for most instances of the PDP since usually only one of the two alternatives, "left" or "right," is viable at any step. It was not clear for a number of years whether or not this algorithm is polynomial in the worst case—sometimes both alternatives are viable. If both "left" and "right" alternatives hold and if this continues to happen in future steps of the algorithm, then the performance of the algorithm starts growing as 2^k where k is the number of such "ambiguous" steps.[8]

Let $T(n)$ be the maximum time PARTIALDIGEST takes to find the solution for an n-point instance of the PDP. If there is only one viable alternative at every step, then PARTIALDIGEST steadily reduces the size of the problem by one and calls itself recursively, so

$$T(n) = T(n-1) + O(n),$$

where $O(n)$ is the work spent adjusting the sets X and L. However, if there are two alternatives, then

$$T(n) = 2T(n-1) + O(n).$$

While the expressions $T(n) = T(n-1) + O(n)$ and $T(n) = 2T(n-1) + O(n)$ bear a superficial similarity in form, they each lead to very different expressions for the algorithm's running time. One is quadratic, as we saw when analyzing SELECTIONSORT, and the other exponential, as we saw with HANOITOWERS. In fact, polynomial algorithms for the PDP were unknown until 2002 when Maurice Nivat and colleagues designed the first one.

4.4 Regulatory Motifs in DNA Sequences

Fruit flies, like humans, are susceptible to infections from bacteria and other pathogens. Although fruit flies do not have as sophisticated an immune system as humans do, they have a small set of *immunity genes* that are usually dormant in the fly genome, but somehow get switched on when the organism gets infected. When these genes are turned on, they produce proteins that destroy the pathogen, usually curing the infection.

One could design an experiment that is rather unpleasant to the flies, but very informative to biologists: infect flies with a bacterium, then grind up the flies and measure (perhaps with a DNA array) which genes are switched on

8. There exist pathological examples forcing the algorithm to explore *both* "left" and "right" alternatives at nearly every step.

as an immune response. From this set of genes, we would like to determine what triggers their activation. It turns out that many immunity genes in the fruit fly genome have strings that are reminiscent of TCGGGGATTTCC, located upstream of the genes' start. These short strings, called NF-κ*B binding sites*, are important examples of *regulatory motifs* that turn on immunity and other genes. Proteins known as *transcription factors* bind to these motifs, encouraging RNA polymerase to transcribe the downstream genes. Motif finding is the problem of discovering such motifs without any prior knowledge of how the motifs look.

Ideally, the fly infection experiment would result in a set of upstream regions from genes in the genome, each region containing at least one NF-κB binding site. Suppose we do not know what the NF-κB pattern looks like, nor do we know where it is located in the experimental sample. The fly infection experiment requires an algorithm that, given a set of sequences from a genome, can find short substrings that seem to occur surprisingly often.

"The Gold Bug", by Edgar Allan Poe, helps to illustrate the spirit, if not the mechanics, of finding motifs in DNA sequences. When the character William Legrand finds a parchment written by the pirate Captain Kidd, Legrand's friend says, "Were all the jewels of Golconda awaiting me upon my solution of this enigma, I am quite sure that I should be unable to earn them." Written on the parchment in question was

```
53++!305))6*;4826)4+.)4+);806*;48!8`60))85;]8*:+*8!
83(88)5*!;46(;88*96*?;8)*+(;485);5*!2:*+(;4956*2(5*
-4)8`8*; 4069285);)6!8)4++;1(+9;48081;8:8+1;48!85;4
 )485!528806*81(+9;48;(88;4(+?34;48)4+;161;:188;+?;
```

Mr. Legrand responds, "It may well be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve." He notices that a combination of three symbols—`; 4 8`—appears very frequently in the text. He also knows that Captain Kidd's pirates speak English and that the most frequent English word is "the." Proceeding under the assumption that `; 4 8` encodes "the," Mr. Legrand deciphers the parchment note and finds the pirate treasure. After making this substitution, Mr. Legrand has a slightly easier text to decipher:

```
53++!305))6*THE26)H+.)H+)TE06*THE!E`60))E5T]E*:+*E!
E3(EE)5*!TH6(TEE*96*?TE)*+(THE5)T5*!2:*+(TH956*2(5*
-H)E`E*T H0692E5)T)6!E)H++T1(+9THE0E1TE:E+1THE!E5TH
 )HE5!52EE06*E1(+9THET(EETH(+?3HTHE)H+T161T:1EET+?T
```

You might try to figure out what the symbol ")" might code for in order to complete the puzzle.

Unfortunately, DNA texts are not that easy to decipher, and there is little doubt that nature has constructed an enigma that human ingenuity cannot entirely solve. However, bioinformaticians borrowed Mr. Legrand's method, and a popular approach to motif finding is based on the assumption that frequent or rare words may correspond to regulatory motifs in DNA. It stands to reason that if a word occurs considerably more frequently than expected, then it is more likely to be some sort of "signal," and it is crucially important to figure out the biological meaning of the signal.

This "DNA linguistics" approach is at the heart of the *pattern-driven* approach to signal finding, which is based on enumerating all possible patterns and choosing the most frequent (or the most statistically surprising) among them.

4.5 Profiles

Figure 4.2 (a) presents seven 32-nucleotide DNA sequences generated randomly. Also shown [fig. 4.2 (b)] are the same sequences with the "secret" pattern P = ATGCAACT of length $l = 8$ implanted at random positions. Suppose you do not know what the pattern P is, or where in each sequence it has been implanted [fig. 4.2 (c)]. Can you reconstruct P by analyzing the DNA sequences?

We could simply count the number of times each l-mer, or string of length l, occurs in the sample. Since there are only $7 \cdot (32 + 8) = 280$ nucleotides in the sample, it is unlikely that any 8-mer other than the implanted pattern appears more than once.[9] After counting all 8-mer occurrences in figure 4.2 (c) we will observe that, although most 8-mers appear in the sample just once (with a few appearing twice), there is one 8-mer that appears in the sample suspiciously many times—seven or more. This overrepresented 8-mer is the pattern P we are trying to find.

Unlike our simple implanted patterns above, DNA uses a more inventive notion of regulatory motifs by allowing for mutations at some nucleotide positions [fig. 4.2 (d)]. For example, table 4.2 shows eighteen different NF-κB motifs; notice that, although none of them are the consensus binding site sequence TCGGGGATTTCC, each one is not substantially different. When the implanted pattern P is allowed to mutate, reconstructing P becomes more

9. The probability that any 8-mer appears in the sample is less than $280/4^8 \approx 0.004$

```
CGGGGCTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAACCAAAGCGGACAAA
GGGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCTC
CTGCTGTACAACTGAGATCATGCTGCTTCAAC
TACATGATCTTTTGTGGATGAGGGAATGATGC
```

(a) Seven random sequences.

```
CGGGGCTATGCAACTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAAATGCAACTCCAAAGCGGACAAA
GGATGCAACTGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGATGCAACTCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCATGCAACTTC
CTGCTGTACAACTGAGATCATGCTGCATGCAACTTTCAAC
TACATGATCTTTTGATGCAACTTGGATGAGGGAATGATGC
```

(b) The same DNA sequences with the implanted pattern ATGCAACT.

```
CGGGGCTATGCAACTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAAATGCAACTCCAAAGCGGACAAA
GGATGCAACTGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGATGCAACTCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCATGCAACTTC
CTGCTGTACAACTGAGATCATGCTGCATGCAACTTTCAAC
TACATGATCTTTTGATGCAACTTGGATGAGGGAATGATGC
```

(c) Same as (b), but hiding the implant locations. Suddenly this problem looks difficult to solve.

```
CGGGGCTATcCAgCTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAAggGCAACTCCAAAGCGGACAAA
GGATGgAtCTGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGAaGCAACcCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCtTGgAACTTC
CTGCTGTACAACTGAGATCATGCTGCATGCcAtTTTCAAC
TACATGATCTTTTGATGgcACTTGGATGAGGGAATGATGC
```

(d) Same as (b), but with the implanted pattern ATGCAACT randomly mutated in two positions; no two implanted instances are the same. If we hide the locations as in (c), the difficult problem becomes nearly impossible.

Figure 4.2 DNA sequences with implanted motifs.

Table 4.2 A small collection of putative NF-κB binding sites.

	T	C	G	G	G	G	A	T	T	T	C	A
	A	C	G	G	G	G	A	T	T	T	T	T
	T	C	G	G	T	A	C	T	T	T	A	C
	T	T	G	G	G	G	A	C	T	T	T	T
	C	C	G	G	T	G	A	T	T	C	C	C
	G	C	G	G	G	G	A	A	T	T	T	C
	T	C	G	G	G	G	A	T	T	C	C	T
	T	C	G	G	G	G	A	T	T	C	C	T
	T	A	G	G	G	G	A	A	C	T	A	C
	T	C	G	G	G	T	A	T	A	A	A	C
	T	C	G	G	G	G	G	T	T	T	T	T
	C	C	G	G	T	G	A	C	T	T	A	C
	C	C	A	G	G	G	A	C	T	C	C	C
	A	A	G	G	G	G	A	C	T	T	C	C
	T	T	G	G	G	G	A	C	T	T	T	T
	T	T	T	G	G	G	A	G	T	C	C	C
	T	C	G	G	T	G	A	T	T	T	C	C
	T	A	G	G	G	G	A	A	G	A	C	C
A:	2	3	1	0	0	1	16	3	1	2	4	1
T:	12	3	1	0	4	1	0	9	15	11	5	6
G:	1	0	16	18	14	16	1	1	1	0	0	0
C:	3	12	0	0	0	0	1	5	1	5	9	11
	T	C	G	G	G	G	A	T	T	T	C	C

complicated, since the 8-mer count does not reveal the pattern. In fact, the string ATGCAACT does not even appear in figure 4.2 (d), but the seven mutated versions of it appear at position 8 in the first sequence, position 19 in the second sequence, 3 in the third, 5 in the fourth, 31 in the fifth, 27 in the sixth, and 15 in the seventh.

In order to unambiguously formulate the motif finding problem, we need to define precisely what we mean by "motif." Relying on a single string to represent a motif often fails to represent the variation of the pattern in real biological sequences, as in figure 4.2 (d). A more flexible representation of a motif uses a profile matrix.

Consider a set of t DNA sequences, each of which has n nucleotides. Select one position in each of these t sequences, thus forming an array $\mathbf{s} =$

```
                       CGGGGCTATcCAgCTGGGTCGTCACATTCCCCTT...
            TTTGAGGGTGCCCAATAAggGCAACTCCAAAGCGGACAAA
                            GGATGgAtCTGATGCCGTTTGACGACCTA...
                          AAGGAaGCAACcCCAGGAGCGCCTTTGCTGG...
AATTTTCTAAAAAGATTATAATGTCGGTCCtTGgAACTTC
    CTGCTGTACAACTGAGATCATGCTGCATGCcAtTTTCAAC
                TACATGATCTTTTGATGgcACTTGGATGAGGGAATGATGC
```

(a) Superposition of the seven highlighted 8-mers from figure 4.2 (d).

		A	T	C	C	A	G	C	T
		G	G	G	C	A	A	C	T
		A	T	G	G	A	T	C	T
Alignment		A	A	G	C	A	A	C	C
		T	T	G	G	A	A	C	T
		A	T	G	C	C	A	T	T
		A	T	G	G	C	A	C	T
	A	5	1	0	0	5	5	0	0
Profile	**T**	1	5	0	0	0	1	1	6
	G	1	1	6	3	0	1	0	0
	C	0	0	1	4	2	0	6	1
Consensus		A	T	G	C	A	A	C	T

(b) The alignment matrix, profile matrix and consensus string formed from the 8-mers starting at positions $\mathbf{s} = (8, 19, 3, 5, 31, 27, 15)$ in figure 4.2 (d).

Figure 4.3 From DNA sample, to alignment matrix, to profile, and, finally, to consensus string. If $\mathbf{s} = (8, 19, 3, 5, 31, 27, 15)$ is an array of starting positions for 8-mers in figure 4.2 (d), then $Score(\mathbf{s}) = 5 + 5 + 6 + 4 + 5 + 5 + 6 + 6 = 42$.

$(s_1, s_2, \ldots, s_t)$, with $1 \le s_i \le n - l + 1$. The l-mers starting at these positions can be compiled into a $t \times l$ *alignment matrix* whose (i, j)th element is the nucleotide in the $s_i + j - 1$th element in the ith sequence (fig. 4.3). Based on the alignment matrix, we can compute the $4 \times l$ *profile matrix* whose (i, j)th element holds the number of times nucleotide i appears in column j of the alignment matrix, where i varies from 1 to 4. The profile matrix, or *profile*, illustrates the variability of nucleotide composition at each position for a particular choice of l-mers. For example, the positions 3, 7, and 8 are highly conserved, while position 4 is not. To further summarize the profile matrix, we can form a *consensus string* from the most popular element in each column of the alignment matrix, which is the nucleotide with the largest entry in the profile matrix. Figure 4.3 shows the alignment matrix for $\mathbf{s} = (8, 19, 3, 5, 31, 27, 15)$, the corresponding profile matrix, and the resulting consensus string ATGCAACT.

By varying the starting positions in s, we can construct a large number of different profile matrices from a given sample. We need some way of grading them against each other. Some profiles represent high conservation of a pattern while others represent no conservation at all. An imprecise formulation of the Motif Finding problem is to find the starting positions s corresponding to the most conserved profile. We now develop a specific measure of conservation, or strength, of a profile.

4.6 The Motif Finding Problem

If $\mathbf{P}(\mathbf{s})$ denotes the profile matrix corresponding to starting positions s, then we will use $M_{\mathbf{P}(\mathbf{s})}(j)$ to denote the largest count in column j of $\mathbf{P}(\mathbf{s})$. For the profile $\mathbf{P}(\mathbf{s})$ in figure 4.3, $M_{\mathbf{P}(\mathbf{s})}(1) = 5$, $M_{\mathbf{P}(\mathbf{s})}(2) = 5$, and $M_{\mathbf{P}(\mathbf{s})}(8) = 6$. Given starting positions $\mathbf{s}$, the *consensus score* is defined to be $Score(\mathbf{s}, DNA) = \sum_{j=1}^{l} M_{\mathbf{P}(\mathbf{s})}(j)$. For the starting positions in figure 4.3, $Score(\mathbf{s}, DNA) = 5 + 5 + 6 + 4 + 5 + 5 + 6 + 6 = 42$. $Score(\mathbf{s}, DNA)$ can be used to measure the strength of a profile corresponding to the starting positions s. A consensus score of $l \cdot t$ corresponds to the best possible alignment, in which each row of a column has the same letter. A consensus score of $\frac{lt}{4}$, however, corresponds to the worst possible alignment, which has an equal mix of all nucleotides in each column. In its simplest form, the Motif Finding problem can be formulated as selecting starting positions s from the sample that maximize $Score(\mathbf{s}, DNA)$.[10]

Motif Finding Problem:
Given a set of DNA sequences, find a set of l-mers, one from each sequence, that maximizes the consensus score.

Input: A $t \times n$ matrix of DNA, and l, the length of the pattern to find.

Output: An array of t starting positions $\mathbf{s} = (s_1, s_2, \ldots, s_t)$ maximizing $Score(\mathbf{s}, DNA)$.

Another view onto this problem is to reframe the Motif Finding problem as the problem of finding a *median string*. Given two l-mers v and w, we can compute the *Hamming distance* between them, $d_H(v, w)$, as the number of positions that differ in the two strings. For example, $d_H(\text{ATTGTC}, \text{ACTCTC}) = 2$:

```
A T T G T C
: x : x : :
A C T C T C
```

Now suppose that $\mathbf{s} = (s_1, s_2, \ldots, s_t)$ is an array of starting positions, and that v is some l-mer. We will abuse our notation a bit and use $d_H(v, \mathbf{s})$ to denote the *total Hamming distance* between v and the l-mers starting at positions $\mathbf{s}$: $d_H(v, \mathbf{s}) = \sum_{i=1}^{t} d_H(v, s_i)$, where $d_H(v, s_i)$ is the Hamming distance between v and the l-mer that starts at s_i in the ith DNA sequence. We will use $TotalDistance(v, DNA) = \min_{\mathbf{s}}(d_H(v, \mathbf{s}))$ to denote the minimum possible total Hamming distance between a given string v and any set of starting positions in the DNA. Finding $TotalDistance(v, DNA)$ is a simple problem: first one has to find the best match for v in the first DNA sequence (i.e., a position minimizing $d_H(v, s_1)$ for $1 \leq s_1 \leq n - l + 1$), then the best match in the

10. Another approach is to maximize the *entropy* of the corresponding profile. Let $\mathbf{P}(\mathbf{s}) = (p_{i,j})$, where $p_{i,j}$ is the count at element (i, j) of the $4 \times l$ profile matrix. Entropy is defined as

$$\sum_{j=1}^{l} \sum_{i=1}^{4} \frac{p_{i,j}}{t} \log \frac{p_{i,j}}{t}$$

where t is the number of sequences in the DNA sample. Although entropy is a more statistically adequate measure of profile strength than the consensus score, for the sake of simplicity we use the consensus score in the examples below.

A	**T**	C	**C**	**A**	G	**C**	**T**
G	G	**G**	**C**	**A**	**A**	**C**	**T**
A	**T**	**G**	G	**A**	T	**C**	**T**
A	A	**G**	**C**	**A**	**A**	**C**	C
T	**T**	**G**	G	**A**	**A**	**C**	**T**
A	**T**	**G**	**C**	C	**A**	T	**T**
A	**T**	**G**	G	C	**A**	**C**	**T**

Figure 4.4 Calculating the total Hamming distance for the consensus string ATG-CAACT (the alignment is the same as in figure 4.3). The bold letters show the consensus sequence; the total Hamming distance can be calculating as the number of nonbold letters.

second one, and so on. That is, the minimum is taken over all possible starting positions s. Finally, we define the *median string* for DNA as the string v that minimizes $TotalDistance(v, DNA)$; this minimization is performed over all 4^l strings v of length l.

We can formulate the problem of finding a median string in DNA sequences as follows.

Median String Problem:
Given a set of DNA sequences, find a median string.

> **Input:** A $t \times n$ matrix DNA, and l, the length of the pattern to find.
>
> **Output:** A string v of l nucleotides that minimizes $TotalDistance(v, DNA)$ over all strings of that length.

Notice that this is a double minimization: we are finding a string v that minimizes $TotalDistance(v, DNA)$, which is in turn the smallest distance among all choices of starting points s in the DNA sequences. That is, we are calculating

$$\min_{\substack{\text{all choices of}\\ l\text{-mers } v}} \quad \min_{\substack{\text{all choices of}\\ \text{starting positions } \mathbf{s}}} \quad d_H(v, \mathbf{s}).$$

Despite the fact that the Median String problem is a minimization problem and the Motif Finding problem is a maximization problem, the two problems are computationally equivalent. Let s be a set of starting positions with consensus score $Score(\mathbf{s}, DNA)$, and let w be the consensus string of the corresponding profile. Then

$$d_H(w, \mathbf{s}) = lt - Score(\mathbf{s}, DNA).$$

For example, in figure 4.4, the Hamming distance between the consensus string w and each of the seven implanted patterns is 2, and $d_H(w, \mathbf{s})$=$2 \cdot 7 = 7 \cdot 8 - 42$.

The consensus string minimizes $d_H(v, \mathbf{s})$ over all choices of v, i.e.,

$$d_H(w, \mathbf{s}) = \min_{\text{all choices of } v} d_H(v, \mathbf{s}) = lt - Score(\mathbf{s}, DNA)$$

Since t and l are constants, the smallest value of d_H can also be obtained by maximizing $Score(\mathbf{s}, DNA)$ over all choices of s:

$$\min_{\text{all choices of } \mathbf{s}} \min_{\text{all choices of } v} d_H(v, \mathbf{s}) = lt - \max_{\text{all choices of } \mathbf{s}} Score(\mathbf{s}, DNA).$$

The problem on the left is the Median String problem while the problem on the right is the Motif Finding problem.

In other words, the consensus string for the solution of the Motif Finding problem is the median string for the input DNA sample. The median string for DNA can be used to generate a profile that solves the Motif Finding problem, by searching in each of the t sequences for the substring with the smallest Hamming distance from the median string.

We introduce this formulation of the Median String problem to give more efficient alternative motif finding algorithms below.

4.7 Search Trees

In both the Median String problem and the Motif Finding problem we have to sift through a large number of alternatives to find the best one but we so far lack the algorithmic tools to do so. For example, in the Motif Finding problem we have to consider all $(n - l + 1)^t$ possible starting positions s:

$$\begin{array}{ccccccc}
(& 1, & 1, & \ldots, & 1, & 1 &) \\
(& 1, & 1, & \ldots, & 1, & 2 &) \\
(& 1, & 1, & \ldots, & 1, & 3 &) \\
 & & & \vdots & & & \\
(& 1, & 1, & \ldots, & 1, & n-l+1 &) \\
(& 1, & 1, & \ldots, & 2, & 1 &) \\
(& 1, & 1, & \ldots, & 2, & 2 &) \\
(& 1, & 1, & \ldots, & 2, & 3 &) \\
 & & & \vdots & & & \\
(& 1, & 1, & \ldots, & 2, & n-l+1 &) \\
 & & & \vdots & & & \\
(& n-l+1, & n-l+1, & \ldots, & n-l+1, & 1 &) \\
(& n-l+1, & n-l+1, & \ldots, & n-l+1, & 2 &) \\
(& n-l+1, & n-l+1, & \ldots, & n-l+1, & 3 &) \\
 & & & \vdots & & & \\
(& n-l+1, & n-l+1, & \ldots, & n-l+1, & n-l+1 &)
\end{array}$$

For the Median String problem we need to consider all 4^l possible l-mers:

AA··· AA
AA··· AT
AA··· AG
AA··· AC
AA··· TA
AA··· TT
AA··· TG
AA··· TC
⋮
CC··· GG
CC··· GC
CC··· CA
CC··· CT
CC··· CG
CC··· CC

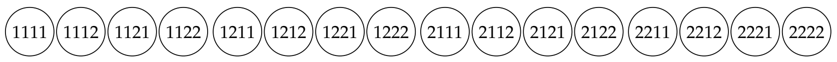


Figure 4.5 All 4-mers in the alphabet of $\{1, 2\}$.

We note that this latter progression is equivalent to the following one if we let 1 stand for A, 2 for T, 3 for G, and 4 for C:

$$\begin{gathered}
(1, 1, \ldots, 1, 1) \\
(1, 1, \ldots, 1, 2) \\
(1, 1, \ldots, 1, 3) \\
(1, 1, \ldots, 1, 4) \\
(1, 1, \ldots, 2, 1) \\
(1, 1, \ldots, 2, 2) \\
(1, 1, \ldots, 2, 3) \\
(1, 1, \ldots, 2, 4) \\
\vdots \\
(4, 4, \ldots, 3, 3) \\
(4, 4, \ldots, 3, 4) \\
(4, 4, \ldots, 4, 1) \\
(4, 4, \ldots, 4, 2) \\
(4, 4, \ldots, 4, 3) \\
(4, 4, \ldots, 4, 4)
\end{gathered}$$

In general, we want to consider all k^L L-mers in a k-letter alphabet. For the Motif Finding problem, $k = n - l + 1$, whereas for the Median String problem, $k = 4$. Figure 4.5 shows all 2^4 4-mers in the two-letter alphabet of 1 and 2. Given an L-mer from a k-letter alphabet, the subroutine NEXTLEAF (below) demonstrates how to jump from an L-mer $\mathbf{a} = (a_1 a_2 \cdots a_L)$ to the next L-mer in the progression. Exactly why this algorithm is called NEXTLEAF will become clear shortly.

```
NEXTLEAF($\mathbf{a}, L, k$)
1  for $i \leftarrow L$ to $1$
2      if $a_i < k$
3          $a_i \leftarrow a_i + 1$
4          return a
5      $a_i \leftarrow 1$
6  return a
```

NEXTLEAF operates in a way that is very similar to the natural process of counting. In most cases, $(a_1, a_2, \ldots, a_L)$ is followed by $(a_1, a_2, \ldots, a_L + 1)$. However, when $a_L = k$, the next invocation of NEXTLEAF will reset a_L to 1 and add 1 to a_{L-1}—compare this to the transition from 3719 to 3720 in counting. However, when there is a long string of the value k on the right-hand side of $\mathbf{a}$, the algorithm needs to reset them all to 1—compare this with the transition from 239999 to 240000. When all entries in $\mathbf{a}$ are k, the algorithm wraps around and returns $(1, 1, \ldots, 1)$, which is one way we can tell that we are finished examining L-mers. In the case that $L = 10$, NEXTLEAF is exactly like counting decimal numbers, except that we use "digits" from 1 to 10, rather than from 0 to 9.

The following algorithm, ALLLEAVES, simply uses NEXTLEAF to output all the 4-mers in the order shown in figure 4.5.

```
ALLLEAVES($L, k$)
1  $\mathbf{a} \leftarrow (1, \ldots, 1)$
2  while forever
3      output a
4      $\mathbf{a} \leftarrow$ NEXTLEAF($\mathbf{a}, L, k$)
5      if $\mathbf{a} = (1, 1, \ldots, 1)$
6          return
```

Even though line 2 of this algorithm seems as though it would loop forever, since NEXTLEAF will eventually loop around to $(1, 1, \ldots, 1)$, the **return** in line 6 will get reached and it will eventually stop.

Computer scientists often represent all L-mers as *leaves in a tree*, as in figure 4.6. L-mer trees will have L levels (excluding the topmost *root* level), and each vertex has k children. L-mers form leaves at the lowest level of the tree, $(L - 1)$-mers form the next level up, and $(L - 2)$-mers a level above that, and so on. For example, the vertices on the third level of the tree represent the eight different 3-mers: $(1, 1, 1)$, $(1, 1, 2)$, $(1, 2, 1)$, $(1, 2, 2)$, $(2, 1, 1)$, $(2, 1, 2)$,

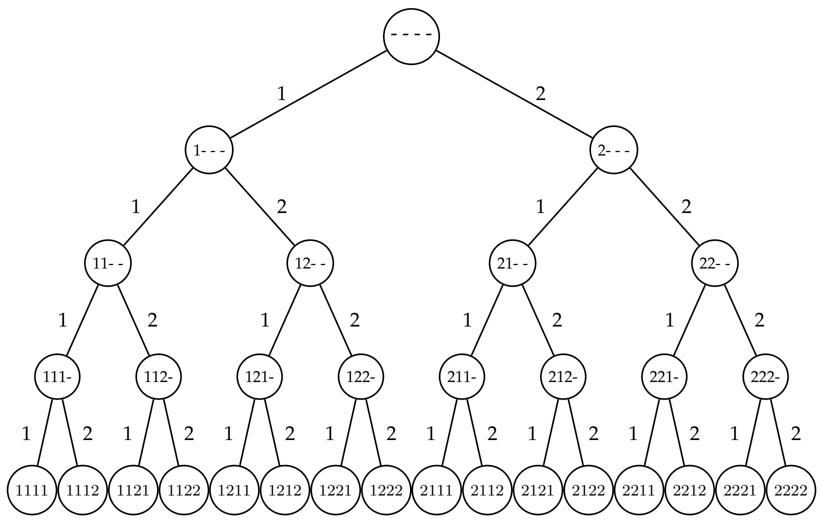


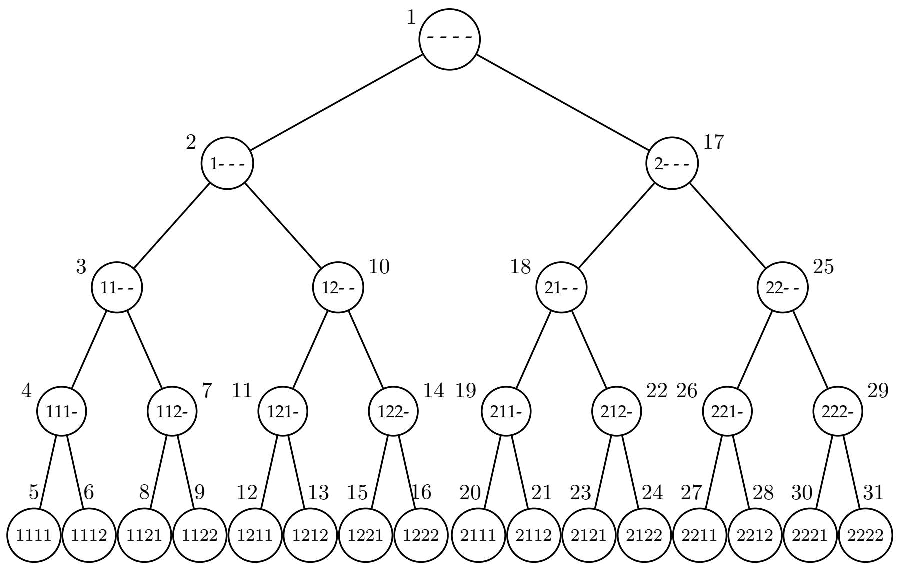
Figure 4.6 All 4-mers in the two-letter alphabet $\{1, 2\}$ can be represented as leaves in a tree.

$(2, 2, 1)$, and $(2, 2, 2)$.

The tree in figure 4.6 with $L = 4$ and $k = 2$ has 31 vertices.[11] Note that in a tree with L levels and k children per vertex, each leaf is equivalent to an array $\mathbf{a}$ of length L in which each element a_i takes on one of k different values. In turn, this is equivalent to an L-long string from an alphabet of size k, which is what we have been referring to as an L-mer. We will consider all of these representations to be equivalent. Internal vertices, on the other hand, can be represented as a pair of items: a list $\mathbf{a}$ of length L and an integer i that specifies the vertex's level. The entries $(a_1, a_2, \ldots, a_i)$ uniquely identify a vertex at level i; we will rely on the useful fact that the representation of an internal vertex is the *prefix* that is common to all of the leaves underneath it.

To represent all possible starting positions for the Motif Finding problem, we can construct the tree with $L = t$ levels[12] and $k = n - l + 1$ children per vertex. For the Median String problem, $L = l$ and $k = 4$. The astute reader may realize that the internal vertices of the tree are somewhat meaningless

11. In general, a tree with k children per vertex has k^i vertices at level i (every vertex at level $i-1$ gives birth to k children); the total number of vertices in the tree is then $\sum_{i=0}^{L} k^i$, while the number of leaves is only k^L.
12. As a reminder, t is the number of DNA sequences, n is the length of each one, and l is the length of the profile we would like to find.



```
PREORDER($v$)
1  output $v$
2  if $v$ has children
3      PREORDER( left child of $v$ )
4      PREORDER( right child of $v$ )
```

Figure 4.7 The order of traversing all vertices in a tree. The recursive algorithm PREORDER demonstrates how the vertices were numbered.

for the purposes of finding motifs, since they do not represent a sensible choice of starting positions in *all* of the t sequences. For this reason, we would like a method of scanning only the leaves of a tree and ignore the internal vertices. In doing this, it will probably appear that we have only complicated matters: we deliberately constructed a tree that contains internal vertices and now we will summarily ignore them. However, using the tree representation wlll allow us to use the branch-and-bound technique to improve upon brute force algorithms.

Listing the leaves of a tree is straightforward, but listing all vertices (i.e., all leaves and all internal vertices) is somewhat trickier. We begin at level 0 (the root) and then consider each of its k children in order. For each child, we again consider each of its k children and so on. Figure 4.7 shows the order of traversing vertices for a tree with $L = 4$ and $k = 2$ and also gives an elegant recursive algorithm to perform this process. The sequence of vertices that PREORDER($root$) would return on the tree of $L = 4$ and $k = 2$ would be as follows:

```
(-,-,-,-)
(1,-,-,-)
(1,1,-,-)
(1,1,1,-)
(1,1,1,1)
(1,1,1,2)
(1,1,2,-)
(1,1,2,1)
(1,1,2,2)
(1,2,-,-)
(1,2,1,-)
(1,2,1,1)
(1,2,1,2)
(1,2,2,-)
(1,2,2,1)
(1,2,2,2)
(2,-,-,-)
(2,1,-,-)
(2,1,1,-)
(2,1,1,1)
(2,1,1,2)
(2,1,2,-)
(2,1,2,1)
(2,1,2,2)
(2,2,-,-)
(2,2,1,-)
(2,2,1,1)
(2,2,1,2)
(2,2,2,-)
(2,2,2,1)
(2,2,2,2)
```

Traversing the complete tree iteratively is implemented in the NEXTVERTEX algorithm, below. NEXTVERTEX takes vertex $\mathbf{a} = (a_1, \ldots, a_L)$ at level i as an input and returns the next vertex in the tree. In reality, at level i NEXTVERTEX only uses the values $a_1, \ldots, a_i$ and ignores $a_{i+1}, \ldots, a_L$. NEXTVERTEX takes inputs that are similar to NEXTLEAF, with the exception that the "current leaf" is now the "current vertex," so it uses the parameter i for the level on which the vertex lies. Given **a**, L, i, and k, NEXTVERTEX returns the next vertex in the tree as the pairing of an array and a level. The algorithm will return a level number of 0 when the traversal is complete.

```
NEXTVERTEX($\mathbf{a}, i, L, k$)
1 if $i < L$
2     $a_{i+1} \leftarrow 1$
3     return $(\mathbf{a}, i + 1)$
4 else
5     for $j \leftarrow L$ to $1$
6         if $a_j < k$
7             $a_j \leftarrow a_j + 1$
8             return $(\mathbf{a}, j)$
9 return $(\mathbf{a}, 0)$
```

When $i < L$, NEXTVERTEX($\mathbf{a}, i, L, k$) moves down to the next lower level and explores that subtree of **a**. If $i = L$, NEXTVERTEX either moves along the lowest level as long as $a_L < k$ or jumps back up in the tree.

We alluded above to using this tree representation to help reduce work in brute force search algorithms. The general branch-and-bound approach will allow us to ignore any children (or grandchildren, great-grandchildren, and so on) of a vertex if we can show that they are all uninteresting. If none of the descendents of a vertex could possibly have a better score than the best leaf that has already been explored, then there really is no point descending into the children of that vertex. At each vertex we calculate a bound–the most wildly optimistic score of any leaves in the subtree rooted at that vertex—and then decide whether or not to consider its children. In fact, the strategy is named branch-and-bound for exactly this reason: at each point we calculate a bound and then decide whether or not to branch out further (figure 4.8).

Branching-and-bounding requires that we can skip an entire subtree rooted at an arbitrary vertex. The subroutine NEXTVERTEX is not up to this task, but the algorithm BYPASS, below, allows us to skip the subtree rooted at vertex $(\mathbf{a}, i)$. If we skip a vertex at level i of the tree, we can just increment a_i (unless $a_i = k$, in which case we need to jump up in the tree). The algorithm BYPASS takes the same type of input and produces the same type of output as NEXTLEAF.

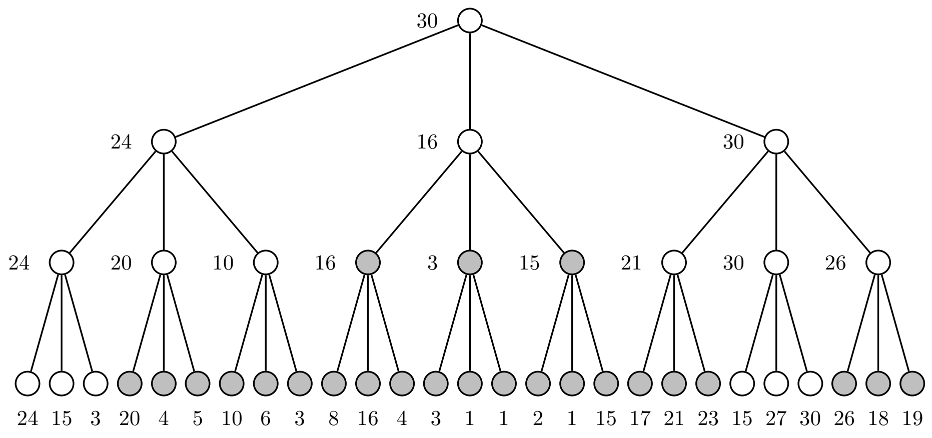


Figure 4.8 A tree that has uninteresting subtrees. The numbers next to a leaf represent the "score" for that L-mer. Scores at internal vertices represent the maximum score in the subtree rooted at that vertex. To improve the brute force algorithm, we want to "prune" (ignore) subtrees that do not contain high-scoring leaves. For example, since the score of the very first leaf is 24, it does not make sense to analyze the 4th, 5th, or 6th leaves whose scores are 20, 4, and 5, respectively. Therefore, the subtree containing these vertices can be ignored.

```
BYPASS($\mathbf{a}, i, L, k$)
1  for $j \leftarrow i$ to $1$
2      if $a_j < k$
3          $a_j \leftarrow a_j + 1$
4          return $(\mathbf{a}, j)$
5  return $(\mathbf{a}, 0)$
```

We pause to remark that the iterative version of tree navigation that we present here is equivalent to the standard recursive approach that would be found in an introductory algorithms text for computer scientists. Rather than rob you of this discovery, the problems at the end of this chapter explore this relationship in more detail. Simply transforming the list of alternatives that need to be searched into a search tree makes many brute force algorithms blatantly obvious; even better, a sensible branch-and-bound strategy will often become clear.

4.8 Finding Motifs

The brute force approach to solve the Motif Finding problem looks through all possible starting positions.

```
BRUTEFORCEMOTIFSEARCH($DNA, t, n, l$)
1  $bestScore \leftarrow 0$
2  for each $(s_1, \ldots, s_t)$ from $(1, \ldots, 1)$ to $(n - l + 1, \ldots, n - l + 1)$
3      if $Score(\mathbf{s}, DNA) > bestScore$
4          $bestScore \leftarrow Score(\mathbf{s}, DNA)$
5          bestMotif $\leftarrow (s_1, s_2, \ldots, s_t)$
6  return bestMotif
```

There are $n-l+1$ choices for the first index (s_1) and for each of those, there are $n-l+1$ choices for the second index (s_2). For each of those choices, there are $n - l + 1$ choices for the third index, and so on. Therefore, the overall number of positions is $(n - l + 1)^t$, which is exponential in t, the number of sequences. For each s, the algorithm calculates $Score(\text{s}, DNA)$, which requires $O(l)$ operations. Thus, the overall complexity of the algorithm is evaluated as $O(ln^t)$.

The only remaining question is how to write line 2 using standard pseudocode operations. This is particularly easy if we use NEXTLEAF from the previous section. In this case, $L = n - l + 1$ and $k = t$. Rewriting BRUTEFORCEMOTIFSEARCH in this way we arrive at BRUTEFORCEMOTIFSEARCHAGAIN.

```
BRUTEFORCEMOTIFSEARCHAGAIN($DNA, t, n, l$)
1  $\mathbf{s} \leftarrow (1, 1, \ldots, 1)$
2  $bestScore \leftarrow Score(\mathbf{s}, DNA)$
3  while forever
4      $\mathbf{s} \leftarrow$ NEXTLEAF($\mathbf{s}, t, n - l + 1$)
5      if $Score(\mathbf{s}, DNA) > bestScore$
6          $bestScore \leftarrow Score(\mathbf{s}, DNA)$
7          bestMotif $\leftarrow (s_1, s_2, \ldots, s_t)$
8      if $s = (1, 1, \ldots, 1)$
9          return bestMotif
```

Finally, to prepare for the branch-and-bound strategy, we will want the equivalent version, SIMPLEMOTIFSEARCH, which uses NEXTVERTEX to explore each leaf.

```
SimpleMotifSearch($DNA, t, n, l$)
1  $\mathbf{s} \leftarrow (1, \ldots, 1)$
2  $bestScore \leftarrow 0$
3  $i \leftarrow 1$
4  while $i > 0$
5      if $i < t$
6          $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, t, n - l + 1$)
7      else
8          if $Score(\mathbf{s}, DNA) > bestScore$
9              $bestScore \leftarrow Score(\mathbf{s}, DNA)$
10             bestMotif $\leftarrow (s_1, s_2, \ldots, s_t)$
11         $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, t, n - l + 1$)
12 return bestMotif
```

Observe that some sets of starting positions can be ruled out immediately without iterating over them, based simply on the most optimistic estimate of their score. For example, if the first i of t starting positions [i.e., $(s_1, s_2, \ldots, s_i)$] form a "weak" profile, then it may not be necessary to even consider any starting positions in the sequences $i + 1, i + 2, \ldots, t$, since the resulting profile could not possibly be better than the highest-scoring profile already found.

Given a set of starting positions $\mathbf{s} = (s_1, s_2, \ldots, s_t)$, define the *partial consensus score*, $Score(\mathbf{s}, i, DNA)$, to be the consensus score of the $i \times l$ alignment matrix involving only the first i rows of DNA corresponding to starting positions $(s_1, s_2, \ldots, s_i, -, -, \ldots, -)$. In this case, a $-$ indicates that we have not chosen any value for that entry in s. If we have the partial consensus score for $s_1, \ldots, s_i$, even in the best of circumstances the remaining $t - i$ rows can only improve the consensus score by $(t - i) \cdot l$; therefore, the score of any alignment matrix with the first i starting positions $(s_1, \ldots, s_i)$ could be at most $Score(\mathbf{s}, i, DNA) + (t - i) \cdot l$. This implies that if $Score(\mathbf{s}, i, DNA) + (t - i) \cdot l$ is less than the currently best score, $bestScore$, then it does not make sense to explore any of the remaining $t - i$ sequences in the sample—with this choice of $(s_1, \ldots, s_i)$—it would obviously result in wasted effort. Therefore, the bound $Score(\mathbf{s}, i, DNA) + (t - i) \cdot l$ could save us the trouble of looking at $(n - l + 1)^{t-i}$ leaves.

```
BranchAndBoundMotifSearch($DNA, t, n, l$)
1  $\mathbf{s} \leftarrow (1, \ldots, 1)$
2  $bestScore \leftarrow 0$
3  $i \leftarrow 1$
4  while $i > 0$
5      if $i < t$
6          $optimisticScore \leftarrow Score(\mathbf{s}, i, DNA) + (t - i) \cdot l$
7          if $optimisticScore < bestScore$
8              $(\mathbf{s}, i) \leftarrow$ Bypass($\mathbf{s}, i, t, n - l + 1$)
9          else
10             $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, t, n - l + 1$)
11     else
12         if $Score(\mathbf{s}, DNA) > bestScore$
13             $bestScore \leftarrow Score(\mathbf{s})$
14             bestMotif $\leftarrow (s_1, s_2, \ldots, s_t)$
15         $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, t, n - l + 1$)
16 return bestMotif
```

Though this branch-and-bound strategy improves our algorithm for some problem instances, we have not improved the worst-case efficiency: you can design a sample with an implanted pattern that requires exponential time to find.

4.9 Finding a Median String

We mentioned above that the Median String problem gives us an alternate approach to finding motifs. If we apply the brute force technique to solve this problem, we arrive at the following algorithm[13]:

13. The parameters t, n, and l in this algorithm are needed to compute the value of TotalDistance($word, DNA$). A more detailed pseudocode would use TotalDistance($word, DNA, t, n, l$) but we omit these details for brevity.

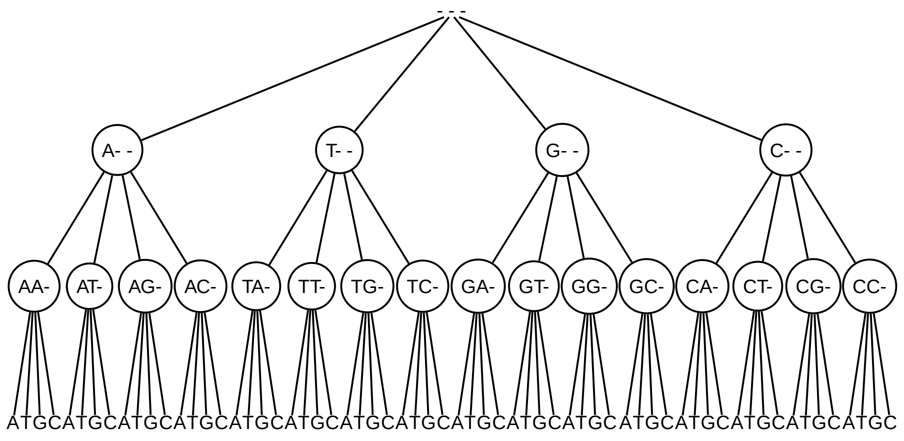


Figure 4.9 A search tree for the Median String problem. Each branching point can give rise to only four children, as opposed to the $n-l+1$ children in the Motif Finding problem.

```
BRUTEFORCEMEDIANSEARCH($DNA, t, n, l$)
1  $bestWord \leftarrow$ AAA···AA
2  $bestDistance \leftarrow \infty$
3  for each $l$-mer $word$ from AAA...A to TTT...T
4      if TOTALDISTANCE($word, DNA$) $< bestDistance$
5          $bestDistance \leftarrow$ TOTALDISTANCE($word, DNA$)
6          $bestWord \leftarrow word$
7  return $bestWord$
```

BRUTEFORCEMEDIANSEARCH considers each of 4^l nucleotide strings of length l and computes TOTALDISTANCE at every step. Given $word$, we can calculate $TotalDistance(word, DNA)$ in a single pass over DNA (i.e., in $O(nt)$ time), rather than by considering all possible starting points in the DNA sample. Therefore, BRUTEFORCEMEDIANSEARCH has running time $O(4^l \cdot n \cdot t)$, which compares favorably with the $O(ln^t)$ of SIMPLEMOTIFSEARCH. A typical motif has a length (l) ranging from eight to fifteen nucleotides, while the typical size of upstream regions that are analyzed have length (n) ranging from 500 to 1000 nucleotides. BRUTEFORCEMEDIANSEARCH is a practical algorithm for finding short motifs while SIMPLEMOTIFSEARCH is not.

We will proceed along similar lines to construct a branch-and-bound strategy for BRUTEFORCEMEDIANSTRING as we did in the transformation of

BRUTEFORCEMOTIFSEARCH into SIMPLEMOTIFSEARCH. We can modify the median string search to explore the entire tree of all l-nucleotide strings (see figure 4.9) rather than only the leaves of that tree as in BRUTEFORCEMEDIANSEARCH. A vertex at level i in this tree represents a nucleotide string of length i, which can be viewed as the i-long prefix of every leaf below that vertex. SIMPLEMEDIANSEARCH assumes that nucleotides A, C, G, T are coded as numerals (1, 2, 3, 4); for example, the assignment in line 1 sets s to the l-mer $(1, 1, \ldots, 1)$, corresponding to the nucleotide string AA . . . A.

```
SIMPLEMEDIANSEARCH($DNA, t, n, l$)
 1  $\mathbf{s} \leftarrow (1, 1, \ldots, 1)$
 2  $bestDistance \leftarrow \infty$
 3  $i \leftarrow 1$
 4  while $i > 0$
 5      if $i < l$
 6          $(\mathbf{s}, i) \leftarrow$ NEXTVERTEX($\mathbf{s}, i, l, 4$)
 7      else
 8          $word \leftarrow$ nucleotide string corresponding to $(s_1, s_2, \ldots s_l)$
 9          if TOTALDISTANCE($word, DNA$) $< bestDistance$
10              $bestDistance \leftarrow$ TOTALDISTANCE($word, DNA$)
11              $bestWord \leftarrow word$
12          $(\mathbf{s}, i) \leftarrow$ NEXTVERTEX($\mathbf{s}, i, l, 4$)
13  return $bestWord$
```

In accordance with the branch-and-bound strategy, we find a bound for $TotalDistance(word, DNA)$ at each vertex. It should be clear that if the total distance between the i-prefix of $word$ and DNA is larger than the smallest seen so far for one of the leaves (nucleotide strings of length l), then there is no point investigating subtrees of the vertex corresponding to that i-prefix of $word$; all extensions of this prefix into an l-mer will have at least the same total distance and probably more. This is what forms our branch-and-bound strategy. The bound in BRANCHANDBOUNDMEDIANSEARCH relies on the optimistic scenario that there *could* be some extension to the prefix that matches every string in the sample, which would add 0 to the total distance calculation.

```
BranchAndBoundMedianSearch($DNA, t, n, l$)
 1  $\mathbf{s} \leftarrow (1, 1, \dots, 1)$
 2  $bestDistance \leftarrow \infty$
 3  $i \leftarrow 1$
 4  while $i > 0$
 5      if $i < l$
 6          $prefix \leftarrow$ nucleotide string corresponding to $(s_1, s_2, \dots, s_i)$
 7          $optimisticDistance \leftarrow$ TotalDistance($prefix, DNA$)
 8          if $optimisticDistance > bestDistance$
 9              $(\mathbf{s}, i) \leftarrow$ Bypass($\mathbf{s}, i, l, 4$)
10          else
11              $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, l, 4$)
12      else
13          $word \leftarrow$ nucleotide string corresponding to $(s_1, s_2, \dots s_l)$
14          if TotalDistance($word, DNA$) $< bestDistance$
15              $bestDistance \leftarrow$ TotalDistance($word, DNA$)
16              $bestWord \leftarrow word$
17          $(\mathbf{s}, i) \leftarrow$ NextVertex($\mathbf{s}, i, l, 4$)
18  return $bestWord$
```

The naive bound in BranchAndBoundMedianSearch is overly generous, and it is possible to design more aggressive bounds (this is left as a problem at the end of the chapter). As usual with branch-and-bound algorithms, BranchAndBoundMedianSearch provides no improvement in the worst-case running time but often results in a practical speedup.

4.10 Notes

In 1965, Werner Arber (32) discovered restriction enzymes, conjecturing that they cut DNA at positions where specific nucleotide patterns occur. In 1970, Hamilton Smith (95) verified Arber's hypothesis by showing that the *Hin*dII restriction enzyme cuts DNA in the middle of palindromes GTGCAC or GTTAAC. Other restriction enzymes have similar properties, but cut DNA at different patterns. Dan Nathans pioneered the application of restriction enzymes to genetics and constructed the first ever restriction map in 1973 (26). All three were awarded the Nobel Prize in 1978.

The PartialDigest algorithm for the construction of restriction maps was proposed by Steven Skiena and colleagues in 1990 (94). In 1994 Zheng Zhang (114) came up with a "difficult" instance of PDP that requires an exponential time to solve using the PARTIALDIGEST algorithm. In 2002, Maurice Nivat and colleagues described a polynomial algorithm to solve the PDP (30).

Studies of gene regulation were pioneered by François Jacob and Jacques Monod in the 1950s. They identified genes (namely, regulatory genes) whose proteins (transcription factors) have as their sole function the regulation of other genes. Twenty years later it was shown that these transcription factors bind specifically in the upstream areas of the genes they regulate and recognize certain patterns (motifs) in DNA. It was later discovered that, in some cases, transcription factors may bind at a distance and regulate a gene from very far (tens of thousands of nucleotides) away.

Computational approaches to motif finding were pioneered by Michael Waterman (89), Gary Stormo (46) and their colleagues in the mid-1980s. Profiles were introduced by Gary Stormo and colleagues in 1982 (101) and further developed by Michael Gribskov and colleagues in 1987 (43). Although the naive exhaustive motif search described in this chapter is too slow, there exist fast and practical branch-and-bound approaches to motif finding (see Marsan and Sagot, 2000 (72), Eskin and Pevzner, 2002 (35)).