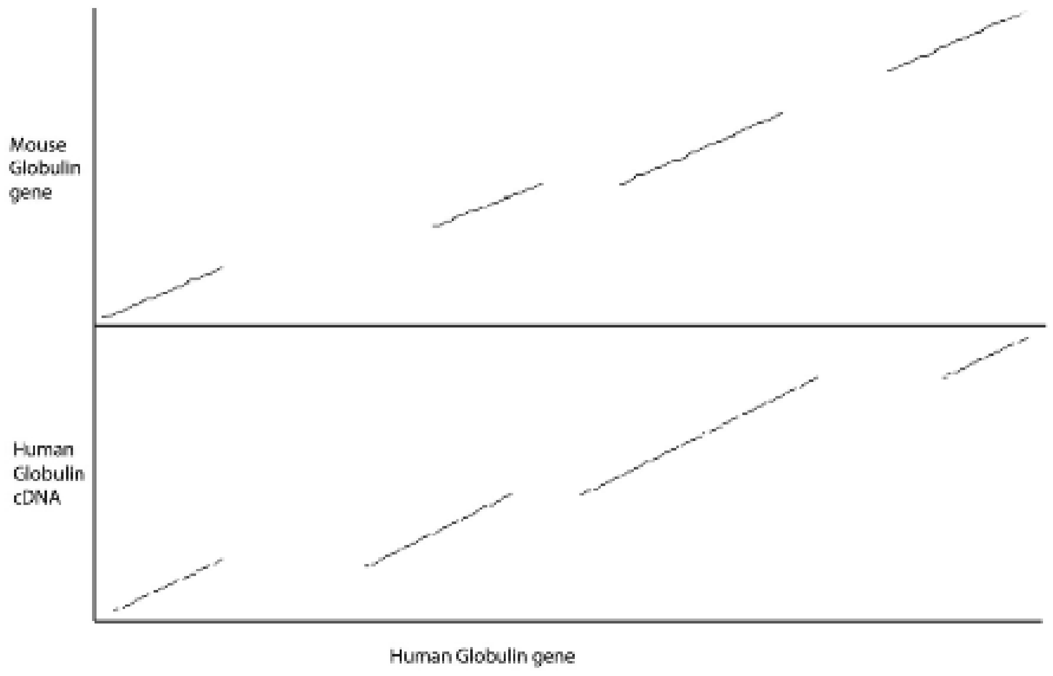


Figure 6.29 Human β-globulin cDNA vs. the gene sequence in two organisms.

In the *chimeric alignment* problem, a string $\mathbf{v}$ and a set of strings $\{\mathbf{w}_1, \ldots, \mathbf{w}_N\}$ are given, and the problem is to find $\max_{1 \le i,j \le N} s(\mathbf{v}, \mathbf{w}_i \circ \mathbf{w}_j)$ where $\mathbf{w}_i \circ \mathbf{w}_j$ is the concatenation of $\mathbf{w}_i$ and $\mathbf{w}_j$ ($s(\cdot, \cdot)$ stand for the score of optimal global alignment).

Problem 6.37

Devise an efficient algorithm for the chimeric alignment problem.

A virus infects a bacterium, and modifies a replication process in the bacterium by inserting

at every A, a polyA of length 1 to 5.
at every C, a polyC of length 1 to 10.
at every G, a polyG of arbitrary length ≥ 1.
at every T, a polyT of arbitrary length ≥ 1.

No gaps or other insertions are allowed in the virally modified DNA. For example, the sequence AAATAAAGGGGCCCCCTTTTTTTCC is an infected version of ATAGCTC.

Problem 6.38

Given sequences $\mathbf{v}$ and $\mathbf{w}$, describe an efficient algorithm that will determine if $\mathbf{v}$ could be an infected version of $\mathbf{w}$.

Problem 6.39

Now assume that for each nucleotide (A, C, G, T) the virus will either delete a letter or insert a run of the letter of arbitrary length. Give an efficient algorithm to detect if $\mathbf{v}$ could be an infected version of $\mathbf{w}$ under these circumstances.

Problem 6.40

Define homodeletion as an operation of deleting a run of the same nucleotide and homoinsertion as an operation of inserting a run of the same nucleotide. For example, ACAAAAAAGCTTTTA is obtained from ACGCTTTTA by a homoinsertions of a run of six A, while ACGCTA is obtained from ACGCTTTTA by homodeletion of a run of three T. The homo-edit distance between two sequences is defined as the minimum number of homodeletions and homoinsertions to transform one sequence into another. Give an efficient algorithm to compute the homoedit distance between two arbitrary strings.

Problem 6.41

Suppose we wish to find an optimal global alignment using a scoring scheme with an affine *mismatch* penalty. That is, the premium for a match is $+1$, the penalty for an indel is $-\rho$, and the penalty for x consecutive mismatches is $-(\rho + \sigma x)$. Give an $O(nm)$ algorithm to align two sequences of length n and m with an affine mismatch penalty. Explain how to construct an appropriate "Manhattan" graph and estimate the running time of your algorithm.

Problem 6.42

Define a *NoDiagonal* global alignment to be an alignment where we disallow matches and mismatches. That is, only indels are allowed. Give a $\Theta(nm)$ algorithm to determine the number of *NoDiagonal* alignments between a sequence of length n and a sequence of length m. Give a closed-form formula for the number of *NoDiagonal* global alignments (e.g., something of the form $f(n,m) = n^2m - \sqrt{n!} + \pi n^m$).

Problem 6.43

Estimate the number of different (not necessarily optimal) global alignments between two n-letter sequences.

Problem 6.44

Devise an algorithm to compute the number of distinct optimal global alignments (optimal paths in edit graph) between a pair of strings.

Problem 6.45

Estimate the number of different (not necessarily optimal) local alignments between two n-letter sequences.

Problem 6.46

Devise an algorithm to compute the number of distinct optimal local alignments (optimal paths in local alignment edit graph) between a pair of strings.

Problem 6.47

Let $s_{i,j}$ be a dynamic programming matrix computed for the LCS problem. Prove that for any i and j, the difference between $s_{i+1,j}$ and $s_{i,j}$ is at most 1.

Let $i_1, \ldots, i_n$ be a sequence of numbers. A subsequence of $i_1, \ldots, i_n$ is called an *increasing subsequence* if elements of this subsequence go in increasing order. Decreasing subsequences are defined similarly. For example, elements 2, 6, 7, 9 of the sequence 8, 2, 1, 6, 5, 7, 4, 3, 9 form an increasing subsequence, while elements 8, 7, 4, 3 form a decreasing subsequence.

Problem 6.48

Devise an efficient algorithm for finding longest increasing and decreasing subsequences in a permutation of integers.

Problem 6.49

Show that in any permutation of n distinct integers, there is either an increasing subsequence of length at least $\sqrt{n}$ or a decreasing subsequence of length at least $\sqrt{n}$.

A subsequence σ of permutation π is 2-*increasing* if, as a set, it can be written as

$$\sigma = \sigma_1 \cup \sigma_2$$

where σ_1 and σ_2 are increasing subsequences of π. For example, 1, 5, 7, 9 and 2, 6 are increasing subsequences of $\pi = 821657439$ forming a 2-increasing subsequence 2, 1, 6, 5, 7, 9 consisting of six elements.

Problem 6.50

Devise an algorithm to find a longest 2-increasing subsequence.

RNAs adopt complex three-dimensional structures that are important for many biological functions. Pairs of positions in RNA with complementary nucleotides can form *bonds*. Bonds (i, j) and (i', j') are interleaving if $i < i' < j < j'$ and noninterleaving otherwise (fig. 6.30). Every set of noninterleaving bonds corresponds to a potential RNA structure. In a very naive formulation of the RNA folding problem, one tries to find a maximum set of noninterleaving bonds. The more adequate model, attempting to find a fold with the minimum energy, is much more difficult.

Problem 6.51

Develop a dynamic programming algorithm for finding the largest set of noninterleaving bonds given an RNA sequences.

The human genome can be viewed as a string of n ($\approx$ 3 billion) nucleotides, partitioned into substrings representing chromosomes. However, for many decades, biologists used a different *band* representation of the genome that is obtained via traditional light microscopy. Figure 6.31 shows 48 bands (as seen on chromosme 4) out of 862 observable bands for the entire human genome. Although several factors (e.g., local G/C frequency) have been postulated to govern the formation of these banding patterns, the mechanism behind their formation remains poorly understood. A mapping between the human genomic sequence (which itself only became available in 2001) and the banding pattern representation would be useful to leverage sequence level

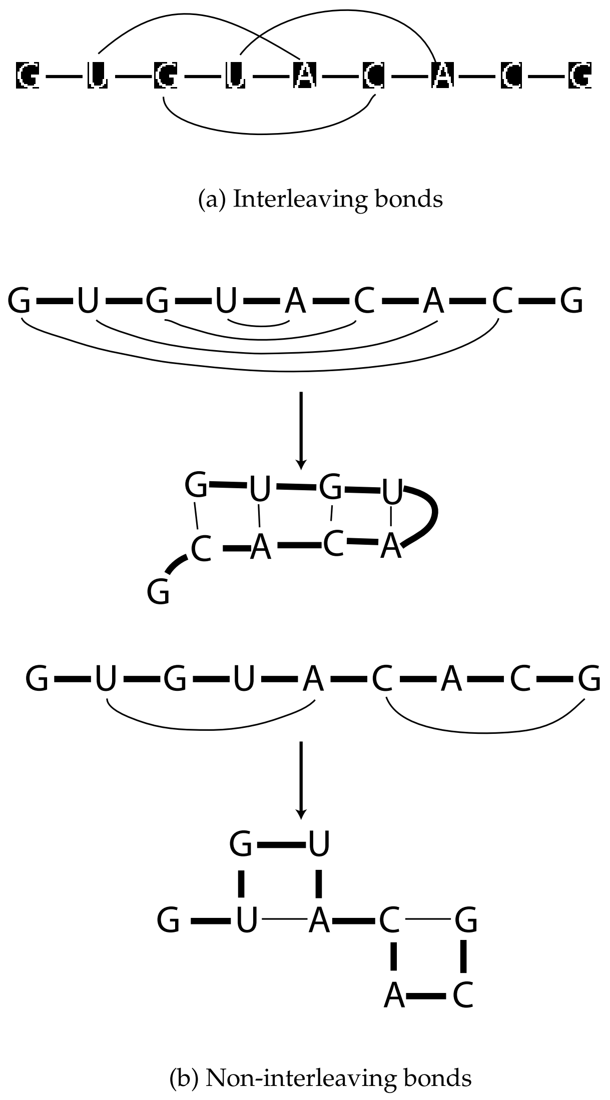


Figure 6.30 Interleaving and noninterleaving bonds in RNA folding.

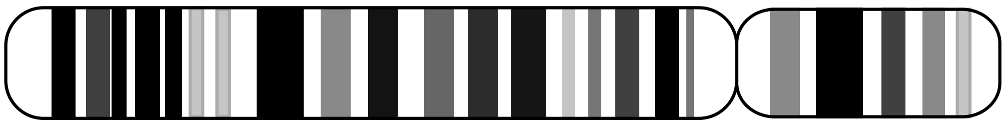


Figure 6.31 Band patterns on human chromosome 4.

gene information against diseases that have been associated with certain band positions. However, until recently, no mapping between these two representations of the genome has been known.

The Band Positioning problem is to find the starting and ending nucleotide positions for each band in the genome (for simplicity we assume that all chromosomes are concatenated to form a single coordinate system). In other words, the Band Positioning problem is to find an increasing array $start(b)$, that contains the starting nucleotide position for each band b in the genome. Each band b begins at the nucleotide given by $start(b)$ and ends at $start(b+1)-1$.[31]

A naive approach to this problem would be to use observed band width data to compute the nucleotide positions. However, this solution is inaccurate because it assumes that band width is perfectly correlated with its length in nucleotides. In reality, this correlation is often quite poor and a different approach is needed.

In the last decade biologists have performed a large number of *FISH (fluorescent in situ hybridization)* experiments that can help to solve the Band Positioning problem. FISH data consist of pairs (x, b), where x is a position in the genome, and b is the index of the band that contains x . FISH data are often subject to experimental error, so some FISH data points may contradict each other.

Given a solution $start(b)$ $(1 \leq b \leq 862)$ of the Band Positioning problem, we define its *FISH quality* as the number of FISH experiments that it supports, that is, the number of FISH experiments (x, b) such that $start(b) \leq x < start(b+1)$.

Problem 6.52

Find a solution to the Band Positioning problem that maximizes its FISH quality.

The FISH quality parameter ignores the width of the bands. A more adequate formulation is to find an optimal solution of the Band Positioning problem that is consistent with band width data, that is, the solution that minimizes

$$\sum_{b=1}^{862} |\mathit{width}(i) - (\mathit{start}(b+1) - \mathit{start}(b))|$$

, where $\mathit{width}(i)$ is the estimated width of the ith band.

Problem 6.53

Find an optimal solution of the Band Positioning problem that minimizes

$$\sum_{b=1}^{862} |width(i) - (start(b+1) - start(b))|$$

Problem 6.54

Describe recurrence relations for multiple alignment of 4 sequences under the SP (sum-of-pairs) scoring rule.

31. For simplicity we assume that $start(863) = n + 1$ thus implying that the last 862th band starts at the nucleotide $start(862)$ and ends at n.

Problem 6.55

Develop a likelihood ratio approach and design an algorithm that utilizes codon usage arrays for gene prediction.

Problem 6.56

Consider the Exon Chaining problem in the case when all intervals have the same weight. Design a greedy algorithm that finds an optimal solution for this limited case of the problem.

Problem 6.57

Estimate the running time of the spliced alignment algorithm. Improve the running time by transforming the spliced alignment graph into a graph with a smaller number of edges. This transformation is hinted at in figure 6.28.

Introns are spliced out of pre-mRNA during mRNA processing and biologists can perform *cDNA sequencing* that provides the nucleotide sequence complementary to the mRNA. The cDNA, therefore, represents the concatenation of exons of a gene. Consequently the exon-intron structure can be determined by aligning the cDNA against the genomic DNA with the aligned regions representing the exons and the large gaps representing the introns. This alignment can be aided by the knowledge of the conserved donor and acceptor splice site sequences (GT at the 5′ splice site and AG at the 3′ splice site).

While a spliced alignment can be used to solve this *cDNA Alignment* problem there exists a faster algorithm to align cDNA against genomic sequence. One approach is to introduce gap penalties that would adequately account for gaps in the cDNA Alignment problem. When aligning cDNA against genomic sequences we want to allow long internal gaps in the cDNA sequence. In addition, long gaps that respect the consensus sequences at the intron-exon junctions are favored over gaps that do not satisfy this property. Such gaps that exceed a given length threshold and respect the donor and acceptor sites should be assigned a constant penalty. This penalty is lower than the affine penalty for long gaps that do not respect the splice site consensus. The input to the cDNA Alignment problem is genomic sequence $\mathbf{v}$, cDNA sequence $\mathbf{w}$, match, mismatch, gap opening and gap extension parameters, as well as L (minimum intron length) and δ_L (fixed penalty for gaps longer than L that respect the consensus sequences). The output is an alignment of $\mathbf{v}$ and $\mathbf{w}$ where aligned regions represent putative exons and gaps in $\mathbf{v}$ represent putative introns.

Problem 6.58

Devise an efficient algorithm for the cDNA Alignment problem.

The spliced alignment algorithm finds exons in genomic DNA by using a related protein as a template. What if a template is not a protein but another (uninterpreted) genomic DNA sequence? Or, in other words, can (unannotated) mouse genomic DNA be used to predict human genes?

Problem 6.59

Generalize the spliced alignment algorithm for alignment of one genomic sequence against another.

Problem 6.60

For simplicity, the Spliced Alignment problem assumes that the genomic sequence and the target protein sequence are both written in the same alphabet. Modify the recurrence relations to handle the case when they are written in different alphabets (specifically, proteins are written in a twenty letter alphabet, and DNA is written in a four letter alphabet).