MODULE 09-2
Finding Repeats Within Strings

Date Prepared: November, 2009

Dina Sokol[1]
Department of Computer and Information Science
Brooklyn College of the City University of NY
Brooklyn, NY 11210
sokol@sci.brooklyn.cuny.edu

Frederick Adkins
Mathematics Department
Indiana University of Pennsylvania
Indiana, Pennsylvania 15705
fadkins@iup.edu

Zhongyuan Che
Department of Mathematics
Penn State University, Beaver Campus
Monaca, PA 15061
zxc10@psu.edu

Kristin Pfabe
Department of Mathematics and Computer Science
Nebraska Wesleyan University
Lincoln, NE 68504
kpfabe@nebrwesleyan.edu

[1] This work has been supported in part by the *National Science Foundation* Grant DB&I 0542751.

Module Description Information

- **Title:**
  F*ind*i*ng* Repeats With*in* Str*in*gs

- **Authors:**

  1. Dina Sokol, Brooklyn College of the City University of NY, Brooklyn, NY 11210, sokol@sci.brooklyn.cuny.edu
  2. Frederick Adkins, Indiana University of Pennsylvania, Indiana, PA 15705, fadkins@iup.edu
  3. Zhongyuan Che, Pennsylvania State University Beaver Campus, Monaca, PA 15061, zxc10@psu.edu
  4. Kristin Pfabe, Nebraska Wesleyan University, Lincoln, NE 68504, kpfabe@nebrwesleyan.edu

- **Abstract:**

  Genomic sequences often contain copies of patterns called *repeats.* Repeats occurring in the genome are important genetic markers for disease diagnosis and mapping studies, as well as for human identity testing. This module presents several algorithms for finding repeats within biological sequences. Both *tandem* repeats, i.e. repeats in which copies are contiguous, and non-tandem repeats are discussed. Dynamic programming is described and a modification of the Smith-Waterman algorithm is shown for finding non-tandem repeats with errors. Algorithms are presented in pseudocode and illustrated with examples, including carefully diagrammed matrices. Each algorithm is analyzed for its asymptotic time complexity, motivating the selection of more efficient techniques. Several exercises and suggestions for additional explorations are given. Finally, programs in C++ or Java are included for the algorithms presented, and are available for running on the web at: http://tandem.sci.brooklyn.cuny.edu/SWrepeats.

- **Informal Description:**
  This module provides an introduction to several computational genomic techniques within the framework of finding repeats within a sequence. It is designed for students who are not necessarily experts in either biology or computer science. In Section 1 we describe some basic background in biology and the importance of repeats found in biological sequences. In Section 2 we present two algorithms for finding tandem

repeats within a sequence. Asymptotic time complexity is explained for the benefit of those who are unfamiliar with "big Oh" notation. In Section 3 we discuss more general repeats, allowing non-tandem repeats that include insertions, deletions, and mismatches. We describe how dynamic programming is used to generate a 2-sequence alignment using the Smith-Waterman algorithm. We then show how this can be modified to find all repeats occurring in a sequence. Section 5 contains additional exercises and Section 6 contains supplementary material including the solutions to all of the exercises. C++ or java code is included for each algorithm in the appendix.

- **Target Audience:**
  Undergraduate students at the sophomore level or above in a computer science, mathematics, or biology department.

- **Prerequisites:**
  An introductory computer science class and familiarity with counting techniques is required. In a computational biology course, this material would follow nicely after coverage of the Smith-Waterman method for local alignment of two strings. However, this presentation is self-contained and requires no specific background in computational biology.

- **Mathematical Field:**
  Computational Biology, Discrete Mathematics, Computer Science

- **Application Areas:**
  Sequence analysis of biological sequences such as DNA, RNA and protein sequences

- **Mathematics Subject Classification:**
  MSC (2010): 68W32, 68Q25, 92D20, 62P10

- **Contact Information:**
  Dina Sokol, Brooklyn College of the City University of NY, Brooklyn, NY 11210, sokol@sci.brooklyn.cuny.edu

- **Other DIMACS modules related to this module:**
  None

Table of Contents

1 Introduction

1.1 Biological background and context

Deoxyribonucleic acid (DNA) resides in the nucleus of the cell of an organism. DNA molecules encode the genetic information necessary for the function of the cell and they control the inheritance of characteristics. Analysis of DNA can describe the relationship between species and provide insight into inherited disease and progression of other diseases.

The DNA is made up of a long, double-stranded helix. Each strand of the helix consists of nucleotide bases of four types: adenine (A), cytosine (C), guanine (G), and thymine (T). These four bases work together in complementary pairs across the double helix. Adenine pairs with thymine and guanine with cytosine. To investigate the genetic makeup of a DNA molecule, it is necessary to look at one strand of the double helix, since the other strand is the complementary sequence of the first strand. That is, if one strand begins ACTGA…, the other strand of the double helix starts TGACT… .

All of the DNA that an organism possesses is called the organism's *genome*. The 3 billion bp (base pairs) in the human genome are organized into 24 distinct, physically separate microscopic units called *chromosomes*. The chromosomes contain the *genes*, the basic units of heredity. The sequence of bases in the genes, also called the *genetic code*, is a blueprint for the synthesis of protein molecules. Proteins are involved in almost every biological function, and are sometimes called the "molecules of life." For a primer on molecular biology and protein synthesis, see:
http://www.ornl.gov/sci/techresources/Human_Genome/publicat/primer/toc.html.

Analysis of the sequence of bases in DNA provides many insights into the biological functions of a cell. In this module we discuss one specific aspect of sequence analysis, namely the search for repeated sequences in a DNA sequence.

1.2 Repeats in DNA sequences

A pattern of nucleotides that occurs more than once in a sequence is called a *repeat*. Most genomes have a high content of repetitive DNA. More than 50% of the human genome is made up of repeats [HGP01]. Some relatively simple organisms like *Amoeba dubia* have large areas of repeated sequences resulting in a much longer genome than other more complex organisms such as *Homo sapiens* [R04]. On each chromosome of *A. thaliana,*[2] a small flowering plant, there is a region that is highly repeated in tandem (i.e. contiguous) and this repetition accounts for 2-5% of its entire genome [LLDA03].

Repetitions in DNA arise and grow through molecular events that copy and insert DNA segments in either dispersed or tandem sites. Repetitive DNA sequences are one of the principle origins of genomic instability because recombination between similar sequences can cause chromosomal rearrangements [LLDA03]. Repeats are important genetic markers for disease diagnosis [SR95] and mapping studies, as well as for human identity testing, sequence homology, and population studies. Repeats are found in both coding and non-coding regions of DNA. Expansions of repeats found in the protein-coding portions of genes can affect the function of the gene by causing synthesis of malfunctioning proteins. Repeats in non-coding regions have been shown to affect biological processes by affecting gene expression, transcription and translation.

1.3 Overview

Section 2 discusses techniques for finding *tandem* (i.e. contiguous) repeats in a sequence. A modification of the Smith-Waterman method for sequence alignment is used to find more general repeats in Section 3. The material in these two sections provides an introduction to computational genomic techniques; it is designed for students who are not necessarily experts in either biology or computer science, but have taken at least an introductory course in computer science. Algorithms are clearly presented, with carefully diagrammed matrices illustrating the computational steps on sample inputs. The

[2] The genome of *A. thaliana* is one of the smallest plant genomes and was the first plant genome to be sequenced. Thus it has been used for understanding the molecular biology of many plant traits.

algorithms are analyzed for computational complexity, motivating the search for more efficient techniques.

2 Tandem Repeats in Strings

2.1 Introduction

There are many types of repeated strings of biological interest. Repeats that occur at contiguous locations are called *tandem repeats* and they have special significance for biologists. Tandem repeats in human DNA are responsible for over 30 inherited diseases, including fragile X syndrome, myotonic dystrophy, Huntington's disease, various spinocerebellar ataxias, Friedreich's ataxia, and others [C92, GZ05, M08]. In a normal gene, there is a stable threshold for the copy number of a repeat, while in individuals in which the copy number exceeds the threshold the disease is manifested. For instance, in a normal FMR-1 gene, the triplet CGG is tandemly repeated 6 to 54 times, while in patients with fragile X syndrome, the pattern occurs more than 200 times.

The tandem repeats in the human genome are the genetic markers used in DNA forensics [J93a, J93b]. Since the number of adjacent copies varies from individual to individual, the copy number of a tandem repeat can be used to identify an individual, and relations such as parent or grandparent. Tandem repeats are also used in population studies [UW93], conservation biology [SH02], and in conjunction with multiple sequence alignments [B97, K96].

To begin the discussion of repeats, the problem is constrained to the special case of finding tandem repeats with exactly two copies. In order to narrow the initial focus of the search for repeats even further, only repeats that are exact, i.e., ones with no errors in the copies of the repeats, will be considered. Formally, an exact tandem repeat is defined as a nonempty string that can be divided into two identical substrings; i.e., if v is a nonempty string, then $r = vv$ is a tandem repeat. The *size* of a tandem repeat r, denoted by $|r|$, is the number of characters in it, and its *period* is $|v|$. Note that the size of a tandem repeat is two times its period.

For example, the string GTCAACAATC has three tandem repeats: AA, AA and CAACAA. The size of AA is 2, with period 1, and the size of CAACAA is 6, with period 3.

In this section, we'll try to find tandem repeats in strings.

> **Exercise 2.1**: Find all tandem repeats in GTTGTTGTTGTT and list the size and period of each.

> **Exercise 2.2**: A counting problem:
> (a) How many tandem repeats are in the string AAAAAA?
> (b) How many tandem repeats are in the string AAAAAAA?

> **Exercise 2.3**: How can we redefine tandem repeats in a way that we are left with fewer repeats (for example, in Exercise 2.2, one repeat would be listed) that succinctly represent all repeats within a string?

2.2 Brute force search algorithms

Problem: Given a string *S* of length *n*, find all exact tandem repeats that are substrings of *S*.

2.2.1 Algorithm 1

Consider every even length substring of the given string *S*. Partition each substring of even length into two equal-length substrings. Compare the substrings to see if they match.

> **Example**: Consider AGCGCTT.
> The even length substrings are AG, GC, CG, GC, CT, TT, AGCG, GCGC, CGCT, GCTT, AGCGCT and GCGCTT. GCGC is a tandem repeat because GC and GC match. AGCGCT is not a tandem repeat because AGC and GCT do not match. The only tandem repeats are TT and GCGC.

Complexity Analysis

Suppose that $f(n)$ is the computing time (loosely speaking, the computing time is the number of steps to perform a task) to run the algorithm on a string of length *n*. We would like to express, in a simple manner, the asymptotic size of $f(n)$ for large n. For this, we introduce "Big Oh" notation.

Definition: We say that $f(n) = O(g(n))$ if there exist positive constants c and k such that $0 \le f(n) \le cg(n)$ for all $n \ge k$. This is read "f is big Oh of g".

Example:

Show that $f(n) = n^2 + 6n + 2 = O(n^2)$. Since $n^2 + 6n + 2 \le n^2 + 6n^2 + 2n^2 = 9n^2$ for $n \ge 1$, $f(n) = O(n^2)$. Here, our constants are $c = 9$ and $k = 1$. Notice that we did not provide the tightest bound. In fact, one can also show that $f(n) \le 2n^2$ for $n \ge 7$. So it's not important which constants c and k you find. What *is* important is the existence of such constants. We have shown that $f(n) = n^2 + 6n + 2 = O(n^2)$. In fact, it is true that $f(n) = O(n^p)$ for any $p \ge 2$.

Exercise 2.4: Suppose that $f(n) = O(n^2)$ and $g(n) = O(n^3)$. Show that $(f+g)(n) = O(n^3)$.

Many functions that model computing time are polynomials, and if f is a polynomial of degree p, then $f(n) = O(n^p)$. For example, $f(n) = 4n^5 + 6n - 3 = O(n^5)$ and $g(n) = 6 = O(1)$.

Additional exercises:

Exercise 2.5: Show that $n! = O(n^n)$.

Exercise 2.6: True or False: If $f(n) = O(g(n))$, then $g(n) = O(f(n))$. (If you answer True, you must provide a proof. If you answer False, you must provide a counterexample.)

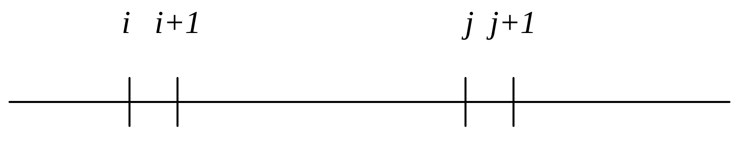
Basic Counting Principles

For our analysis of the computing time of algorithms, it will be helpful to understand some counting principles. How many ways are there to arrange 2 items from a collection of n items? There are n possibilities for selecting the first and $n-1$ for the second. Thus, there are $n(n-1)$ ways to arrange these items. Now, suppose you don't care about how they are arranged and you just want to know how many groups of 2 items from a collection of n items you can form. As an example, ab and ba are different arrangements of the letters a and b, but they are the same sets of letters. Thus $n(n-1)$ is too big by a factor of 2, and therefore the number of groups of 2 items from a collection of n items is $n(n-1)/2$. These groups of unordered items are called *combinations*. The notation $C(n,2)$ denotes that number of combinations of n items taken 2 at a time. An alternate notation is $\binom{n}{2}$. We have discovered that $\binom{n}{2} = \frac{n(n-1)}{2}$.

Getting back to finding all exact tandem repeats that are substrings of a string of length n, let's find our algorithm's order. First we need to determine the number of substrings of even length. Each substring is determined by a left and right endpoint. The total number of substrings of length greater than 1 is $n(n-1)/2 = O(n^2)$. The total number of even length substrings is about half of this, $n(n-1)/4 = O(n^2)$. For each even length substring, you must divide it in half and compare the two halves. This step takes no more than $n/2 = O(n)$ character comparisons since the longest half to compare has length $\frac{n}{2}$.

Thus the total computing time for this algorithm on a string of length n is approximately $\frac{n(n-1)}{4} \cdot \frac{n}{2} = O(n^3)$.

In this method, there are many repeated comparisons of the same pairs of characters. Consider the substring (*i,j*) and the substring (i*+1,j+1),* both of which have the same size. This motivates us to search for an algorithm that takes less time.



In the next exercise, we'll apply our notion of combinations to the counting problem mentioned in Exercise 2.2.

Exercise 2.7: A counting problem, revisited:

(a) Compute the number of tandem repeats in the string AAAAAA using combinations. Hint: alternately place two characters, say / and \$ between each A, and at the beginning and end of the string. This gives us /A\$A/A\$A/A\$A/. Consider how choosing two of the same character relates to tandem repeats.

(b) Apply your technique in part (a) to compute the number of tandem repeats in the string AAAAAAA?

(c) Let A^n denote the string of n consecutive A's. How many tandem repeats are in the string A^n?

2.2.2 Algorithm 2

If we compare the string with shifted copies of itself, we can identify tandem repeats in less time. To find all repeats of size $2i$, line up the original sequence with the sequence shifted by i characters to the right. Check the alignment for matching substrings of length i.

Example: Suppose we want to find tandem repeats in the string ABCECEFFG. (Note that this is not a genomic sequence.) To find all tandem repeats of size 2, shift the full sequence by one to see where there are matches. In the table below, we find that "FF" is a tandem repeat of size 2.

A	B	C	E	C	E	F	F	G		
	A	B	C	E	C	E	F	F	G	

To find all tandem repeats of length 4, shift the full sequence by 2. In the table, one finds that "CECE" is a repeated sequence.

A	B	C	E	C	E	F	F	G		
		A	B	C	E	C	E	F	F	G

Complexity Analysis:
First note that a tandem repeat can have size no greater than n, so there are no more than $n/2$ shifts. For each shift there are no more than n comparisons. So the computing time of the algorithm is approximately $\frac{n}{2} \cdot n = O(n^2)$.

Notice that in this algorithm we are matching the same string with itself at every possible shift. Information from previous shifts can be used to speed up the computations at each new shift using standard pattern matching techniques. This yields much more efficient algorithms. The best known algorithm for finding exact tandem repeats (with any number of copies) has linear time ($O(n)$ time) and was developed by Kolpakov and Kucherov [KK99].

Exercise 2.8: A *multiple tandem repeat* is a tandem repeat that contains two or more copies. A string *r* is a *multiple* tandem repeat of period |*v*| if it can be partitioned into *n+1* substrings, the first *n* substrings being identical, say *v*, and the last substring being a (possibly empty) prefix of *v*. Formally, $r = v^n v', n \geq 2$, v' is a (possibly empty) prefix of *v*. For example, ACTACTACTAC is a multiple tandem repeat with period 3, and size 11. Can you develop a brute force method for finding multiple tandem repeats? Show how your algorithm works using the input string ATATATATGC.

Exercise 2.9: The *Hamming distance* between two strings of equal length is defined as the number of mismatching characters between the two strings. For example, the Hamming distance between AGCTA and ACCTT is 2. Propose an algorithm for finding *approximate tandem repeats*, i.e. repeats in which there are two “copies” of the repeat, but mismatches between the “copies” are allowed. Specifically, find all approximate tandem repeats $r = v\bar{v}$, where $|v| = |\bar{v}|$, and the Hamming distance between *v* and $\bar{v}$ is less than *k*, for some given integer *k*.

2.2.3 Pseudocode for Algorithms 1 and 2

The input and output for Algorithms 1 and 2 are as follows.

Input: A string *S* of length *n*. (We assume that the characters are numbered *1…n.*)

Output: All tandem repeats in *S*.

Algorithm 1

```
len=2;
while len <= n do        // len stands for the length of the substring being checked
                         // for tandem repeats
    begin
        p=len/2;         // p stands for the period of the repeat
        for (i=1 to n-len+1) do
            begin
                // compare contiguous substrings of length p, one starting at position i,
                // the other starting at position i+p
                if  (S[i…i+(p–1)]==S[i+p…i+(len–1)])
                    report repeat of size len beginning at location i.
            end  // for i
        len=len+2;
    end  // while
```

Algorithm 2

```
match=0;
for p = 1 to ⌊n/2⌋ do   // p stands for the period of the repeat
  begin
    for i=1 to n-p  do       // for each i, check whether a repeat ends at location i+p
      begin
        if (S[i] == S[i+p])
          begin
```

```
                $match$=$match$+1;   // count how many consecutive characters match
                if ($match$ >= $p$)
                    report a repeat of length $2p$ beginning at location $i - p+1$.
            end   // if S[i]==S[i+p]

        else                // this character does not match, begin again
            $match$=0;

    end // for i
end // for p
```

3 More General Repeats

3.1 Properties of general repeats

One reason for comparing DNA sequences is to find evidence that they come from a common ancestor. Through the mutation process, insertions, deletions and transcription errors change the sequence.

We will now classify more general repeats, having one or more of the following properties:

1. Copies of a repeat may be non-contiguous (i.e. non-tandem). (e.g. **AB**FGAB)
2. Copies of a repeat may be overlapping. (e.g. in **ABCAB**CAB, **ABCAB** is overlapped with **AB**CAB)
3. Copies of a repeat may contain errors, such as mismatches, insertions and deletions. (e.g. **GTCA**GTCC with a mismatched character or **GTBC**GTC with a missing character) Usually the number of errors allowed is much smaller than the length of the repeat.

A general repeat may fall into more than one of these categories. For example, in the string AGCTCTGAA, the substring CTC overlaps CTG, and this repeat has errors.

Exercise 3.1: Find twenty (or more) general repeats in GCGAGAGACGCC.

Remark: Although we are discussing *general* repeats, we are still only allowing exactly two parts to each repeat. *Multiple* repeats are even more general, allowing several copies within a given repeat. More discussion of multiple (tandem) repeats is included with the exercises of Section 2.

3.2 Scoring repeats

Consider the string BEERBEAR, which has more than one general repeat. Observe these two repeats: **BE**ERBEAR and **BEER**BEAR. Which one is "better"? In order to answer this, one must introduce a scoring function, which will calculate a *score* for a given repeat. We can then compare the scores of two different repeats, and the one with the higher score will be considered better. We introduce the following scoring function: when comparing a repeat character by character, a character match increases the score by 1 and a mismatch decreases the score by 2. With the repeats seen in **BE**ERBEAR, we match the substrings **BE** and BE to get a score of 2. For **BEER**BEAR, we match **BEER** and BEAR to get a score of 1. With the given scoring scheme, BE is the better repeat of the two given.

> **Exercise 3.2**: Use the scoring function in which a match increases the score by 2 and a mismatch decreases it by 1 to rescore the two general repeats **BE**ERBEAR and **BEER**BEAR.

Due to chemical properties of DNA bases, mismatches between certain pairs of bases are more likely to occur. Specifically, an A can easily be replaced with a G, and a C can easily be replaced with a T. In text processing, an assumption of all mismatches being equally likely is also nonrealistic. For example, it is more likely to mistakenly replace a vowel with another vowel. Thus, we would want to score vowel mismatches with less of a penalty than consonant mismatches. This would require a *weighted* scoring scheme, in which the score depends upon the particular characters that mismatch.

> **Exercise 3.3**: Modify the prior scoring scheme to one where each exact match increases the score by 2, each mismatch of two vowels decreases it by 1/2 and each mismatch of two consonants or consonant with vowel decreases it by 2. Use this scheme to rescore the two general repeats **BE**ERBEAR and **BEER**BEAR.

A scoring function can also be viewed as a matrix, in which each location contains the score between the character on top and the character on the left. The following matrix represents a scoring function for the four DNA bases. Note that the penalty (or negative score) given for a mismatch between the purines (A and G) and pyrimidines (T and C), is less than the penalty for mismatches between a purine and a pyrimidine.

	A	C	G	T
A	2	-1	1	-1
C	-1	2	-1	1
G	1	-1	2	-1
T	-1	1	-1	2

Exercise 3.4: Use the above matrix to compute the score for the following two repeats: **AACGT**<u>AACCA</u> and **AACGT**<u>GGCGT.</u> Explain why a different score was achieved for these two repeats, even though they both have 3 matching characters.

The discussion up until now included only matching and mismatching characters. As mentioned in property 3, Section 3.1, a general repeat may also contain errors in the form of deletion or insertion of bases. In the alignment of the copies of a repeat, an insertion/deletion is represented by a "gap" character (-). To illustrate this idea, we will align FEAR and FAR by inserting a gap in FAR.

FEAR
F–AR

In order to score this alignment, we need to introduce a scoring scheme that accounts for gaps. We use the scoring function in which a match increases the score by 1, and a mismatch or gap decreases the score by 2. Obviously, we do not allow a gap in one string to align with a gap in the other string. The score of the FEAR and F–AR alignment is 1.

Henceforth, we denote the *scoring function* by *s(a,b).* For each application, *s(a,b)* must be given, and it must assign a value, or "score," to each possible pair in an alignment, including all pairs of characters, as well as each character against a gap. To obtain the score of a 2-sequence alignment, we sum over the scores of all pairs in the alignment.

The scoring function may differ for different applications. In the above example the scoring function used was: *s(a,a)=+1, s(a,b)=s(a,-)=s(-,a)=-2 (for all characters a,b in the alphabet,* $a \neq b$ *).*

> **Exercise 3.5**: Use the scoring function: *s(a,a)=+1, s(a,-)=s(-,a)=-2, s(a,b)=-1 (for all characters a,b in the alphabet,* $a \neq b$ *).* For the two sequences SOME and SCORE, find the alignment that produces the greatest score. In other words, you will find an optimal 2-sequence alignment.

In Section 3.4 we include a more advanced scoring scheme which allows gaps and is weighted according to the purines and pyrimidines. For this section, we use a simplified scoring function, to ease the exposition of the algorithms. The simplified scoring function allows gaps but does not weigh for purines and pyrimidines.

3.3 Algorithm for finding General Repeats

The algorithm for finding general repeats is a direct modification of the Smith-Waterman (SW) algorithm for the local alignment of two sequences [W95]. Since understanding the SW algorithm is critical for understanding the algorithm for general repeats, we briefly describe the SW algorithm. Classes that covered the SW algorithm as part of the topic of sequence alignment may omit Section 3.3.1.

3.3.1 Dynamic Programming and 2-Sequence Alignment

Dynamic programming is a method of solving a problem, in which solutions to subproblems are stored in a *table*. Solutions to larger subproblems are calculated from previous values in the table. A very simple example of dynamic programming, in one dimension, would be to compute the *n*th Fibonacci number by completing an array of size *n+1* (say Fib[*0...n*]). Initially, Fib[0] is set to 0, and Fib[1] is set to 1. Each entry, Fib[*i*] for *i*≥2, is computed from the two previous entries, by the formula: Fib[*i*] = Fib[*i-1*] + Fib[*i-2*]. In this section we describe a dynamic programming algorithm in two dimensions in which an optimal 2-sequence alignment is obtained for two given sequences.

A *2-sequence alignment* is a way of arranging two sequences of DNA or proteins to identify similarities between the sequences. Typically, a scoring function assigns a score to each pair of residues (which are represented as characters in the computational model). A residue matching with a gap (-) is assigned a penalty score, and the score of the alignment is defined as the sum of the pairwise scores. The goal is to align the two sequences in a way that maximizes the score of the alignment. A *global* sequence alignment optimizes the alignment of the two entire sequences. A *local* sequence alignment attempts to optimally align *subsequences* of the input sequences.

For example, given a simple scoring function of *s(a,a)=1* for all characters *a*, and –1 otherwise, the following is the global and local alignment of the two sequences `TGGTATGCC` and `TTATCCG.`

Global Sequence Alignment:

TGGTATGCC-
T--TAT-CCG
score=2

Local Sequence alignment (shown in Bold characters):

TGG**TATGCC**
  T**TAT-CC**G
score=4

Given two sequences, *P* of length *n* and *Q* of length *m,* over some alphabet, a scoring function *s(a,b)* over all symbols in the alphabet, and a gap penalty *s(a,-)* for each symbol, the Smith-Waterman [SW81] algorithm computes the optimal local alignment between *P* and *Q* in *O(nm)* time. The algorithm uses dynamic programming to compute an *n x m* matrix, *M*, as follows. Initially, *M[i,0]=0* and *M[0,j]=0*. The rest of the entries in *M* are computed row-by-row, from left to right using the following formula. The goal is to maximize the value of each cell of *M*.

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + s(p_i, q_j) \\ M[i-1, j] + s(p_i, -) \\ M[i, j-1] + s(-, q_j) \\ 0 \end{cases}$$

Following, we show the matrix M for the two strings and the scoring function considered in the above example. Let P=`TGGTATGCC` and Q=`TTATCCG.` The string P is placed on the left of the matrix, labeling the rows, and Q is placed on top, labeling the columns of the matrix. (We use *i* to index P and *j* to index Q.)

INDEX	*j*	0	1	2	3	4	5	6	7
i		Q=	T	T	A	T	C	C	G
0	P=	0	0	0	0	0	0	0	0
1	T	0	1	1	0	1	0	0	0
2	G	0	0	0	0	0	0	0	1
3	G	0	0	0	0	0	0	0	1
4	T	0	1	1	0	1	0	0	0
5	A	0	0	0	2	0	0	0	0
6	T	0	1	1	1	**3**	2	1	0
7	G	0	0	0	0	2	2	1	2
8	C	0	0	0	0	1	3	3	2
9	C	0	0	0	0	0	2	4	3

After the initialization of the first row and column to zeros, each *M[i,j]* is determined by looking at 3 neighboring cells: the previous one on the diagonal, *M[i-1,j-1]*, the one above, *M[i-1,j]*, and the one to the left, *M[i,j-1]*. We summarize the meaning of each of these actions in terms of the alignment:

(1) Come from the "diagonal:" Align the character at position *i* of P with the character at position *j* of Q.

(2) Come from "above:" Insert a gap in Q (i.e. align – with p_i).

(3) Come from the "left:" Insert a gap in P (i.e. align – with q_j).

(4) If all three possibilities are negative, then *M[i,j]* is reset to 0, allowing a new start

of a local alignment.

For example, consider location M[6,4] in the above matrix. The maximum comes from adding s(T,T)=1 to the value 2 on the diagonal. The value in M[6,5] is calculated by subtracting 1 from M[6,4].

In the Smith-Waterman algorithm, the goal is to maximize the value of each cell of M. Upon completion, the maximum value in M corresponds to the optimal local alignment. In fact, due to the optimal substructure property of dynamic programming, the value of each entry $M[i,j]$ represents the score of the optimal alignment of a subsequence of P ending at p_i and a subsequence of Q ending at q_j . By tracing back the path that resulted in the computation of the maximum value, the optimal local alignment of the two strings can be obtained. Since M[9,6]=4 is the largest value in the matrix in our example, the optimal local alignment ends at p_9 and q_6 , and it is as follows.

Optimal Local Sequence alignment of P and Q:

$p_4 \ldots p_9$:	**`TATGCC`**
$q_2 \ldots q_6$:	**`TAT-CC`**

Note that there are many other possible alignments ending at p_9 and q_6, however, M[9,6] gives the highest score of all such alignments.

Since this module concentrates on finding repeats within sequences, we describe the trace-back method in detail in the next section in the context of repeats.

Note: The Smith-Waterman algorithm for local alignments is a modification of the Needleman and Wunsch [NW70] algorithm used for global alignments. The key differences in the algorithms are in the initialization of the first row and column, and in the inclusion of zero in the definition of M[i,j]. We refer the reader to an excellent text by Jones and Pevzner [JP04] for more detail.

3.3.2 Using Dynamic Programming to find Repeats

To motivate the use of dynamic programming, let us first think about a brute force method that would attempt to locate general repeats. One could take each subsequence of the sequence and try to align it with every other subsequence, looking for alignments with large scores. This is an extremely expensive method of finding repeats. In fact, since a local alignment of all possible pairs of substrings is necessary, this would yield a $O(n^6)$ time algorithm. (There are $O(n^2)$ substrings, all possible pairs of substrings gives $O(n^4)$, and each alignment allowing gaps costs $O(n^2)$.) However, it turns out that a modification of the SW method for local alignment can be used exactly *one time* to locate all repeats within a string. Since we are filling in an $n \times n$ matrix, this method has time and space $O(n^2)$.

The idea is to align the sequence with a copy of *itself*, and compute the best possible local alignment ending at every possible point. A matrix is built by placing copies of the given string to the left and above as shown for AACTAAT in Figure 1.

The modifications to the dynamic programming matrix for finding repeats are:
(1) place the *string S* both on the top and to the left of the matrix to align the string with itself,
(2) set all elements on the diagonal of the matrix to 0 to avoid aligning characters with themselves,
(3) compute only the upper triangular matrix ($M[i,j]=M[j,i]$ since both strings are identical).

Specifically, given a string of length n, we compute the *upper-right portion of the n x n* matrix M using Algorithm 3, given in pseudocode in the next subsection. We sometimes refer to the string on the left of the matrix as the "left" string, and the string on the top of the matrix as the "top" string. Of course, both of these strings are identical.

3.3.3 Algorithm 3 in pseudocode – Compute the dynamic programming matrix

Input: A string $S = x_1,\ldots,x_n$, and a scoring function $s(a,b)$ defined on all characters in the alphabet, and a gap penalty $s(a,-),s(-,a)$ defined over all characters in the alphabet.
Output: The upper right part of an $n \times n$ matrix M, containing the scores of the local alignments of the string S with itself.

Algorithm 3 - BuildMatrix

```
for i=0 to n
  begin
    M[0,i]=0;   // initialize first row
    M[i,i]=0;   // initialize diagonal
  end
for i=1 to n
    for j=i+1 to n
        $M[i,j] = \max\{M[i-1,j-1]+s(x_i,x_j),\ M[i-1,j]+s(x_i,-),\ M[i,j-1]+s(-,x_j),\ 0\}$
```

3.3.4 Illustration of Algorithm 3

The matrix in Figure 1 is the result of running Algorithm 3 on the input string AACTAAT. The following scoring function was used:

$$s(a,b) = \begin{cases} 1 \text{ if } a = b \\ -1 \text{ otherwise} \end{cases} \qquad s(a,-) = s(-,a) = -1$$

where a and b are any of the characters in the alphabet and – represents a gap.

Note that only the upper-triangular portion of the matrix is filled in. It's important to place 0's on the diagonal to avoid matching the string with itself, a trivial repeat.

		0	1	2	3	4	5	6	7
			A	**A**	**C**	**T**	**A**	**A**	**T**
0		0	0	0	0	0	0	0	0
1	**A**		0	1	0	0	1	1	0
2	**A**			0	0	0	1	2	1
3	**C**				0	0	0	1	1
4	**T**					0	0	0	2
5	**A**						0	1	1
6	**A**							0	0
7	**T**								0

Figure 1 – Matrix M aligning the sequence with itself.

Figure 2 describes how to fill cell (4,7). Suppose the values in cells (3,6), (3,7), and (4,6) are already known as shown in Figure 1. If we are computing the value in cell (4,7) we must consider the possible contributions marked by the arrows to the cumulative scores in the other cells. The vertical arrow contributes -1 to the score from (3,7) as it reflects insertion of a gap in the top string (specifically, following the T at position 7). The horizontal arrow contributes -1 to the score from (4,6) as it reflects insertion of a gap in the left string (following the T at position 4). The diagonal arrow reflects matching T (at position 4 in left string) with T (at position 7 in top string). It contributes +1 to the score from (3,6) and yields the maximum of 2 when calculating $M(4,7)$.

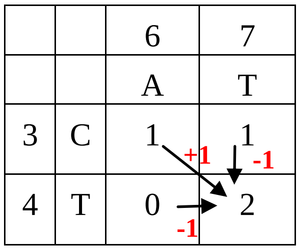


Figure 2 – Calculating the score for cell *M[4,7]*.

3.3.5 Trace-back for finding repeats

Once the matrix is filled, it is necessary to retrieve the actual repeats. Recall that the *(i,j)* entry in the matrix *M* is the score of the best local alignment(s) ending at the pairing (x_i, x_j). (Remember, there could be gaps following x_i or x_j, but not both.) Thus, the largest entries in the matrix represent the highest scoring repeats. We locate the largest element in the matrix to retrieve its repeat. If there is more than one entry with the max value, we arbitrarily choose one of these entries. To retrieve the alignment of the repeat, we will use what is called the "trace-back" method. The traceback method traces the path that resulted in the computation of the maximum value. The traceback always ends at a location in the matrix with the value 0.

In our matrix, we see that the largest scores are in the (2,6) and (4,7) positions. Let's focus on the 2 in the (4,7) position and determine where this came from. The 2 came

from an entry in one of the three positions, (3,6) (from the diagonal entry), (3,7) (from above) or (4,6) (from the left). We can rule out both (3,7) and (4,6) because a cell to the left or above (4,7) can only decrease the score of (4,7), and the entry in (4,7) is larger than those in (3,7) and (4,6). So the entry in position (4,7) came from (3,6). Let's reconstruct the alignment of the repeats from right to left, one character at a time.
In the following diagrams, TopIndex refers to the index into the top string, and TopString refers to characters from the top string. The same is true for LeftIndex and LeftString. The rightmost pair in the alignment is:

TopIndex				7
TopString				**T**
LeftString				**T**
LeftIndex				4

The next step is to determine where the entry in position (3,6) came from. It came from either (2,5), (2,6) or (3,5). It could not have come from (3,5) because moving horizontally decreases the score because of the gap penalty. It could not have come from the (2,5) entry because we had a mismatch and so if we had gone diagonally, this would have produced a 0 in the (3,6) position. Hence, location (3,6) was calculated from above, location (2,6). A vertical move corresponds to inserting a gap in the string on the top (since we are "consuming" a character on the left, but not on the top). This gives the second to last pair in the local alignment as follows.

TopIndex				7
TopString			–	**T**
LeftString			**C**	**T**
LeftIndex			3	4

We continue in this manner, resulting in the following – alignment. The process terminates when a 0 is reached, in this case at location (0,4).

TopIndex	5	6		7
TopString	**A**	**A**	–	**T**
LeftString	**A**	**A**	**C**	**T**
LeftIndex	1	2	3	4

Figure 3 shows the traceback path shaded in the matrix M.

Let's check our work by scoring our repeat:

#matches – #mismatches – #gaps $= 3 - 0 - 1 = 2$.

		0	1	2	3	4	5	6	7
			A	**A**	**C**	**T**	**A**	**A**	**T**
0		0	0	0	0	0	0	0	0
1	**A**		0	1	0	0	1	1	0
2	**A**			0	0	0	1	2	1
3	**C**				0	0	0	1	1
4	**T**					0	0	0	2
5	**A**						0	1	1
6	**A**							0	0
7	**T**								0

Figure 3: This is the matrix M shown in Figure 1 with the traceback path of the optimal local alignment shaded.

Comment: If you change the scoring function, you may get a different "best" repeat alignment.

We have found a best scoring repeat. Notice that the one found happened to be a tandem repeat. Suppose now that you want to find other interesting repeats. The obvious thing to do would be to get the next largest value in the matrix (or another occurrence of the max), and trace it back, as we did above. However, we do not want to report any part of the repeat that was already found. For example, we don't want to report the repeated strings AA in positions 1, 2 and AA in positions 5,6. In other words, we want to restrict matching parts of the found repeat from being included in subsequent repeats. Therefore, we cannot simply take the next largest number in the matrix (or possibly another occurrence of the max) and trace it back because that might pick up portions of the alignment of the first repeat that was already found.

The solution is to adjust the values in the matrix that were affected by the matching characters in the found repeat. Specifically, we adjust all values on the path of the found repeat, and all values that were computed from a value on the path. This process of adjusting the entries must start at the beginning of the path. Note that the path producing the highest scoring repeat consists of the cells in order: (0,4), (1,5), (2,6), (3,6), (4,7). That is, the reverse of the traceback arrows in Figure 5.

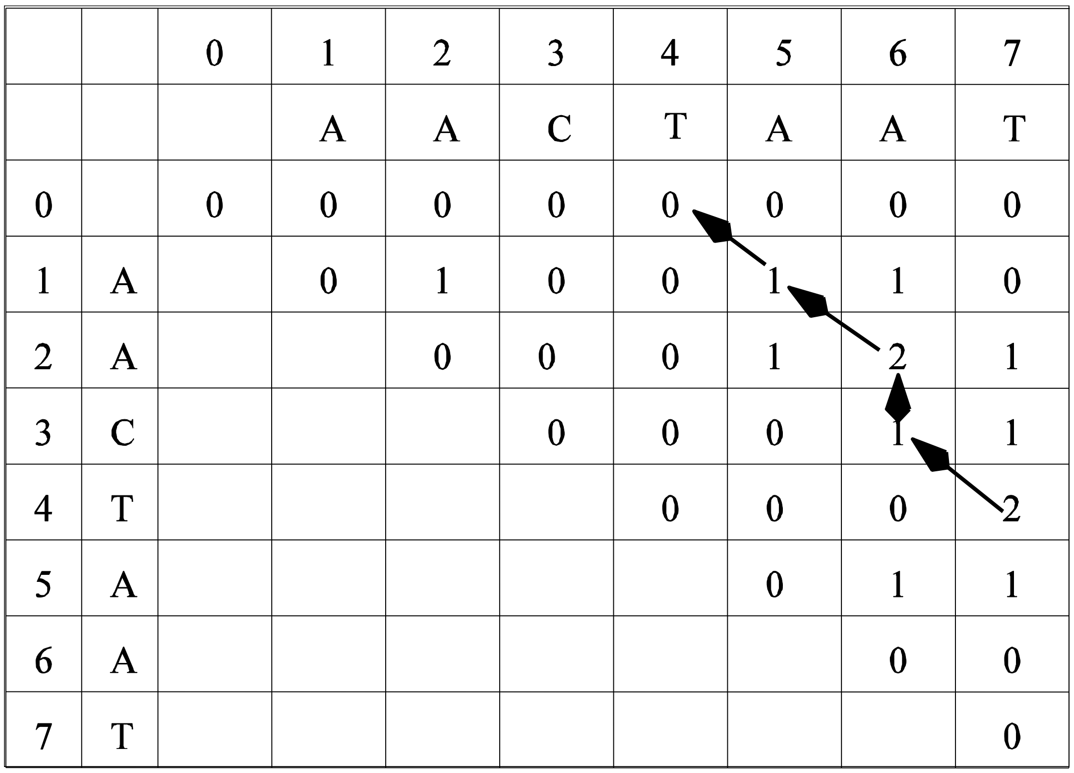
The adjustment to the matrix is done in two phases. In phase 1, we shade the cells that will need to be recomputed. In Figure 4, the region that must be adjusted is shaded. The cells that have dark shading are on the actual path of the repeat (excluding the first cell with the zero value). The lightly shaded cells are those affected by the path of the highest scoring repeat, i.e. their values have been computed from either a value on the path, or another lightly shaded cell. In case of a tie, where a value can be computed from a non-shaded cell or from a shaded cell, it is not necessary to shade the cell. For example, consider the 0 in cell (4,6). It could come from the 1 above in darkly shaded cell (3,6) (on path) or from the zero in the formula M[4,6] = {M[3,6]-1, M[4,5]-1, M[3,5]-1, 0}. Hence, we do not shade cell (4,6).

The procedure to perform the shading begins with the second cell of the path and works row-by-row from left to right. Each row begins under the previous row's leftmost shaded cell (light or dark). Dark shading is done to every cell on the path. A cell is lightly shaded if its score is forced from a light or dark shaded cell. If its score can be gotten from an unshaded cell then we do not shade it, since lowering the entries in the path cells will not affect this cell.

		0	1	2	3	4	5	6	7
			A	**A**	**C**	**T**	**A**	**A**	**T**
0		0	0	0	0	0	0	0	0
1	**A**		0	1	0	0	1	1	0
2	**A**			0	0	0	1	2	1
3	**C**				0	0	0	1	1
4	**T**					0	0	0	2
5	**A**						0	1	1
6	**A**							0	0
7	**T**								0

Figure 4: All shaded cells must be adjusted in order to proceed with the next best repeat. The darkly shaded cells represent the path of the optimal alignment, and the lightly shaded cells are those cells that are affected by the path.

Remark: However, in our code, it is sometimes simpler to shade extra cells, rather than to check for several possibilities. Although this is unnecessary, it does no harm, since when the cell is recomputed, it will get its same value.



		0	1	2	3	4	5	6	7
			A	A	C	T	A	A	T
0		0	0	0	0	0	0	0	0
1	A		0	1	0	0	1	1	0
2	A			0	0	0	1	2	1
3	C				0	0	0	1	1
4	T					0	0	0	2
5	A						0	1	1
6	A							0	0
7	T								0

Figure 5 – Traceback Arrows on Matrix M for Finding Repeats

Phase 2 of the adjustment consists of recalculating all shaded cells. Once again, the calculations are done row-by-row, from the left to right. For each shaded cell, we simply recompute the maximum as described in Section 3.3.2, excluding the entry that is on the path.

Referring to Figure 4, we start with the (1,5) entry, which is the second cell in our path. The 1 came from the match (on the diagonal), and so recompute disallowing that match but otherwise following the scoring scheme. We take the maximum of {*M[0,5]-1, M[1,4]-1, 0*}. The (1,5) entry becomes 0.

Now consider the entry in the (1,6) position. Since the 1 came from the match (on the diagonal), and the entry (0,5) is not on the path nor affected by the path (unshaded in Figure 1), the (1,6) entry does not get changed. We proceed to the next row.

In row 2 we begin with the first entry beneath a path entry in row 1, that is, the (2,5) entry. It came from the match and entry (1,4) is unshaded in Figure 1, so entry (2,5) doesn't change. Now consider (2,6). Its entry came from a match in the original high scoring string, so this contribution is not considered in the maximization. Maximizing over the other possibilities gives us a 0. For the (2,7) entry, we don't consider the score moving horizontally. Maximizing over the other possibilities gives 0. Continuing in this fashion, we adjust the rest of the affected area. Figure 6 shows the adjusted matrix.

		0	1	2	3	4	5	6	7
			A	**A**	**C**	**T**	**A**	**A**	**T**
0		0	0	0	0	0	0	0	0
1	**A**		0	1	0	0	0	1	0
2	**A**			0	0	0	1	0	0
3	**C**				0	0	0	0	0
4	**T**					0	0	0	0
5	**A**						0	1	0
6	**A**							0	0
7	**T**								0

Figure 6 – Modified Matrix for Finding Different Repeats

The maximum score on our adjusted matrix is a 1, which corresponds to pairing A's together. The four cells containing the value of 1 correspond to four different repeats:

2	5	6	6
A	A	A	A
A	A	A	A
1	2	1	5

Note that we do not get the repeats

5	6	7
A	A	T
A	A	T
1	2	4

which are part of the original highest scoring repeat

	5	6		7
	A	**A**	–	**T**
	A	**A**	**C**	**T**
	1	2	3	4

In general, you would repeat this process until your maximum scores are less than some predetermined threshold.

3.3.6 Overview for finding general repeats

We now summarize the algorithm.

Algorithm 4: finding general repeats within a string:

1. Align the given string with itself, using a dynamic programming matrix. (Algorithm 3).
2. Find a largest entry in the matrix and trace back to find the repeat, R.
3. (optional) Score the repeat, R, to check your work.
4. Adjust the matrix to disallow the matches and mismatches that occurred in R.
5. Repeat steps 2-4 until the largest score is less than some predetermined value.

This algorithm is given in Java (written by Louise Yan) with the supplemental materials in Section 6. And the website http://tandem.sci.brooklyn.cuny.edu/SWrepeats (developed by Rivka Levitan) runs Algorithm 4. Simply enter a sequence into the text box. You can view the matrices, in which a * marks a cell on the path, and a $ marks an affected cell.

3.4 Advanced scoring functions

In Figure 7 we include another scoring matrix for the four bases of DNA. The scoring function, *s(a,b)*, in Figure 7 assigns the highest penalty for inserting a gap, the next highest penalty for cross pairing purines (A and G) with pyrimidines (T and C), and a low penalty for interchanging purine with purine or pyrimidine with pyrimidine.

s(a,b)		*b*				
		A	T	G	C	-
a	A	1	-.75	-.25	-.75	-1
	T	-.75	1	-.75	-.25	-1
	G	-.25	-.75	1	-.75	-1
	C	-.75	-.25	-.75	1	-1
	-	-1	-1	-1	-1	X

Figure 7 – Scoring Function for Bases in DNA

Exercise 3.6: Repeat the dynamic programming algorithm with the scoring function in Figure 7, for string AACTAAT (similar to what was done for Figure 1). What is the optimal alignment with this new scoring function? Why is there an X in the matrix?

More advanced scoring schemes use *affine gap penalties*, rather than the fixed gap penalty. A fixed gap penalty assigns a fixed cost per insertion/deletion. Affine gap penalties assign a penalty for the start of the gap (that is, the first gap in – – – . . . –), and then a lower penalty for each gap that follows. This encourages the extension of gaps rather than the introduction of new gaps. It is possible to further modify the dynamic programming algorithm for affine gap penalties, as in [Gotoh82], but with an asymptotic change to the time complexity. However, this is beyond the scope of the current module on finding repeats.

4 Conclusion

Repeats within DNA strings give scientists important information used in diagnosing diseases and identity testing. This has prompted the pursuit of efficient methods for finding repeats.

In the search for these methods, a good starting place is a brute force algorithm. After analyzing its complexity, one continues to search for better algorithms. The elegant Smith-Waterman algorithm provides the steps to find general repeats, and can be performed iteratively to generate many repeats within the same string.

The search for repeats is one of many endeavors in the area of DNA sequencing. The interested reader is encouraged to explore the many facets of this field [JP04, M04, C94].

5 Additional Exercises

5.1 Problem Listing

Homework 1: Can you develop your own brute force method of finding tandem repeats?

Homework 2: In Exercise 3.3 a scoring scheme for ranking misspelled words suggested looking at the increased likelihood of interchanging vowels while spelling. Propose a scoring scheme for ranking replacements for mistyped words that takes into account that an adjacent keyboard key to the correct one might be more likely pressed than other keys further away. Use your scoring scheme to rank replacement of LOST or LOVE for mistyped word LODR.

Homework 3: For each of the following strings, find the highest scoring general repeat using the scoring scheme $s(a,b) = \begin{cases} 1 \text{ if } a = b \\ -1 \text{ otherwise} \end{cases}$. After finding it, adjust the matrix to find the next- highest scoring repeat that is not part of the highest scoring repeat. Identify the type of repeats you found.

(a) GTCGTCGT
(b) AAGCCAGCTAAGCC
(c) MISSISSIPPI

5.2 Hints and Solutions

Solution Homework 1:
Pick a substring and compare it with the adjacent substring (on its right) of the same length. If they match, you've found a tandem repeat. Do this for all substrings for which this process is valid. In order to structure this process, you may want to start with all substrings of length 1, then length 2, up until length n/2.

Solution Homework 2:
A possible scoring scheme could score exact matches as +2, mismatches from adjacent

keys as -1, mismatches from keys on the same hand as -2 and other mismatches as -3.

```
LOST    score =2
LODR   (2 matches, and 2 mismatches from adjacent keys)

LOVE   score=1
LODR    (2 matches, 1 mismatch from keys of left hand, 1 adjacent)
```

Solution Homework 3:

This is an overlapping repeat: **GTCGT**CGT

		0	1	2	3	4	5	6	7	8
			G	**T**	**C**	**G**	**T**	**C**	**G**	**T**
0		0	0	0	0	0	0	0	0	0
1	**G**		0	0	0	1	0	0	1	0
2	**T**			0	0	0	2	1	0	2
3	**C**				0	0	1	3	2	1
4	**G**					0	0	2	4	3
5	**T**						0	1	3	5
6	**C**							0	2	4
7	**G**								0	3
8	**T**									0

The largest number in the matrix is 5, and so tracing back from this cell gives us the following general repeat:

4	5	6	7	8
G	**T**	**C**	**G**	**T**
G	**T**	**C**	**G**	**T**
1	2	3	4	5

If you are searching for repeats with smaller scores, but do not want to report any part of the repeat already found, you can adjust the matrix. In this case, the adjusted matrix is

		0	1	2	3	4	5	6	7	8
			G	**T**	**C**	**G**	**T**	**C**	**G**	**T**
0		0	0	0	0	0	0	0	0	0
1	**G**		0	0	0	0	0	0	1	0
2	**T**			0	0	0	0	0	0	2
3	**C**				0	0	0	0	0	1
4	**G**					0	0	0	0	0
5	**T**						0	0	0	0
6	**C**							0	0	0
7	**G**								0	0
8	**T**									0

The largest number in the matrix is 2, and so tracing back from this cell gives us the following general repeat:

7	8
G	**T**
G	**T**
1	2

This is a repeat at a distance: **GT**CGTCGT

Adjusting the matrix once more yields the 0 matrix.

For the solutions to 3b and 3c we ran our code (included at the end of this module) and we present the output of the code. Dark shading is represented by * and light shading by $.

3(b) AAGCCAGCTAAGCC

```
Matrix built:
  |   | A | A | G | C | C | A | G | C | T | A | A | G | C | C |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
A | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 0 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1 | 3 | 2 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 2 | 1 | 0 | 2 | 4 | 3 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 1 | 0 | 1 | 3 | 5 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 2 | 1 | 2 | 4 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 3 | 2 | 3 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 3 |
T | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Top string:   10 AAGCC  14
Left string:   1 AAGCC   5

Matrix with path:
  |   | A | A | G | C | C | A | G | C | T | A | A | G | C | C |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0*| 0 | 0 | 0 | 0 | 0 |
A | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1*| 1 | 0 | 0 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2*| 1 | 0 | 0 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1 | 3*| 2 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 2 | 1 | 0 | 2 | 4*| 3 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 1 | 0 | 1 | 3 | 5*|
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 2 | 1 | 2 | 4 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 3 | 2 | 3 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 3 |
T | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Matrix with shading:
  |   | A | A | G | C | C | A | G | C | T | A | A | G | C | C |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0*| 0 | 0 | 0 | 0 | 0 |
A | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1*| 1 | 0 | 0 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2*| 1$| 0 | 0 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1$| 3*| 2$| 1$|
C | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 2 | 1 | 0 | 2$| 4*| 3$|
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 1 | 0 | 1$| 3$| 5*|
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 2 | 1 | 2$| 4$|
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 3 | 2 | 3$|
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 3 |
T | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Adjusted Matrix:
  |   | A | A | G | C | C | A | G | C | T | A | A | G | C | C |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
A | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
C | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 2 | 1 | 0 | 0 | 0 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 1 | 0 | 0 | 1 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 2 | 1 | 0 | 0 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 3 | 2 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 3 |
T | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 |
A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

3(c) MISSISSIPPI

```
Matrix built:
  |   | M | I | S | S | I | S | S | I | P | P | I |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
S | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 1 | 0 | 0 | 0 | 0 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 2 | 1 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 3 | 2 | 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 | 2 | 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 1 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Top string:    5 ISSI   8
Left string:   2 ISSI   5

Matrix with path:
  |   | M | I | S | S | I | S | S | I | P | P | I |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
M | 0 | 0 | 0 | 0 | 0*| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 1*| 0 | 0 | 1 | 0 | 0 | 1 |
S | 0 | 0 | 0 | 0 | 1 | 0 | 2*| 1 | 0 | 0 | 0 | 0 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3*| 2 | 1 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4*| 3 | 2 | 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 | 2 | 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 1 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Matrix with shading:
  |   | M | I | S | S | I | S | S | I | P | P | I |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
M | 0 | 0 | 0 | 0 | 0*| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 1*| 0 | 0 | 1 | 0 | 0 | 1 |
S | 0 | 0 | 0 | 0 | 1 | 0 | 2*| 1 | 0 | 0 | 0 | 0 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3*| 2$| 1$| 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2$| 4*| 3$| 2$| 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3$| 3$| 2$| 1$|
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2$| 2$| 2$| 1$|
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1$| 1$| 3$|
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2$| 2$|
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1$|
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Adjusted Matrix:
  |   | M | I | S | S | I | S | S | I | P | P | I |
  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
S | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

6 Supplemental Material

6.1 Teacher's Notes

6.1.1 Additional Definitions

In this section we review some known definitions on string periodicity, providing some useful background knowledge for the instructor.

Definition 1 [prefix/suffix]:
A *prefix* of a string $S=s_1, ..., s_n$, is a substring of S, $s_1,...,s_j$; a *suffix* of S is a substring of s, $s_j,...,s_n$, for $1 \leq j \leq n$.

Definition 2 [primitive]:
A string S is *cyclic* in string v if S is of the form v^k, for $k>1$. S is a *primitive* string if S is not cyclic in any string v.

Definition 3 [periodic]:
A string S is *periodic* in v if $S=v^k v'$ where v' is a (possibly empty) prefix of v, v is primitive, and $k \geq 2$. The *period* of S is $|v|$, the number of characters in v.
An alternative definition: a string $S=s_1...s_n$ is periodic in p if $s_1...s_{n-p} = s_{p+1}...s_n$.

Lemma:
A periodic string *S* can be expressed as $v^k v'$ for one *unique* primitive *v*.

Relating Periodicity to Repeats:
An exact tandem repeat (i.e. containing **no** errors) can be viewed as a periodic string. The problem of finding all exact tandem repeats in a sequence is then equivalent to finding all periodic substrings of a string. When multiple copies of a repeat are allowed only primitive repeats are interesting. For example, consider the repeat, TATATATATATAT. This can be viewed as TA repeated 6.5 times (period=2), or TATA repeated 3.25 times (period=4), or TATATA repeated 2.16 times (period=6). The repeat with period 2 is primitive; the others are not significant.

The tandem repeat TATATATATATAT would be found by Algorithm 4 as an overlapping repeat of TATATATATAT. In fact, using the alternative definition of periodicity, every overlapping repeat found by Algorithm 4, for which the overlap is more than half the repeat's length is by definition a tandem repeat. This algorithm is used in practice for finding tandem repeats.

Remark: Many of the efficient algorithms that locate all **tandem** repeats in a string, locate all maximal repeats, and perform a separate test to eliminate non-primitive repeats. Suppose that an algorithm finds repeats starting with smaller period sizes. A simple check of whether the new repeat spans the identical substring, with a period that is a multiple of the smaller period, tells whether the larger period repeat is primitive. In the above example, since 4 and 6 are multiples of 2, the repeats are not primitive.

6.1.2 Suggestions for Presentation of Material

Depending on the background of your audience, you may need a half hour or more of time to provide them with basic DNA and biology information to motivate finding repeats in strings. Material can be presented without the framework of DNA sequencing using the motivation of text searching.

If you are looking for a good overview of basic DNA and genomic facts consider Department of Energy's Genome Programs home page:

http://www.doegenomes.org/ [DOE04].

This site contains a vast amount of information, a good starting place is: http://www.ornl.gov/sci/techresources/Human_Genome/publicat/primer/ [HGP03]. There is also a glossary: http://www.ornl.gov/sci/techresources/Human_Genome/glossary

Online encyclopedia's can be used for reference reading. The Wikipedia has detailed articles on genetics, consider: http://en.wikipedia.org/wiki/Chromosome [W04].

If you want to get to the application faster and spend less time on algorithm analysis, you could use in Section 2.2.1 a simplified polynomial definition of $O(n^k)$ time:

> Suppose that $f(n)$ is the computing time (loosely speaking, the computing time is the number of steps to perform a task) to run the algorithm on a string of length n. If $f(n) \le cn^k$ for some constants c and k, then we say the algorithm takes $O(n^k)$ time (which is read " $f(n)$ is big O of n^k " or " f is order n^k ").

Alternatively, for more advanced students, a good exploratory exercise would be to see if they could arrive at the general definition if given the above simplified version and then asked "How could one compare the order of an algorithm to other functions such as $\log(x)$, e^x, or general $g(x)$."

Provide students a template matrix similar to Figure 1 with the diagonal entries empty, while the letters along the top, and the initial row of zeros is entered. Have students enter the diagonal of zeros and calculate the upper triangle scores. Be aware that students often shift the diagonal of zeros and do not get it positioned correctly.

Additionally, to help students when they are first trying to fill out the matrix shown in Figure 1, tell them that the vertical and horizontal arrows will always subtract and they need to check the diagonal for a match or mismatch. Wait until they have the matrix filled out and are investigating the trace-back arrows to press for a deeper understanding of the meaning and the process of gap insertion.

6.2 Solutions to Numbered Exercises

Solution 2.1: We'll list the 12 tandem repeats of GTTGTTGTTGTT and show their positions in the string in bold.

There are 4 occurrences of TT. TT has size 2 and period 1.

G**TT**GTTGTTGTT

GTTG**TT**GTTGTT

GTTGTTG**TT**GTT

GTTGTTGTTG**TT**

There are 2 occurrences of TTGTTG. TTGTTG has size 6 and period 3.

G**TTGTTG**TTGTT

GTTG**TTGTTG**TT

There are 2 occurrences of TGTTGT. TGTTGT has size 6 and period 3.

GT**TGTTGT**TGTT

GTTGT**TGTTGT**T

There are 3 occurrences of GTTGTT. GTTGTT has size 6 and period 3.

GTTGTTGTTGTT

GTT**GTTGTT**GTT

GTTGTT**GTTGTT**

There is one occurrence of GTTGTTGTTGTT, the string itself. It has size 12 and period 6.

Solution 2.2:

(a) There are a total of 9 tandem repeats. AA occurs 5 times, AAAA occurs 3 times, and AAAAAA occurs once.

(b) There are a total of 12 tandem repeats. AA occurs 6 times, AAAA occurs 4 times, and AAAAAA occurs twice.

Solution 2.3: We can redefine a tandem repeat to allow the repeat to have several copies. A string *r* is a *multiple* tandem repeat if it can be partitioned into consecutive subwords, $r = v^n v', n \geq 2$ and v' is a (possibly empty) prefix of *v*. The significant tandem repeats of this type must be *primitive*, i.e. $v \neq s^k, k > 1$ and *maximal*, that is they cannot be extended by adding characters to the left or the right. With this new definition, for Exercise 2.2, the only tandem repeat in the string A^n is "A repeated *n* times." Note that this succinctly represents all $O(n^2)$ repeats in A^n . (See Section 2.2 for definition of $O(n^2)$. See also Exercise 2.7, and definitions in Section 6.1).

Solution 2.4: Since $f(n) = O(n^2)$ and $g(n) = O(n^3)$ there exist constants c_1, c_2, k_1 and k_2 such that $0 \le f(n) \le c_1 n^2$ for all $n \ge k_1$ and $0 \le g(n) \le c_2 n^3$ for all $n \ge k_2$. Let $a = \max\{c_1, c_2\}$ and $k = \max\{k_1, k_2\}$. Then $(f+g)(n) \le c_1 n^2 + c_2 n^3 \le an^2 + an^3 \le an^3 + an^3 = 2an^3$ for all $n \ge k$. Here, $c = 2a$.

Solution 2.5: $n! = n(n-1)(n-2)\cdots 3\cdot 2\cdot 1 \le n\cdot n\cdot n\cdots n\cdot n\cdot n = n^n$ holds for $n \ge 1$. Here $c = 1$.

Solution 2.6: False. Big Oh is not symmetric. Let $f(n) = n$ and $g(n) = n^2$. Then $f(n) = O(g(n))$, but $g(n) \ne O(f(n))$. It is easy to show that $f(n) = O(g(n))$. To rigorously show that $g(n) \ne O(f(n))$, you must use the definition of Big oh. The statement $g(n) \ne O(f(n))$ means that no matter which c and k you choose, you will always be able to find an $N \ge k$ such that $g(N) > cf(N)$, i.e. that $N^2 > cN$ or that $N > c$. So consider an arbitrary c and k. Let $N = \max\{k, c+1\}$. Then $N > c$ and $N \ge k$. Hence $g(n) \ne O(f(n))$.

Solution 2.7:

(a) Using the hint, we consider the modified string, /A$A/A$A/A$A/. Notice that if you choose two $'s, the string of A's between them is a tandem repeat. Therefore, the number of tandem repeats is the sum of the number of ways we can choose two /'s and the number of ways we can choose two $'s. This gives us $C(4,2) + C(3,2) = (4\cdot 3)/2 + (3\cdot 2)/2 = 9$.

(b) Our modified string is /AA/AA/AA/A. Using the same technique as in part (a), we see that the number of tandem repeats is $C(4,2) + C(4,2) = (4\cdot 3)/2 + (4\cdot 3)/2 = 12$.

(c) From our solutions to (a) and (b), we can see that the formula for the number of tandem repeats in A^n will depend on whether n is even or odd.

If n is even, the character / would appear $\frac{n}{2}+1$ times, and the character \$ would appear $\frac{n}{2}$ times. The number of tandem repeats would be

$$C(\frac{n}{2}+1,2)+C(\frac{n}{2},2)=\frac{\left(\frac{n}{2}+1\right)\left(\frac{n}{2}\right)}{2}+\frac{\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)}{2}=\frac{\frac{n}{2}\left(\frac{n}{2}+1+\frac{n}{2}-1\right)}{2}=\frac{n^2}{4}.$$

If n is odd, the characters / and \$ appear the same number of times, namely $\frac{n-1}{2}+1$.

The number of tandem repeats would be:

$$C(\frac{n+1}{2},2)+C(\frac{n+1}{2},2)=2C(\frac{n+1}{2},2)=2\frac{\left(\frac{n+1}{2}\right)\left(\frac{n+1}{2}-1\right)}{2}$$

$$=\frac{(n+1)(n-1)}{4}=\frac{n^2-1}{4}$$

Solution 2.8: Consider a modification of Algorithm 2 in Section 2. For ease of exposition, we number the locations of the input string of length n from $0...n-1$. The modified algorithm compares the string with the suffix of itself beginning at every possible location $0<i\le\frac{n}{2}$. When aligning with a shift i, we are searching for repeats with period i. Locate all maximal matching segments with length > i. Each matching segment is a repeat. This will locate multiple tandem repeats in $O(n^2)$ time.

Example: Given the string ATATATATGC, we search for repeats with period i=1,2,3,4,5. We show the alignment for period 2. Since there is a matching segment of length at least 2, we have a multiple tandem repeat. The length of the repeat is that of the match, plus the shift (which is i). The result is the repeat of length 8, ATATATAT, spanning positions 0-7.

```
                                  0 1 2 3 4 5 6 7 8 9
Original string                   A T A T A T A T G C
                                  | | | | | |
Suffix beginning at i=2       A T A T A T A T G C
(shift=period=i=2)            0 1 2 3 4 5 6 7 8 9
```

Solution 2.9: To locate repeats with a Hamming distance of *k*, we can use either brute force method presented in Section 2 (Algorithm 1 or 2). We describe a modification of Algorithm 1, which checks every even length substring. When checking a given substring, keep a count of the number of mismatches, initialized to zero. Each time a mismatch is encountered, increment the count. If the count is $< k$ when done comparing, then an approximate tandem repeat with Hamming distance $< k$ has been found.

Consider also the modification of Algorithm 2 (which is discussed in Solution 2.8).

Solution 3.1: Find every general repeat in GCGAGAGACGCC

(1) repeats at any distance

tandem GCGAGAGACG**C**C

GC**GA**GAGACGCC

GCG**AG**AGACGCC

GCGA**GA**GACGCC

non-tandem (period >1)

GCGAGAGACGCC

G**CG**AGAGACGCC

GC**GA**GAGACGCC

(2) overlapping repeats (period >1)

GC**GAG**AGACGCC **GAG is overlapped with G**AG

GCG**AGA**GACGCC **AGA** is overlapped with **A**GA

GC**GAGA**GACGCC **GAGA** is overlapped with **GA**GA

(3) repeats with 1 error (period >2)

with a mismatch **GCG**AGAGACGCC

GCGAGAGACGCC

GCGAGAGACGCC

G**CGA**GAGACGCC

G**CGA**GAGACGCC

GC**GAG**AGACGCC

GCGAGA**GAC**GCC

GCGAGAGACGCC

repeats with a missing character (period >2)

GCGAGAGACGCC

GCGAGAGACGCC
G**CG**AGAGACGCC
G**CG**AGAGACGCC
GC**GA**GAGACGCC
GC**GA**GAGACGCC
GCG**AG**AGACGCC
GCG**AG**AGACGCC
GCGA**GA**GACGCC
GCGAG**AG**ACGCC
GCGAGAGACGCC
GCGAGAGACGCC
GCGAGAGACGCC
G**CGA**GAGACGCC
G**CGA**GAGACGCC
G**CGA**GAGACGCC
GC**GAG**AGACGCC
GC**GAG**AGACGCC
GC**GAG**AGACGCC
GCG**AGA**GACGCC

Solution 3.2 The score for **BE**ERBEAR is 4; the score for **BEER**BEAR is 5.

Solution 3.3: The score for **BE**ERBEAR is 4; the score for **BEER**BEAR is 5.5.

Solution 3.4: The first repeat, **AACGT**AACCA, has a score of 4. The second repeat, **AACGT**GGCGT, has a score of 8. Even though both repeats contain 3 matching characters, which contribute 6 to the score, the first repeat has mismatches between purines and pyrimidines, while the mismatches in the second repeat are within the same type of base (A to G).

Solution 3.5: The alignment is:

```
S -OME
SCORE          with score 0
```

Solution 3.6:

The following matrix is the dynamic programming matrix computed for finding repeats in the string AACTAAT, based on the scoring function in Figure 7.

		0	1	2	3	4	5	6	7
			A	**A**	**C**	**T**	**A**	**A**	**T**
0		0	0	0	0	0	0	0	0
1	**A**		0	1	0	0	1	1	0
2	**A**			0	0.25	0	1	2	1
3	**C**				0	0	0	1	1.75
4	**T**					0	0	0	2
5	**A**						0	1	1
6	**A**							0	0.25
7	**T**								0

The best alignment with the new scoring function is

AA -T
AACT   its score is still 2.

There is an X in the matrix that provides the scoring function because the algorithm does not allow a gap to be matched against another gap. This is effectively the same as choosing X= $-\infty$.

7 Bibliography

[B95] G. Benson. A space-efficient algorithm for finding best scoring non-overlapping alignments. *Theoretical Computer Science*, 145:357-369, 1995.

[B97] G. Benson. Sequence alignment with tandem duplication. *J. Comp. Biology*, 4:351–367, 1997.

[B99] G. Benson. Tandem repeats finder – a program to analyze DNA sequences. *Nucleic Acids Research*, 27:573-580, 1999.

[BW94] G. Benson and M. Waterman. A method for fast database search for all k-nucleotide repeats. *Nucleic Acids Research*, 22:4828-4836, 1994.

[C92] C. T. Caskey et al. An unstable triplet repeat in a gene related to Myotonic Dystrophy. *Science*, 255:1256-1258, 1992.

[C94] N. G. Cooper. *The Human Genome Project: Deciphering the Blueprint of Heredity*. University Science Books, 1994.

[DOE04] U.S. Department of Energy Office of Science. Genome Programs. Retrieved October 5, 2004 from http://www.doegenomes.org/

[D90] R. F. Doolittle. Searching through sequence databases. *Methods in Enzymology.*, 183:99-110, 1990.

[Gotoh82] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biology*, 162:705-708, 1982.

[GMM04] R Groult, M. Leonard, and L. Mouchard. Speeding up the detection of evolutive tandem repeats. *Theoretical Computer Science*, 310(1-3):309-328, 2004.

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press, 1997.

[GZ05] J. R. Gatchel and H.Y. Zoghbi. Diseases of unstable repeat expansion: Mechanisms and common principles. *Nature Reviews Genetics*, 6:743755, 2005.

[HGP01] Human Genome Project. Announcements on the first analysis of Genomic Sequence (February 12, 2001). Retrieved October 5, 2004 from http://www.ornl.gov/sci/techresources/Human_Genome/project/feb_pr/vignettes.shtml

[HGP03] Human Genome Project. Genomics Primers. Retrieved October 5, 2004 from http://www.ornl.gov/sci/techresources/Human_Genome/publicat/primer

[J93a] A. J. Jeffreys. 1992 William Allan Award Address. Am. J. Hum. Genet., 53(1):1–5, 1993.

[J93b] A. J. Jeffreys. DNA typing: approaches and applications. Journal of the Forensic Science Society 33, pages 204–211, 1993.

[JP04] N. C. Jones and P.A. Pevzner. *An Introduction to Bioinformatics Algorithms.* MIT Press, 2004.

[K96] H. Kitada, K. Tono, M. Yamamoto, T. Mitamura, A. Ohuchi, T. Ohyanagi, and N. Matsushima. Multiple alignment of biological sequences containing tandem repeats. *Genome Informatics*, 7:276–277, 1996.

[KM96] S. K. Kannan and E. W. Myers. An algorithm for locating nonoverlapping regions of maximum alignments score. *SIAM J. Comput.*, 25(3):648-662, 1996.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322-350, 1977.

[KK99] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. *Proc. of Symposium on Foundations of Computer Science (FOCS)*, 596--604, 1999.

[KK01] R. Kolpakov and G. Kucherov. Finding approximate repetitions under Hamming distance. *Lecture Notes in Computer Science*, 2161:170+, 2001.

[LSS01] G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8:1-18, 2001.

[LLDA03] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319-326, 2003.

[L89] A. M. Lesk. *Computational molecular biology*. Oxford University Press, 1989.

[M04] D. W. Mount. *Bioinformatics Sequence and Genome Analysis, Second Edition*. Cold Spring Harbor Laboratory Press, 2004.

[ML84] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422-432, 1984.

[M08] S. M. Mirkin. DNA structures, repeat expansions and human hereditary disorders. *Current Opinion in Structural biology*, 16(3):351–358, 2008.

[M92] W. Miller. An algorithm for locating a repeated region. *manuscript*, 1992.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443-453, 1970.

[R04] E. Rivals. A survey on algorithmic aspects of tandem repeats evolution. *International Journal of Foundations of Computer Science*, 15(2):225-257, 2004.

[S98] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27(4):972-992, 1998.

[SM97] J. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS Publishing Company, 1997.

[SR95] G. Sutherland and R. Richards. *Simple tandem DNA repeats and human genetic disease.* Proc. Natl. Acad. Sci. USA, 92:3636-3641, 1995.

[SW81] T. F. Smith and M. S. Waterman, Identification of Common Molecular Subsequences, *Journal of Molecular Biology*, 147:195-197, 1981.

[UW93] M.W. Uform and R.K. Wayne. Microsatellites and their application to population genetic studies. *Current Opinion in Genetics and Development*, 3:939–943, 1993.

[SH02] G. Spong and L. Hellborg. A near-extinction event in lynx: do microsatellite data tell the tale? *Conservation Ecology*, 6(1):15, 2002. http://www.consecol.org/vol6/iss/art15/ 1.

[W95] M. S. Waterman. *Introduction to computational biology: maps, sequences and genomes*. Chapman & Hall, 1995.

[WYKG04] Y. Wexler and Z. Yakhini and Y. Kashi and D. Geiger. Finding approximate tandem repeats in genomic sequences. *Proc. of the 8th Ann. Conf. on Research in Comp. Biol. (RECOMB)*, 2004.

[W04] Wikipedia. Chromosome. Retrieved October 5, 2004 from http://en.wikipedia.org/wiki/Chromosome

Appendix – Source Code for Algorithms 1-4

Algorithm 1 (C++ code)

```
/***************************************************************
This C++ code implements Algorithm 1 presented in the DIMACS Education
Module, "Finding Repeats Within Strings."

Input: a string S of length n.
Output: all tandem repeats that are found in S.

Algorithm 1 uses the brute force (naive) method to report all tandem
repeats within the given sequence. A tandem repeat here is exact, i.e.
contains no errors, and contains exactly two parts (also called a
"square").
***************************************************************/

#include <stdio.h>
#include <string.h>

#define SIZE 1000

int main(void)
{
char S[SIZE];
char repeat[SIZE];

int len=2, p, i;

printf("please enter a string: ");
fgets(S, SIZE-1, stdin);

int n=strlen(S);

printf("S=%s, length=%d \n",S, n);

while (len <=n)
    // len stands for the length of the substring being checked
                                        // for tandem repeats

 {
   p = len/2;  // p stands for the period of the repeat

   for (i=0; i<=n-len; i++)
                 // compare contiguous substrings of length p, one
starting at position i,
                 // the other starting at position i+p
     {
      if (strncmp(S+i,S+i+p,p)==0)
        {
                 // copy over repeat for printing purposes
                 strncpy(repeat,&S[i],len);
                 repeat[len]='\0';
            // report repeat beginning at position i of length len
                 printf("repeat found at position %d  (%s) of length
%d\n",i+1,repeat,len);
        }
     } // end for loop
   len=len+2;
 }

return 0;
}
```

Algorithm 2 (C++ code)

```
/****************************************************************
This C++ code implements Algorithm 2 presented in the DIMACS Education
Module, "Finding Repeats Within Strings."

Input: a string S of length n.
Output: all tandem repeats that are found in S.

Algorithm 2 is an improvement on Algorithm 1, and has worst case time
complexity of O(n^2).
All tandem repeats within the given sequence are reported.
A tandem repeat here is exact, i.e.
contains no errors, and contains exactly two parts (also called a
"square").
****************************************************************/


#include <stdio.h>
#include <string.h>

#define SIZE 1000

int main(void)
{
char S[SIZE];
char repeat[SIZE];
int match=0, p, i;

printf("please enter a string: ");
fgets(S, SIZE-1, stdin);
//gets(S);

int n=strlen(S);
printf("S=%s, length=%d \n",S, n);


for (p=1; p<=n/2; p++)  // p stands for the period of the repeat
{
  for (i=0;i<n-p;i++)    // for each i, check whether a repeat ends at
location i+p
    { //printf("p=%d i=%d  ",p,i);
    // count how many consecutive characters match
      if (S[i]==S[i+p]) {
            match++;

            if (match>=p)
               {
                // copy over repeat for printing purposes
                 strncpy(repeat,&S[i-p+1],2*p);
                 repeat[2*p]='\0';
                // report a repeat of length 2p beginning at location
i-p+1 and ending at i+p
                 printf("new repeat of length %d at location %d is:
%s\n ",2*p,i-p+2,repeat);
               } // end if match>=p
            } // end if S[i]==S[i+p]

      else match=0;
     }  // end for i
} // end for p

return 0;
}
```

Algorithms 3 and 4 (Java code)

```
/***********************************************************************
This Java code implements Algorithms 3 and 4 presented in the DIMACS
Education
Module, "Finding Repeats Within Strings."

Input: 1.  a string S of length n
       2.  a score for matches, mismatches, and gaps
       3.  an integer k.

Output: all general repeats that are found in S that have a total score
<=k.

To COMPILE the program, type on the command line:
```

```
javac SW.java
```

To RUN, type on the command line:

```
java SW string k
```

where string is the input sequence of characters and k is an integer (i.e. the error threshold).

The scores are represented by the static variables at the beginning of the program: MATCHSCORE, MISMATCHSCORE AND GAPSCORE.
These can be changed before compilation by the user.

Note: it is not difficult to change this to use a scoring matrix, instead of these variables, however, the program does not work that way right now.
IMPLEMENTATION DESCRIPTION:

The program is based upon the Smith-Waterman (SW) algorithm for local sequence alignment. It uses the SW algorithm to align the input string against itself to find the repeats within the input sequence. All repeats with score <= input threshold are reported.

To aid in understanding of the algorithm, the matrix is printed several times. This can be commented out by the user.

Descriptions of functions:

buildMatrix() builds the dynamic programming matrix, as shown in Algorithm 3. The string can be thought of as being placed on the top and to the left of the matrix. The diagonal of the matrix is set to 0. Only the upper-right portion of the matrix is computed. The value or score for an element in the matrix is computed from the max of three values and 0. The three values correspond to a move in the diagonal, a move vertically down (from the top), and a move horizontally to the right. For a move along the diagonal, the characters at the given positions in the string are compared. If the characters match, the cell takes a score equal to the score of the diagonal cell plus MATCHSCORE. If the characters do not match, the cell takes a score equal to the score of the diagonal cell plus MISMATCHSCORE. For a move vertically down, a gap is taken for the top substring. The score of the cell is equal to the score of the top cell plus GAPSCORE. For a move horizontally to the right, a gap is taken for the left substring. The score of the cell is equal to the score of the left cell minus GAPSCORE.

findMax(): After buildMatrix returns, findMax() finds the highest score in the matrix and returns its position.

```
traceback():  Using the position of the highest score in the matrix
(returned by findmax()), traceback() looks at each of the possible
cells that the position could have been calculated from and traces
these calculations back until it finds a zero. These locations are
called the path. The actual alignment is
retrieved while tracing back the path.


adjustMatrix()  adjusts the matrix to dissallow all parts of the
previously reported repeat. First, the function
shade() is called to shade the cells that need to be recomputed. Then,
the function adjustMatrix
takes the max of the diagonal, top, left and 0, just like
buildMatrix(), HOWEVER, it disallows the path.

shade() labels the positions as shaded if they are affected by the
path. Every value on the path can be thought of as shaded. Any position
that must be calculated from a shaded cell (can't be calculated from a
non-shaded) is also shaded.
shade() does this by looking at cells beginning with the first non-zero
position on the path and continues on that row until it finds a non-
shaded cell.
For every row after that, it begins checking under the leftmost shaded
cell of the previous row.

The algorithm goes through traceback and adjusting until the highest
score in the matrix is below the given threshold.

****************************************************************/




import java.lang.StringBuffer;


class SW{
      public static final int MATCHSCORE = 1;               //score for
a match

      public static final int MISMATCHSCORE = -1;           //score for
a mismatch
      public static final int GAPSCORE = -1;                //gap
penalty

      public static void main(String[] args) {
      String str = args[0];                           //takes a
command-line argument
      int threshold = Integer.parseInt(args[1]);            //takes a
command-line argument, and parses it using the parseInt method in the
Integer class
```

```
      int[][] matrix = buildMatrix(str);

      System.out.println("Matrix built:");
      printMatrix(matrix, str);
      findRepeats(matrix,str,threshold);

      }



      /* buildMatrix takes a String and compares it with itself to
produce a matrix of scores.
         buildMatrix creates a matrix with the first row and the
diagonal set to all 0s. The score set in the matrix
         for each cell depends on the max of the diagonal score plus a
MATCHSCORE or MISMATCHSCORE, the left score plus
         GAPSCORE, the top score plus GAPSCORE,and zero.
         buildMatrix returns the filled matrix
      */
      public static int[][] buildMatrix(final String str) {
            int[][] matrix = new int[str.length()+1][str.length()+1];
            int score;
            for(int i = 0; i <= str.length(); i++){
                  matrix[0][i] = 0;
      //initialize the first row
                  matrix[i][i] = 0;
      //initialize the diagonal
            }
      for(int i = 1; i <= str.length(); i++) {
            for(int j = i + 1; j <= str.length(); j++) {
                  if(str.charAt(j-1) == str.charAt(i-1))
      //test for a match
                        score = MATCHSCORE;
                  else
                        score = MISMATCHSCORE;
                  matrix[i][j] = max(matrix[i-1][j-1] + score,
matrix[i-1][j] + GAPSCORE, matrix[i][j-1] + GAPSCORE, 0);
            }
      }
      return matrix;
      }




      /* findRepeats takes a 2D matrix of int, and a string.
            findRepeats calls traceback, adjustMatrix and findMax in a
loop until the
         max score in the matrix is lower than the threshold.
      */
      public static void findRepeats(int[][] matrix, String str, int
threshold) {
            Position[] path;
            Position end_path = findMax(matrix);
            while(matrix[end_path.getRow()][end_path.getCol()] >=
threshold) {
                  path = traceBack(matrix, str, end_path);
                  matrix = adjustMatrix(matrix, path, str);
                  System.out.println("Adjusted Matrix: ");
                  printMatrix(matrix, str);

                  end_path = findMax(matrix);
            }
      }




      /* findMax searches a 2d matrix for Position of maximum score and
returns the Position */
      public static Position findMax(int[][] matrix) {
            int max = 0;
            Position maxPos = new Position();
            for(int i = 1; i < matrix.length;i++) {
                  for(int j = i; j < matrix.length; j++) {
                        if(matrix[i][j] >= max) {
                              max = matrix[i][j];
                              maxPos = new Position(i,j);
                        }
                  }
            }
            return maxPos;
      }



      /* traceback takes a 2D matrix of int representing scores, a
string and a Position that contains the highest score.
         This Position is also the end of the path. traceback traces
the path by figuring out where the score could have
         been calculated. If the score came from the left, a gap occurs
in the left string; if the score came form the top,
         a gap occurs in the top string. traceback prints the topstring
and leftstring and returns the path
      */
      public static Position[] traceBack(int[][] matrix, String str,
Position end_path) {
            int i = end_path.getRow();
            int j = end_path.getCol();

            Position[] tmppath = new Position[2*i];
            int path_i = 0;
      //keeps track of the path position
            StringBuffer topStrbf = new StringBuffer();
            StringBuffer leftStrbf = new StringBuffer();


            while(matrix[i][j] != 0 ) {
                tmppath[path_i] = new Position(i,j);
        //save the path Position
                path_i++;

                if(matrix[i][j] == matrix[i][j-1] + GAPSCORE) {
        //compare score with left score minus gap penalty
                    topStrbf.insert(0, str.charAt(j-1));
                    leftStrbf.insert(0, "-");
        //horizontal move corresponds to a gap in the left string
                    j--;
                }
                else if(matrix[i][j] == matrix[i-1][j] + GAPSCORE) {
        //compare score with the top score minus gap penalty
                    topStrbf.insert(0,"-");
        //vertical move corresponds to a gap in the top string
                    leftStrbf.insert(0,str.charAt(i-1));
                    i--;
                }
                else {
        //diagonal move
                    topStrbf.insert(0, str.charAt(j-1));
                    leftStrbf.insert(0,str.charAt(i-1));
                    i--; j--;
                }
            }
            tmppath[path_i] = new Position(i,j);
        //save the path Position

            //create a smaller array to store the Positions in the
correct order
            //(tmppath is larger and is in reverse order)
            Position[] path = new Position[path_i + 1];
            for(i = 0; i < path.length; i++) {
                path[i] = tmppath[path_i - i];
            }

            String topStr = topStrbf.toString();
        //convert the Java immutable StringBuffer object to String
            String leftStr = leftStrbf.toString();
            System.out.println();
            System.out.print("Top string:  ");
            System.out.printf("%3d",(path[0].getCol()+1));
        //print the index where the top String begins
            System.out.print(" " + topStr + " ");
        //print the top string
            System.out.printf("%3d",(path[path.length-1].getCol()));
        //print the index where the top string ends
            System.out.println();
            System.out.print("Left string: ");
            System.out.printf("%3d",(path[0].getRow()+1));
        //print the index where the left string begins
            System.out.print(" " + leftStr + " ");
        //print the top string
```

```
            System.out.printf("%3d", (path[path.length-1].getRow()));
      //print the index where the left string ends
            System.out.println("\n");

            printMatrix(matrix, str, path);
            return path;
      }


      /* max takes four ints and returns the max of the four */
      public static int max(int diag, int top, int left, int zero) {
            int max = 0;
            max = diag;
            if(max < top)
                  max = top;
            if(max < left)
                  max = left;
            if(max < zero)
                  max = zero;
            return max;
      }



      /* adjustMatrix takes a 2D matrix of scores, an array of Position
that represent the path, and a string.
         adjustMatrix calls shade() to get a 2D array representing the
shaded cells of the matrix. adjustMatrix looks at
         every cell that is shaded and adjusts them by calculating a
new score but the new score can not be gotten from
         the path. adjustMatrix returns the adjusted matrix
      */
      public static int[][] adjustMatrix(int[][] matrix, Position[]
path, String str) {
            Position pathfirstpos = path[0];
            int pathfirstrow = pathfirstpos.getRow();
            int[][] shaded = new int[matrix.length][2];
            shaded = shade(matrix, path, str);
            printMatrix(matrix, str, path, shaded);
            int top, diag, left, score;
            //look at the shaded cells and update those cells,
disallowing the path during the calculation
            //continue adjusting until a leftmost value is -1 (if
leftmost value is -1, no shaded cells are on row)
            for(int i = pathfirstrow+1; i < matrix.length &&
shaded[i][0] != -1; i++) {
                  for(int j = shaded[i][0]; j <= shaded[i][1]; j++) {
                        if(str.charAt(i-1) == str.charAt(j-1))
      //compare string to itself for a match
                              score = MATCHSCORE;
                        else
                              score = MISMATCHSCORE;
                        if(isPath(path, i-1, j-1, i, j))          //if
the position is on the path and diag is on the path
                              diag = 0;                     //set the
score for the diagonal to 0
                        else diag = matrix[i-1][j-1] + score;
      //otherwise, calculate the score for the diagonal

                        if(isPath(path, i-1, j, i, j))
      //if the position is on the path and the top is on the path
                              top = 0;                      //set the
score for the top to 0
                        else top = matrix[i-1][j] + GAPSCORE;
      //otherwise calculate the score for the top

                        if(isPath(path, i, j-1, i, j))
      //if the position is on the path and left is on the path
                              left = 0;                     //set the
score for the left to 0
                        else left= matrix[i][j-1] + GAPSCORE;
      //otherwise calculate the score for the left

                        matrix[i][j] = max(diag, top, left, 0);
      //update the matrix, setting a new max for the cell
                  }
            }
            return matrix;
      }




      /* shade takes a 2D matrix of scores, the path, and a string.
         shade checks the matrix going row by row starting with the
first position on the path and decides whether a cell on the matrix is
to be shaded. A            cell is to be shaded if its score is
affected by the path or a shaded cell. In this function, the leftmost
shaded cell of a row must be found (not
         -1) before the rightmost column can be  found. The rightmost
column is updated until the end of the row or until the score is not
affected by the

path or any shaded cell.
         shade returns a 2D array of int. The row number of the 2D
array corresponds to the row number of the matrix. The first column of
the 2D array               represents the column number of the leftmost
shaded cell of the matrix on the given row while the second column
represents the column number of             the rightmost shaded cell
of the matrix on the given row.
      */
      public static int[][] shade(int[][] matrix, Position[] path,
String str) {
            int score, max;

            int[][] shaded = new int[matrix.length][2];
            //initialize the first column of shaded to -1 for every
row.
            //(-1 in the first column of shaded means that the leftmost
value has not been found or does not exist for the given row)
            for(int i = 0; i < shaded.length; i++) {
                  shaded[i][0] = -1;
            }
            //set leftmost and rightmost shaded according to the path
            for(int i = 0; i < path.length; i++) {
                  int row = path[i].getRow();
                  if(shaded[row][0] == -1)
                        shaded[row][0] = shaded[row][1] =
path[i].getCol();
                  else
                        shaded[row][1] = path[i].getCol();
            }




            //start shading from the second row on the path
            //continue as long as previous row is shaded (leftmost
shaded column has been found for the previous row)
            for(int i = (path[0].getRow())+1; i < matrix.length &&
shaded[i-1][0] != -1; i++) {
                  boolean foundEnd = false;
                  //begin shading from the previous row's leftmost
shaded column unless it's on the diagonal(i = j). if it is, start after
the diagonal


                  for(int j = (shaded[i-1][0] > i ? shaded[i-1][0] :
i+1); j < matrix[0].length && !foundEnd; j++) {
                        if(str.charAt(i-1) == str.charAt(j-1))
      //test for a match
                              score = MATCHSCORE;
                        else
                              score = MISMATCHSCORE;
                        max = max(matrix[i-1][j-1] + score, matrix[i-
1][j] + GAPSCORE, matrix[i][j-1] + GAPSCORE, 0);


                        //check if this cell comes from shaded cell
                        boolean comesFromShaded =
                              ! ((max == matrix[i-1][j-1] + score) &&
!isShaded(shaded, i-1, j-1) ||      //diag score is max, and diag is
not shaded
                                (max == matrix[i-1][j] + GAPSCORE) &&
!isShaded(shaded, i-1, j) ||  //top score is max, and top is not shaded
                                (max == matrix[i][j-1] + GAPSCORE) &&
!isShaded(shaded, i, j-1));   //left score is max, and max is not
shaded
```

```
                        //if a nonzero max comes from shaded, update
the leftmost shaded column if it hasn't been found yet.
                        //and update the rightmost shaded column each
time.
                        if (max != 0 && comesFromShaded)
      //if the nonzero max comes from shaded
                        {
                              if(shaded[i][0]== -1)
      //if the leftmost shaded column hasn't been found yet
                                    shaded[i][0] = shaded[i][1] =  j;
      //set leftmost & rightmost
                              else if (shaded[i][0] != -1 && j <
shaded[i][0] )     //if current column is less than leftmost shaded
column
                                    shaded[i][0] = j;
                        //set leftmost shaded

                              else if (j > shaded[i][1])
      //if the current column is greater than the rightmost
                                    shaded[i][1] = j;
      //update rightmost shaded column
                        }


                        // if the leftmost shaded column has been found
and
                        // the top, diagonal, and left for the next
cell are unshaded

                        // then there can be no more shaded cells for
the current row
                        if (shaded[i][0] != -1 && !(isShaded(shaded, i-
1,j) || isShaded(shaded, i-1, j+1) || isShaded(shaded, i, j))
)

                              foundEnd = true;

                        // if the leftmost shaded column has not been
found and
                        // the current cell is one past the cell
diagonal to the last column of the previous row's rightmost shaded
column
                        // then the row does not have any shaded cells
                        if (shaded[i][0] == -1 && j > shaded[i-1][1]+1)
                              foundEnd = true;
                  }

            }


            return shaded;
      }
```

```
	/* isShaded takes a 2D array of int. The row number of the 2D
array corresponds to the row number of the matrix. The first column of
the 2D array               represents the leftmost shadedcell of the
matrix on the given row while the second column represents the
rightmost shaded cell of the matrix on
	   the given row. The function also takes an int row and int col
which represent a position in the matrix.
	   isShaded returns true if the position represented by int row
and int col is a shaded cell
	*/
	public static boolean isShaded(int[][] shaded, int row, int col)
{
		if(shaded[row][0] != -1 && col >= shaded[row][0] && col <=
shaded[row][1]) { //if the column is in the range between the start and
end
				return true;					//it
is shaded
		}
		return false;
	//otherwise, it is not shaded
	}




	/* isPath takes an array of Position taht are on the path, and
four ints; int prev row and int prev col represent the position that
should precede
	   the position represented by int row and int col.
	   isPath returns true if the path position can be represented by
row and col and its preceding position can be represented by prevrow
and prevcol
	*/
	public static boolean isPath(Position[] path, int prevrow, int
prevcol, int row, int col) {
		Position pathpos = path[0];		//assign the value of
the first position on the path
		//iterate through the array and look for a position that
matches the given row and col
		//then check if the position before it matches prevrow and
prevcol
		for(int i = 0; i < path.length; i++) {
			pathpos = path[i];		//update the value the
the given path position
			if(pathpos.getRow() == row && pathpos.getCol() ==
col) {	//if the position matches the given row and col
				if(path[i-1].getRow() == prevrow && path[i-
1].getCol() == prevcol) //if the position matches prevrow and prevcol
					return true;
		//position matches row and col and prevrow, prevcol
```

```
                        else
                                return false;
        //position matches row and col but not prevrow and prevcol
                        }
                }
                return false;           //no positions on the path match
the passed row and col

        }




        /* isPath takes an array of Position that are on the path, and
two ints which represent a row and column
           isPath returns true if the path contains the Position that is
represented by the row and col
        */
        public static boolean isPath(Position[] path, int row, int col) {
                Position pathpos = path[0];         //assign the value of
the first position on the path
                //iterate through the path array and look for a position
that matches the given row and col
                for(int i = 0; i < path.length; i++) {
                        pathpos = path[i];      //assign a new value to
pathpos
                        if(pathpos.getRow() == row && pathpos.getCol()
== col)
                                return true;      //if the row and col of
that pathpos is equal to the passed row and col, return true

                }
                return false;                       //the path does not
have a Position with a row and col equal to the passed row and col
        }




        /* printMatrix is given a matrix and a string
           printMatrix prints out the matrix and prints the string twice
(horizontally above the matrix, and vertically to the left of the
matrix
        */
        public static void printMatrix(int[][] matrix, String str){
                System.out.print("  |   | ");
                //print out the top string
                for(int i = 0; i < str.length(); i++) {
                        System.out.print(str.charAt(i) + " | ");  //charAt
prints the character at position i of str
                }
                System.out.println();
                //print out left string and matrix values
                for(int i = 0; i < matrix.length; i++) {
                  if(i == 0)
                        System.out.print("  | ");                //if
it is the first row of the matrix, print " | "
                  else
                        System.out.print(str.charAt(i-1) + " | ");
      //otherwise, print the character at the given position
                  //print the matrix values
                  for(int j = 0; j < matrix.length; j++)
                        System.out.print(matrix[i][j] + " | ");
                  System.out.println();
            }
      }

      /* printMatrix is given a matrix, string, an array of Positions
that are on the path
         printMatrix prints out the matrix with the path cell values
suceeded by '*' and shaded cell values succeeded by '$'
       */
      public static void printMatrix(int[][] matrix, String str,
Position[] path) {
            System.out.println("Matrix with path: ");
            System.out.print("  |   | ");
            //print out the top string
            for(int i = 0; i < str.length(); i++) {
                  System.out.print(str.charAt(i) + " | ");
            }
            System.out.println();
            //print out left string and matrix values, mark path cells
with a '*'
            for(int i = 0; i < matrix.length; i++) {
                  System.out.print(i == 0 ? "  | " : str.charAt(i-1) +
" | ");     //if it is the first row of the matrix, print "  | "


      //otherwise, print string character at given position

                  for(int j = 0; j < matrix.length; j++) {
                        if(isPath(path, i,j))
                              System.out.print(matrix[i][j]+"*| ");
      //if the position is on the path, succeed the printed value with
*
                        else
                              System.out.print(matrix[i][j] + " | ");
      //otherwise, just print the value
                  }
                  System.out.println();
            }

      }
      /* printMatrix is given a matrix, string, an array of Positions,
and a 2D array of ints representing the shaded cells of the matrix
         printMatrix prints out the matrix with the path cell values
suceeded by '*' and shaded cell values succeeded by '$'
       */
      public static void printMatrix(int[][] matrix, String str,
Position[] path, int[][] shaded) {
            System.out.println("Matrix with shading: ");
            System.out.print("  |   | ");
            //print out the top string
            for(int i = 0; i < str.length(); i++) {
                  System.out.print(str.charAt(i) + " | ");
            }
            System.out.println();
            //print out left string and matrix values, mark path cells
with a '*', and shaded cells with a '$'
            for(int i = 0; i < matrix.length; i++) {
                  System.out.print(i == 0 ? "  | " : str.charAt(i-1) +
" | ");     //if it is the first row of the matrix, print "  | "


      //otherwise, print string character at given position
                  for(int j = 0; j < matrix.length; j++) {
                        if(isPath(path, i, j))
                              System.out.print(matrix[i][j]+"*| ");
      //if position is on the path, succeed printed value with *
                        else if(isShaded(shaded, i,j))
                              System.out.print(matrix[i][j]+"$| ");
      //if position is shaded, succed printed value with $
                        else
                              System.out.print(matrix[i][j] + " | ");
      //otherwise, print cell value normally
                  }
                  System.out.println();
            }
      }
}


class Position{
      private int row, col;

      Position(){;}

      Position(int row, int col){
            this.row = row;
            this.col = col;
      }


      public int getRow() {return row;}
      public int getCol() {return col;}

}
```