CS100 B Fall 1999

Professor David I. Schwartz
Programming Assignment 4

Due: Thursday, November 4

1. Goals

This assignment will help you develop skills in using arrays and software development. You will:

- develop software that uses one- and two-dimensional arrays
- determine good sample cases to test your software
- solve some problems in computational biology using your software

2. Motivation

Today's advanced technology can perform DNA sequencing on a truly industrial scale. Research generates new data at an explosive rate. Starting about 1984, the volume of DNA-sequence data has doubled approximately every 15 months! This flood of information provides one of Biology's greatest challenges – transforming this raw data into meaningful information. The field of ***computational genomics*** endeavors to provide biological "understanding" from this data-flood through mathematical models and computer algorithms. Since DNA is so fundamental to Biology, advances in this field will impact medicine and agriculture, and will improve insights into Biology.

3. Background

This section presents material for those who wish to know the underlying concepts. If you don't understand some of the biological details but understand enough to do the assignment, that's O.K.

3.1 Genome

Let's back up a little. What's a ***genome***? And, why is it important?

A genome is all the DNA in an organism, including the organism's genes. A ***gene*** carries information for making all the proteins required by all organisms. These proteins determine, among other things, how the organism looks, how well its body metabolizes food or fights infection, and even how the organism behaves.

3.2 DNA

DNA literally means deoxyribonucleic acid. But, what is it?

A group of similar chemicals called ***nucleotides*** make up DNA. The four types of ***nucleotides***, also called ***bases***, are Adenine, Cytosine, Guanine, and Thymine. Don't worry about

the names of the four nucleotides. Instead, refer to each nucleic acid by its first letter. You can now refer to the four nucleotides as A, C, G, and T. These bases repeat millions or billions of times throughout a genome. The human genome, for example, has 3 billion pairs of bases.

You may have heard that DNA forms a ***double helix***. A double helix has two strands of DNA that bind together and twist into a helical shape. For this assignment, ignore this fact, and think of only a single strand of DNA. The other strand in the double helix is always "complimentary." Knowing what one strand "looks" like (i.e., which nucleotides compose the strand) uniquely determines what the complementary strand looks like. So, considering only one strand of DNA does not discard any essential data.

DNA is essentially a linear molecule. Each base strongly binds to no more than two other bases. These two bases bind before and after the original base. Considering other bases and their "links" provides a model of DNA-strand as a sequence of nucleotides. The particular order of As, Ts, Cs, and Gs is extremely important! The order underlies all of life's diversity, even dictating an organism's species. Different orders generate humans, yeast, rice, or even fruit flies.[1] Because all organisms relate through similarities in DNA sequences, insights gained from nonhuman genomes provide new knowledge about human biology.

3.3 A Computer Scientist's View of DNA Sequences

Computer scientists think of a DNA sequence as a string from an alphabet of 4 characters: A, T, C, and G. This model encompasses the idea that the four types of bases arrange in any order and that the order is important. Why? Just imagine that the English strings *mate*, *meat*, *team*, and *tame* all meant the same thing!

3.4 DNA Sequence Analysis

DNA Sequence Analysis describes the portion of computational genomics involved with creating and using tools. These tools perform ***data reduction*** by taking large amounts of DNA sequence data and deriving smaller amounts of data which are more easily understood. Some of this derived information includes:

- Comparing DNA sequences from different species or from different places in one species' genome to investigate similarities. Your assignment will focus on this approach.
- Determining which portions of a DNA sequence are the "coding regions" (genes) and which portions regulate the genes. Regulating a gene means determining when the gene is "turned on," meaning when the protein encoded by the gene is produced.
- Compiling statistics about large sections of DNA, such as the percentage of As or the percentage of As plus Ts, or the number of times the substring "ATA" appears.

3.5 Rationale for DNA Sequence Comparison

DNA sequence comparison provides many uses. Current scientific theories suggest that very similar DNA sequences have a common ancestor. The more similar two sequences are, the more recently they evolved from a single ancestral sequence. For this reason, measures of sequence similarity help reconstruct ***phylogenetic trees***, sometimes known as a "tree of life." A

1. In fact, each of these species has a full-scale genome project devoted to mapping, and understanding the entire genome.

phylogenetic tree graphically shows how long ago various organisms diverged and which species are closely related.

Very similar DNA sequences encode similar proteins, which perform similar functions. Biologists compare newly discovered genes with all known DNA sequences and hypothesize that the new gene functions similarly to the genes which it closely resembles. Laboratory biologists then test this hypothesis in the lab and greatly shorten the time learning the behavior of the new gene.

4. DNA Sequence Comparison

This section develops algorithms for performing a problem of computational genomics.

4.1 Simple Distance

The number of bases two DNA sequences have in common provides a simple measure of similarity. A converse criteria called ***distance*** measures the dissimilarity by counting the number of positions with different bases. The ***Hamming distance*** refers to this measure[2]. For example, inspect the two DNA sequences shown in Figure 1. Because two nucleotides differ (2nd and 8th position), the DNA sequences have a Hamming distance of 2.

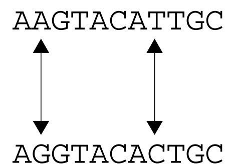


Figure 1: Hamming DIstance

4.2 Weighted Distance

The Hamming distance assumes that all differences are "created equal." For example, the difference between `A` and `G` matches the difference between `A` and `C`. However, the model of Hamming distance oversimplifies the actual biological process. Due to size, shape, charge, and other chemical properties of the various nucleotides, some bases replace others more easily. For instance, a `G` more easily replaces an `A` than a `C` does. Therefore, applying a weight to each difference between two DNA sequences helps improve the model of distance. Substitutions more likely to occur during evolution have a lower weight than those less likely to occur.

4.3 DNA substitution matrices

A ***substitution matrix*** typically arranges the weights in a 2-dimensional array, as depicted in Figure 2:

- Each row corresponds to the base in one sequence, or the ancestral base.
- Each column corresponds to the base in the other sequence, or the evolved base.

2. Check out EE 445 (Computer Networks and Telecommunications)

A number occupying (row, col) = (A, T) location represents the weight for changing A to T. The diagonal values (A, A), (C, C), (T, T), and (G, G) are zero. Why? There's no difference between A and A! Figure 3 demonstrates an example substitution matrix.

The Hamming distance can be considered a weighted distance with an equal distance between all differing nucleotides. Figure 4 shows the weighting matrix for the Hamming distance. Thus, the weighted distance described here is often called the ***weighted Hamming distance***.

Ancestral Base \ Evolved Base	A	C	G	T
A				
C				
G				
T				

Figure 2: Substitution Matrix

Ancestral Base \ Evolved Base	A	C	G	T
A	0	3	1	3
C	3	0	3	1
G	1	3	0	3
T	3	1	3	0

Figure 3: Example Values

Ancestral Base \ Evolved Base	A	C	G	T
A	0	1	1	1
C	1	0	1	1
G	1	1	0	1
T	1	1	1	0

Figure 4: Hamming DIstance

4.4 Matrix Notation

You can express the matrix in Figure 3 as M. To reference a particular element at the coordinate of row i and column j, use the notation M_{ij}. For instance, $M_{11} = 0$ and $M_{12} = 3$.

4.5 Matrix Symmetry

This matrix has a special feature you might notice. Inspect the matrix in Figure 5. The four shaded elements M_{11}, M_{22}, M_{33}, and M_{44} comprise the ***main diagonal*** of the matrix. Now, look at elements in directions perpendicular to the main diagonal. As shown in Figure 5, you will find that these elements match. For example, the fourth row has the same values as fourth column.

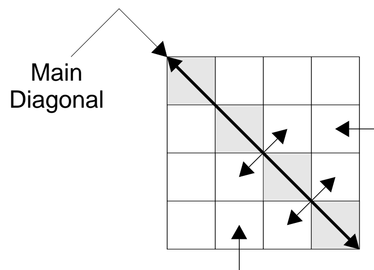


Figure 5: Matrix Symmetry

These conditions give ***matrix symmetry***. In general, a symmetric matrix has the condition

$$M_{ij} = M_{ji}. \qquad (1)$$

A symmetric substitution matrix helps model distance. You can assume that the distance between two bases is identical regardless of which direction you choose. For instance, the distance going from `A` to `G` matches the distance going from `G` to `A`. Consequently, the distance from one DNA string to another is the same in either direction.

4.6 Alignment with Gaps

So far, DNA sequences have assumed the same number of nucleotides. Also, you have compared positions sequentially, working from left to right. However, sometimes Nature inserts or deletes a base in the middle of a sequence. For example, suppose you start with an ancestral DNA string of `AGCTAACATACGAT`. Now, suppose that over time an ***insertion event*** duplicates the first G and, independently, the second `G` is deleted or vanishes. Our modern DNA string is `AGGCTAACATACAT`. An attempt to align these two strings illustrates many differences, as shown in Figure 6:

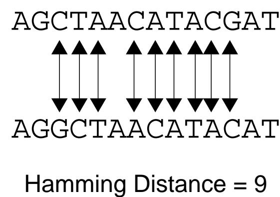


Figure 6: Differences Between Strings

However, allowing the addition of “gaps” can create a better alignment, one with no differences between corresponding bases, as shown in Figure 7:

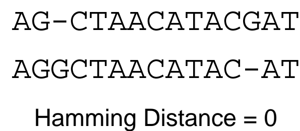


Figure 7: Gaps

By adding gaps, the two input DNA sequences do not need to be the same length. The shorter sequence will receive more gaps than the longer sequence.

4.7 Gap Penalty

Now, you have a new problem. Differing nucleotides will never align. Why? The algorithm would add gaps whenever the corresponding bases in the two DNA sequences differ, shown in Figure 8:

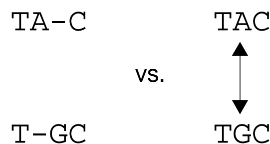


Figure 8: Lack of Alignment

You can correct for this problem by assessing a ***gap penalty***, an amount added to the distance to account for this evolutionary change. Then, you can get the second alignment shown in Figure 8. With this gap penalty, the Hamming distance in Figure 7 is now twice the gap penalty.

So, where should you add gaps?

4.8 Minimum Distance

There are many ways to align two DNA sequences. Any number of gaps can be inserted anywhere in either of the two DNA sequences. Somehow, an optimal alignment will generate the minimum possible distance between them. Computer Scientists often call this minimum distance the ***edit distance***. At first glance, this optimal alignment looks extremely difficult to compute. However, a computationally efficient method called the Smith-Waterman algorithm finds the optimal alignment, given a substitution matrix and a gap penalty.[3]

But first, some terminology:

- A ***prefix*** is an initial substring, i.e., the first i characters of the string for some number i.
- The shortest possible prefix is the empty string, where $i = 0$.
- The longest prefix is the entire string.

For example, the string “`CAT`” has four prefixes: the empty string “ ”, “`C`”, “`CA`”, and “`CAT`”.

Understanding why the Smith-Waterman[4] algorithm works is complicated, and you will *not* be expected to understand it for this class. However, you will be expected to understand what the algorithm does and how to implement it, but not why it is correct. If you’re curious why it works, consult the next section.

4.9 Rationale of The Smith-Waterman Algorithm (Optional)

Look at the last character of each of the two DNA sequences: you can align these two characters only three ways:

- Align the two characters together.
- Align the character from the first sequence with a final gap.
- Align the character from the second sequence with a final gap.

A final gap refers to a gap placed as the last character in a sequence.

Suppose you know the optimal alignment of two DNA sequences. Suppose, also, that the last character of each of the two DNA sequences align with each other in the optimal alignment. Then, the first part of the alignment optimally aligns prefixes of these two sequences which include all characters except the last character.[5] In other words, taking the optimal alignment of the two DNA sequences and removing the final pair of characters results with the optimal alignment for the resulting sequences. Each sequence loses one character from the original DNA sequence.

---

3. To Computer Scientists, this algorithm demonstrates the concept of *dynamic programming*. If you continue with computer science, you may see this algorithm in CS 482 (Introduction to Analysis of Algorithms) and CS 681 (Design and Analysis of Algorithms).
4. Needleman and Wunsch first proposed the definition and basic approach. Smith and Waterman improved this approach.
5. Please trust us on this method!

So, suppose you do not know the complete optimal alignment of the two DNA sequences. But, you do know that the last characters of our two DNA sequences align with each other in an optimal alignment. Therefore, you could just remove the last characters from the sequences and find the optimal alignment of these two smaller sequences. After finding the optimal alignment for this shorter problem, you could restore the final characters and, thus, achieve an optimal alignment for the original problem.

Similarly, suppose you know that the last character of one of the DNA sequences should align with a final gap. Remove that last character and find the optimal alignment of a smaller pair of sequences. In this case, only one of the DNA sequences is shorter than the given DNA sequences. Of course, how the final characters align in an optimal alignment is unknown. But, given only three choices, you can try all three possibilities as if you did know and choose the best alignment of the three.

5. Implementation of The Smith-Waterman Algorithm

Based on Section 4.9, looking at progressively larger substrings of two given DNA sequences builds an optimal alignment.

5.1 Terminology

Before presenting the core of the algorithm, consider some more terminology:

- Denote the first DNA sequence **`Dseq1`**, and the second DNA sequence **`Dseq2`**.
- Let **`P(Dseq,i)`** mean the prefix of DNA sequence **`Dseq`** of length **`i`**, i.e., the first **`i`** characters of **`Dseq`**.
- Let **`Dseq[i]`** mean the **`i`**th character of **`Dseq`**.
- **`Dseq[1]`** indicates the first character of **`Dseq`**, so that the prefix of length **`i`** will include **`Dseq[i]`**. Beware that this behavior is not natural in Java!

Remember that Java considers the first array index to be **`0`**, not **`1`**. Also, note that this approach allows a prefix of length **`0`**, which does not include any of the characters from the DNA string.

5.2 Initialization

You can easily align two empty strings – two equivalent strings have zero distance. Similarly, you can align any prefix with an empty string – just add gaps to the empty string until the size matches the non-empty prefix. These empty strings provide an initialization for the implementation.

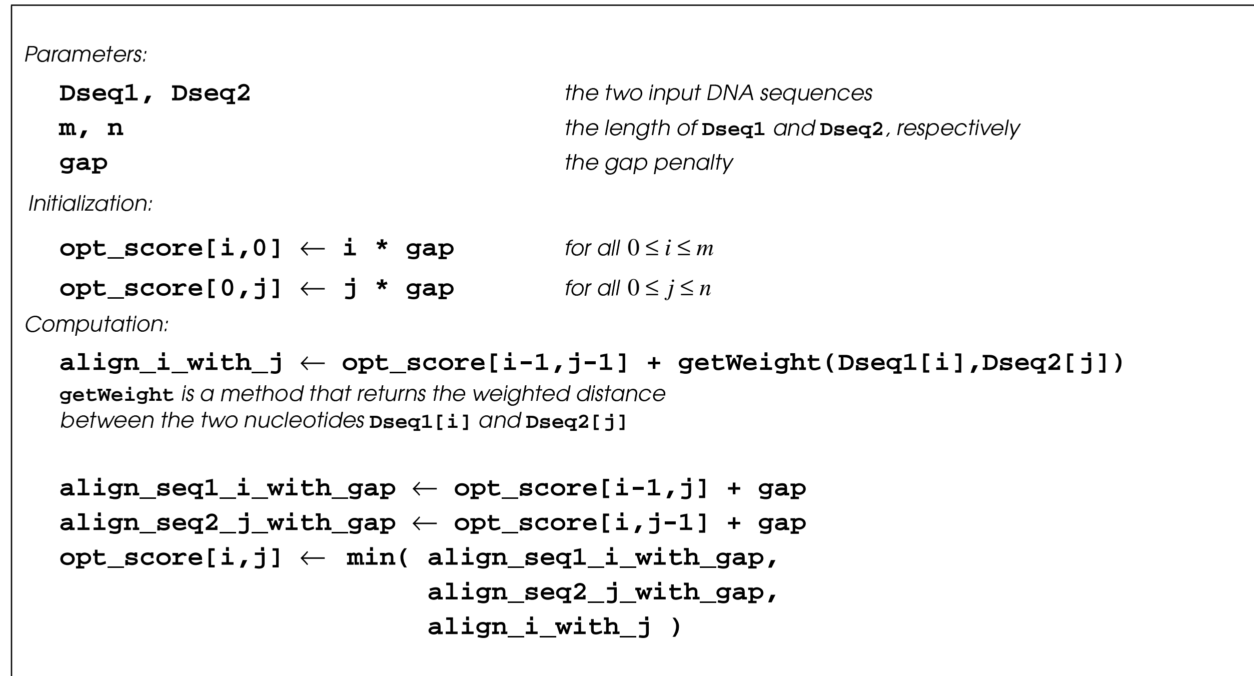
5.3 Choosing Optimal Alignment

You can determine the optimal alignment of **`P(Dseq1,i)`** with **`P(Dseq2,j)`**, where neither **`i`** nor **`j`** are **`0`**. If **`i`** and/or **`j`** are **`0`**, use the approach discussed in Section 5.2. Otherwise, choose the optimal alignment from the "best" of the following choices:

1. Start with the optimal alignment of **`P(Dseq1,i-1)`** with **`P(Dseq2,j-1)`**, then align **`Dseq1[i]`** with **`Dseq2[j]`**.
2. Start with the optimal alignment of **`P(Dseq1,i-1)`** with **`P(Dseq2,j)`**, then align **`Dseq1[i]`** with a gap.
3. Start with the optimal alignment of **`P(Dseq1,i)`** with **`P(Dseq2,j-1)`**, then align **`Dseq2[j]`** with a gap.

5.4 Algorithm

More specifically, you will compute a two-dimensional matrix which contains the optimal score for aligning prefixes of the two strings of increasing lengths.Use the weighted Hamming distance with a gap penalty called **`gap`**. Let **`opt_score`** store the matrix of scores. Each **`i`**,**`j`**th entry contains the optimal score for aligning the first **`i`** characters of the first string with the first **`j`** characters of the second string. Review the algorithm in Figure 9.



```
Parameters:
  Dseq1, Dseq2                       the two input DNA sequences
  m, n                               the length of Dseq1 and Dseq2, respectively
  gap                                the gap penalty
Initialization:
  opt_score[i,0] ← i * gap           for all 0 ≤ i ≤ m
  opt_score[0,j] ← j * gap           for all 0 ≤ j ≤ n
Computation:
  align_i_with_j ← opt_score[i-1,j-1] + getWeight(Dseq1[i],Dseq2[j])
  getWeight is a method that returns the weighted distance
  between the two nucleotides Dseq1[i] and Dseq2[j]

  align_seq1_i_with_gap ← opt_score[i-1,j] + gap
  align_seq2_j_with_gap ← opt_score[i,j-1] + gap
  opt_score[i,j] ← min( align_seq1_i_with_gap,
                        align_seq2_j_with_gap,
                        align_i_with_j )
```

Figure 9: Smith-Waterman Algorithm

Note that the last line takes the minimum of 3 possible scores.

6. Problems

You will develop and test software that implements the material reviewed in the previous sections. Use data encapsulation for all your code. Robust error checking is also required, i.e. check for, and recover from, possible errors.

6.1 Simple Distance

Write a program to compute the Hamming distance of two strings of equal size. You should do the following tasks:

- Write a class called **`DNA_sequence`**.
- Store the DNA sequence in your class as an array of **`char`**. This approach facilitates looking at one character at a time. You should be able to handle DNA sequences of up to 20 nucleotides long.
- Include methods to do the following:
    - Provide appropriate constructor(s)

- Input a DNA sequence from the user. Do not allow illegal characters, i.e., write code that allows only **`A`**, **`C`**, **`G`**, **`T`** as valid character input and makes the user re-enter a sequence, otherwise.

You should read your DNA sequence in as a **`String`**, then convert the string to the character array that stores the string. Hint: the **`String`** class has a method **`charAt`** which takes an integer parameter.

- Get the length of the sequence.
- Get the `i`th character of the sequence.
    - You want the first character of your DNA sequence to be returned when you ask for character `1`, not character `0`.
    - You need to follow this approach to allow the algorithm for minimum distance to work properly. Asking for character `0` is incorrect, so don't!
- Compute the Hamming distance. This method requires input DNA sequences to have the same length.
- Print the DNA sequence.

6.2 Weighted Distance

Write a program to compute the distance of two strings of equal size using the DNA substitution matrix shown in Figure 3. You will need to add the following to your **`DNA_sequence`** class:

- Read in the DNA substitution matrix. Implement the matrix as a two-dimensional array.
- Get the substitution cost for two bases, i.e., the cost from the substitution matrix for substituting one base for another.
- Compute the weighted distance.

This method requires that both input DNA sequences have the same length.

6.3 Minimum Distance

Write a program to compute the distance score for the optimal alignment of two strings which might have unequal size. You will need to add the following to your **`DNA_sequence`** class:

- Compute the minimum distance as described in Section 4.8.
- Use a gap penalty of 5.

6.4 Minimum Distance Alignment

Of course, you don't want just the optimal score. You also want to know what the optimal alignment looks like, i.e. an alignment that produces the optimal score. Tell how you would compute this optimal alignment from the score matrix computed above. Do *not* write a program for this. Hints: Along with the matrix **`opt_score`** you could write a second two-dimensional array which indicates which of the 3 possible choices was minimum. This method is sometimes called *traceback*.

6.5 Software Testing

Endeavor to develop robust software! Robust software will handle a wide variety of conditions. Ideally, your program should anticipate every possible input, but this lofty goal lacks practicality. Imagine how many possible pairs of DNA sequences may exist, even with limited length. For

example, a 10-element sequence of the 4 nucleotides has 4^{10} different combinations. A pair of these sequences generates $4^{10}4^{10}$, or over a trillion, pairs of 10-nucleotide sequences!

Of course, you do not need to test your software for all possible inputs. Instead, test your software for “reasonable” robustness with a few “typical” inputs and with some “atypical” inputs. Consider, also, “extreme” inputs, since code that works for some “typical” cases will likely break under “extreme” inputs. What does “extreme” mean? Part of your task consists of finding out, but consider these hints:

- two maximum length DNA strings
- two identical DNA strings
- DNA strings which differ only in the first position
- DNA strings which differ only in the last position
- illegal input of various types

You should test input from all of the above sample types, as well as others.

Also, how could you test your weighted distance method? Hint: Consider using the equal-weight matrix shown in Figure 4. How would the results compare for the Hamming distance and weighted distance methods?

6.6 Discussion

After developing, testing, and running your code, answer these questions. You *must* type all answers on a separate sheet of paper. Your answer for Section 5.4 may be on this same sheet.

- Describe your testing strategy for code from Sections 6.1, 6.2, and 6.3. You should describe the types of inputs you used.
- In what ways did your testing strategy differ for the different distance methods? Why?
- How did the results of the different distance methods vary? Why?

7. Submitting Your Work

7.1 Due Date

This assignment is due in lecture on Thursday, November 4, 1999. You may turn it in to a consultant before that date in the consulting room in Carpenter. Do not turn it in at Carpenter on the due date. Late programs will not be accepted.

7.2 Labeling Your Work

Always write your name, Cornell ID#, and the day/time/instructor for your section in the first comment of each program you hand in for credit. All solutions and commentary must be typed! If you wrote the program with a partner, turn in only one printout with your partner’s name and ID# in the comment, as well as your own. The comment must also include the section day/time/ instructor for the partner. The program will be returned to the first person listed. Sign your name(s) in the comment. Please staple the pages of your assignment together.

7.3 Grading

This assignment will be given two grades: the first based on correctness, the second on program organization and style. Each grade will be a 0, 1, or 2. Not only should your program work, but it should contain adequate comments to guide the reader who is interested in understanding it. The declaration of every significant variable should include a comment describing that variable. There should be appropriate comments in the code so the reader can see the structure of the program, but not so many that the program text is hard to read.

7.4 Academic Integrity

All work submitted should be your own or your partner's. The output submitted should exactly match the code submitted. Issues of academic integrity will be taken very seriously. See the Academic Integrity link from the CS100B web page for more information.

7.5 What to Hand In

Read Section 4 (Programs) of the *CS100 General Information* packet for explanations of requirements. For this assignment, staple the following sheets together:

- Cover sheet with your name(s), student ID(s), Section day&time, Section instructor, the title PROGRAMMING ASSIGNMENT 4, and the subtitle CS100B.
- The program(s) for Sections 6.1, 6.2, and 6.3. If you use the same code over, you do not need to print it twice, so long as your code can be clearly followed.
- Typed answers to questions in Sections 6.4 and 6.6. Don't forget to title this sheet appropriately.
- As specified in Section 6.5, the output of the code written for Sections 6.1, 6.2, and 6.3 with enough test cases to adequately test your software.

8. References

8.1 Cornell

- Cornell Genomics Initiative: *http://www.genomics.cornell.edu/*
- Institute for Computational Genomics (Cornell Theory Center) – *http://www.tc.cornell.edu/Research/ICG/*
- EE 445 – *http://www.ee.cornell.edu/study/courses.html*
- CS 482 – *http://www.cs.cornell.edu/cs482/*
- CS 681 – *http://www.cs.cornell.edu/cs681-fa99/*
- Agricultural genome databases (USDA) – *http://genome.cornell.edu/*
- Institute for Genomic Diversity – *http://www.cals.cornell.edu/dept/igd/*

8.2 Books

- Durbin, R., S. Eddy, A. Krogh, G. Mitchison (1998). *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*, Cambridge University Press, Cambridge.
- Farach-Colton, M., F.S. Roberts, M. Vingron, M. Waterman, Editors (1999). *Mathematical Support for Molecular Biology*, American Mathematical Society, Rhode Island.
- Gindikin, S, Editor (1992). *Mathematical Methods of Analysis of Biopolymer Sequences*, American Mathematical Society, Rhode Island.

- Waterman, M. S. (1995). *Introduction to Computational Biology: Maps, sequences and genomes*, Chapman & Hall, London.

8.3 Other Websites

- U.S. Human Genome Project – *http://www.ornl.gov/hgmis/*
- Biology Hypertextbook – *http://esg-www.mit.edu:8001/esgbio/chapters.html*
- National Center for Biotechnology Information – *http://www.ncbi.nlm.nih.gov/*
- PHYLIP (programs for phylogenetic analyses) – *http://bioweb.pasteur.fr/seqanal/phylogeny/phylip-uk.html*
- WebGene tools for prediction and analysis of protein-coding gene structure – *http://www.itba.mi.cnr.it/webgene/*
- Computational Genomics Group at the Sanger Center – *http://genomic.sanger.ac.uk/*
- Oak Ridge National Lab Computational Biosciences – *http://compbio.ornl.gov/*
- European Bioinformatics Institute – *http://www2.ebi.ac.uk/*

8.4 Cool ASCII Graphics

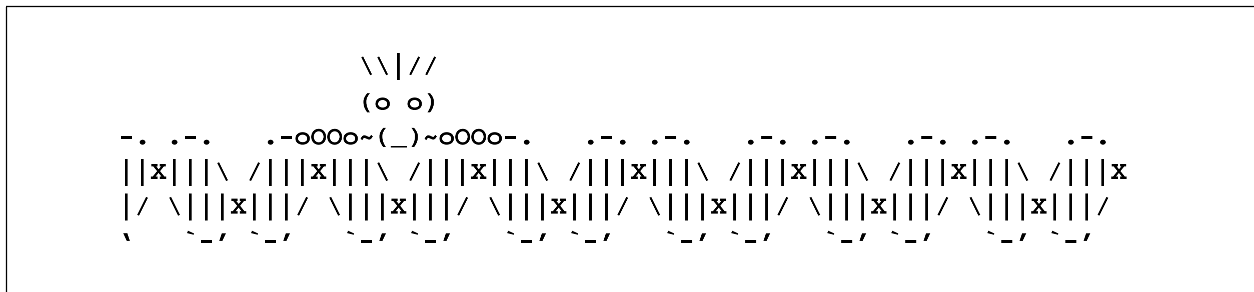


Figure 10: Cool ASCII Image from GENSCAN at MIT (*http://ccr-081.mit.edu/GENSCAN.html*)