Lab 7: Strings, Functions and Bioinformatics...

Due date: Monday, 7, beginning of the lab period.

Lab Assignment

Assignment Preparation

Lab type. This is a **pair programming lab**. You select your partner. As always, you are encouraged to select partners you have not worked with yet.

Purpose. The lab allows you to practice the use of strings in C programs. All lab assignments come from the area of bioinformatics, and all programs are designed to output information useful for biologists and biochemists. More about the bioinformatics aspects of this assignment below.

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties. Also note the the Lab 2 requirement for the content of the header comment in each file you submit applies **to each assignment** (lab, programming assignment, homework) in this course.

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

Program Outputs must co-incide. Any deviation in the output is subject to penalties. **PLEASE, USE BINARY EXECUTABLES PROVIDED BY THE INSTRUCTOR!**. The exception is made in case of floating point computations leading to differences in the last few decimal digits.

You can check whether or not a program produces correct output by running the `diff` command.

Please, make sure you test all your programs prior to submission! Feel free to test your programs on test cases you have invented on your own.

About this assignment

Among other disciplines Computer Science is unique in one important aspect. Computer Science (and, to large degree, Software Engineering,) study the use of computers for solving problems.

It is important to notice, that computer scientists and/or software engineers **rarely are the sources** of the problems themselves. More often than not, problems that can be solved via the use of computers (and via software development) come from other sciences, disciplines and fields.

As such, a majority of computer scientists, software developers and software engineers over course of their professional careers wind up working on problems in other subject areas: from business and finance, to agriculture, to arts, to natural sciences.

Working on problems of others and developing software solutions for them is what you will be doing a lot throughout your professional lives. Being able to do so successfully requires a certain set of skills, among which we find:

- Ability to understand problem domains. Be it warehousing or financial markets prediction or management of results of astronomical observations, if you have to develop software for it, you need to have understanding of important concepts from the problem area.
- Ability to work across disciplines. People who need software are likely NOT to be experts in computer science, software engineering or software development. They will rely on your expertise in that area. **However**, they are likely to be **experts in their fields**, and their knowledge and expertise is a crucial source of information for you during the software development process. To be efficient, you must be able to communicate well with experts from other fields.

  This is harder that one would believe. Occasionally, even the technical language you speak will be different. Details that are of importance to you will seem trivial to your customers, and vice versa – the customers will be expressing concerns over things that barely register on your radar.

It is never too early to start developing apporpriate skills. This lab introduces you to the art and the craft of solving problems from a domain you may be unfamiliar with. While the problems you are asked to solve are *simple*, they are **real**: actual bioinformatics software that is used by biochemists all over the world solves all of these problems all the time as part of performing more complex analyses.

Enjoy!

Bioinformatics in a Nutshell

Wikipedia[1] defines *bioinformatics* as follows:

> **Bioinformatics** is the application of information technology to the field of molecular biology.

The article continues:

> *Bioinformatics now entails the creation and advancement of databases, algorithms, computational and statistical techniques, and theory to solve formal and practical problems arising from the management and analysis of biological data.*

While some of the problems addressed in the field of bioinformatics require years of education and advanced degrees in **both** Biology and Computer Science to study and solve, a *wide range* of bioinformatics-related tasks can be performed using relatively simple programs. This lab offers five such problems for you to solve.

Bioinformatics: the Data

Among the wide range of bioinformatics applications, we will concentrate on the problems associated with the field of *genomics*, i.e., the study of genomes of organisms. In particular, one of the key subject matters of *genomics* is the discovery of the DNA Sequences of various organisms.

DNA a.k.a. **deoxiribonucleic acid** is a molecule that contains the *genetic instructions* used in the development and functioning of all known living organisms[2].

DNA is an extremely large and complex molecule. It consists of a large number of simpler *components* called **nucleotides**. A nucleotide molecule consists of a three components, two of which, *the phosphate* and *the sugar* have the same chemical structure for all nucleotides. The third component, **the base** is the one that distinguishes different nucleotides. There are four types of **nucleotide bases** present in DNA molecules:

- Adenine
- Cytosine
- Guanine
- Thymine

DNA Structure. One of the greatest scientific discoveries of the 20th century was the discovery of the structure of a DNA molecule. A DNA molecule is a **double helix** consisting of **two strands** of **nucleotides**[3]. One of the key properties of DNA is **base pairing**: the four nucleotides form specific pairings:

[1] http://en.wikipedia.org/wiki/Bioinformatics
[2] http://en.wikipedia.org/wiki/DNA
[3] In addition to the double helix structure, a DNA molecule may have a non-trivial three-dimensional structure, but the problems we will be studying in this assignment do not take this into account.

- If one strand has an Adenine base, then the corresponding position on the other strand is occupied by the Thymine base and vice versa.
- If one strand has a Cytosine, the the corresponding position on the other strand is occupied by Guanine and vice versa.

The **base pairing** property means that

A single DNA strand uniquely determines the full structure of a DNA molecule.

The Adenine – Thymine and Cytosine – Guanine pairs of nucleotide bases are called base pairs. The two strands are called complementary to each other.

DNA Sequences. A **DNA Sequence**, is a representation of a DNA molecule as a string. In particular, a single DNA sequence represents one strand of a DNA molecule. The four nucleotides found in DNA are encoded with four letters according the following coding table:

Nucleotide base	Letter
Adenine	A
Cytosine	C
Guanine	G
Thymine	T

Thus, a sequence of Adenine, Adenine, Cytosine, Thymine, Guanine, Thymine will be encoded as a DNA sequence "AACTGT".

DNA Sequence Directionality and reverse complement. A strand on a DNA molecule has **orientation**. The two ends of a strand have different chemical structure, and this allows researchers to label them and to represent all DNA sequences as being headed in the same direction.

The two ends of a DNA strand are referred to as **5′** ("five prime") and **3′** ("three prime") – an allusion to the chemical structures found on each end.

Direction of DNA sequences. All DNA Sequences are presented starting at the **5′** end and ending at the **3′** end.

DNA strands in a helix have opposite orientation. The two DNA strands have opposite orientation.

Figure 1 represents a DNA molecule fragment. The orientation of the light-gray DNA strand is left-to-right. The orientation of the dark-gray DNA strand is right-to-left. The the light-gray DNA strand contains the following sequence of nucleotides:

TCTTGATCGAAA

The dark-gray DNA strand contains the complementary sequence AGACTAGCTTT, **however**, when shown in this order, the sequence runs from the **3′** end to the **5′** end. To correctly represent this sequence, we must **reverse** it, i.e., read it from right to left. The resulting sequence of nucleotides,

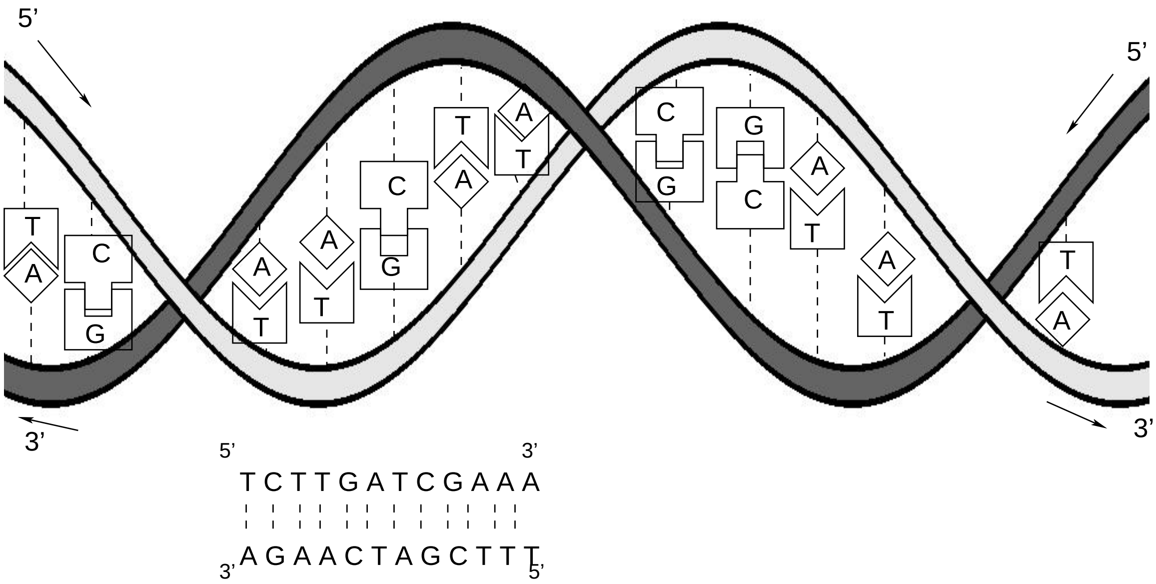

Figure 1: A double-helix DNA fragment and the DNA sequences representing it.

TTTCGATCAAGA

has the correct orientation and is called the **reverse complement** of the sequence TCTTGATCGAAA.

reverse complements. In general given a DNA sequence $S = L_1 L_2 \dots L_N$, where L_i represents one of the four letters A,T,C,G, its **reverse complement** is the DNA sequence that must occupy the corresponding bases on the other DNA strand, ordered from $\mathbf{5'}$ to $\mathbf{3'}$.

Remember that we have the following complement relationship between the nucleotides:

$$\boxed{\begin{array}{l} \mathsf{compl}(\mathsf{A}) = \mathsf{T} \\ \mathsf{compl}(\mathsf{T}) = \mathsf{A} \\ \mathsf{compl}(\mathsf{C}) = \mathsf{G} \\ \mathsf{compl}(\mathsf{G}) = \mathsf{C} \end{array}}$$

To construct an reverse complement of a sequence $S = L_1 L_2 \dots L_N$, we perform two steps:

1. **Step 1: complement.** Each letter L_i in the sequence is replaced with its complement $\mathsf{compl}(L_i)$. The resulting sequence is $S^C = \mathsf{compl}(L_1)\mathsf{compl}(L_2)\dots\mathsf{compl}(L_N)$.

Amino Acid	Three-letter code	One-letter code
Alanine	*Ala*	A
Arginine	*Arg*	R
Asparagine	*Asn*	N
Aspartic acid	*Asp*	D
Cysteine	*Cys*	C
Glutamic acid	*Glu*	E
Glutamine	*Gln*	Q
Glycine	*Gly*	G
Histidine	*His*	H
Isoleucine	*Ile*	I
Leucine	*Leu*	L
Lysine	*Lys*	K
Methionine	*Met*	M
Phenylalanine	*Phe*	F
Proline	*Pro*	P
Serine	*Ser*	S
Threonine	*Thr*	T
Tryptophan	*Trp*	W
Tyrosine	*Tyr*	Y
Valine	*Val*	V

Table 1: Amino Acids.

2. **Step 2: Reversion:** The sequence S^C is rewritten **from right to left** to form the **reverse complement** sequence

$$S^I = \mathsf{compl}(L_N)\mathsf{compl}(L_{N-1})\ldots\mathsf{compl}(L_2)\mathsf{compl}(L_1).$$

From Nucleotide Sequences to Amino Acids

Amino Acids. Amino acids are molecules that are used in building proteins. Amino acids are found in many other compounds in living organizms, and they play important roles in a variety of biochemical processes. More importantly (for us), amino acids form the next layer in the hierarchy of DNA encoding.

There are twenty amino acids. The table with their full names, three-letter abbreviations and (more importantly), single-letter codes is shown in Table 1.

Certain sequences of amino acids form **proteins**.

Genetic Code. In DNA molecules, **each triple of consecutive *nucleotides* encodes one *amino acid*** or serves as a special **marker**. The triple of nucleotides is called a codon.

The **translation** of the nucleotide triples (codones) into amino acid codes is called **the genetic code**.

With four nucleotides, there are **64 possible codons** that encode 20 amino acids and two special markers. A single amino acid can be encoded by multiple

First Nucleotide	Second Nucleotide			
	T	C	A	G
T	TTT → *Phe* F	TCT → *Ser* S	TAT → *Tyr* Y	TGT → *Cys* C
	TTC → *Phe* F	TCC → *Ser* S	TAC → *Tyr* Y	TGC → *Cys* C
	TTA → *Leu* L	TCA → *Ser* S	TAA → **Stop**	TGA → **Stop**
	TTG → *Leu* L	TCG → *Ser* S	TAG → **Stop**	TGG → *Trp* W
C	CTT → *Leu* L	CCT → *Pro* P	CAT → *His* H	CGT → *Arg* R
	CTC → *Leu* L	CCC → *Pro* P	CAC → *His* H	CGC → *Arg* R
	CTA → *Leu* L	CCA → *Pro* P	CAA → *Gln* Q	CGA → *Arg* R
	CTG → *Leu* L	CCG → *Pro* P	CAG → *Gln* Q	CGG → *Arg* R
A	ATT → *Ile* I	ACT → *Thr* T	AAT → *Asn* N	AGT → *Ser* S
	ATC → *Ile* I	ACC → *Thr* T	AAC → *Asn* N	ACC → *Ser* S
	ATA → *Ile* I	ACA → *Thr* T	AAA → *Lys* K	AGA → *Arg* G
	ATG → *Met*/**Start** M	ACG → *Thr* T	AAG → *Lys* K	AGG → *Arg* G
G	GTT → *Val* V	GCT → *Ala* A	GAT → *Asp* D	GGT → *Gly* G
	GTC → *Val* V	GCC → *Ala* A	GAC → *Asp* D	GGC → *Gly* G
	GTA → *Val* V	GCA → *Ala* A	GAA → *Glu* E	GGA → *Gly* G
	GTG → *Val* V	GCG → *Ala* A	GAG → *Glu* E	GGG → *Gly* G

Table 2: Genetic Code.

codons.

Table 2 shows the **genetic code**.

Start and Stop codons. Three codons, TAA, TAG and TGA do not encode any amino acids. Instead, these codons represent **the end of a protein molecule** encoded by a DNA sequence. They are called stop codons.

Additionally, the ATG codon encodes Methionine, the amino acid that serves as the **beginning of encodings of all proteins** within a DNA sequence. Whenever Methionine is found at the beginning of a protein, the ATG sequence encoding it is called a start codon.

Translation from Nucleotide sequences to Amino Acid sequences. Using the **genetic code** table, any DNA sequence written in the alphabet of nucleotides can be translated into a sequence of amino acids. To do this,

1. Start at the **5′** end of the DNA sequence.
2. For each triple of nucleotides, find, using the **genetic code table**, the matching amino acid (or start/stop codon).
3. Write out the amino acids and/or start/stop codons in a sequence following the **5′** to **3′** order.

Example. Consider the following DNA sequence:

TCTTGATCGAAA

We split it into triples of nucleotides:

TCT TGA TCG AAA

We then use the **genetic code** table, to substitute each triple with the matching amino acid:

TCT	TGA	TCG	AAA
S	**Stop**	S	K

Frames. Typically, DNA sequences represent portions of a DNA molecule. DNA molecules consist of millions of individual nucleotides, but current *DNA sequencing equipment* is capable of sequencing (i.e., discovering from a sample) only small "chunks" at a time. Given a DNA sequence in a nucleotide alphabet, there are three possibilities for translating it into a sequence of amino acids. These possibilities are called **frames**.

1. **Frame 1.** First codon starts at the first nucleotide.
2. **Frame 2.** First nucleotide is **ignored**; first codon starts at the second nucleotide.
3. **Frame 3.** First and second nucleotides are ignored; first codon start at the third nucleotide.

Intuitively, the frames can be explained as follows. Given a DNA sequence, we do not know whether it starts at the beginning of an amino acid encoding, or in the middle of it. There are three cases, and in order to translate a DNA fragment into a sequence of amino acids, we need to consider all possibilities. The three **frames** indicate where the first full amino acid of the fragment starts: at the beginning of the fragment, at the second nucleotide or at the third nucleodtide.

Example. Consider the following DNA sequence:

TCTTAATCGAATCGAT

This sequence can be split into codons in three different ways:

Frame 1: TCT TAA TCG AAT CGA T
Frame 2: T CTT AAT CGA ATC GAT
Frame 3: TC TTA ATC GAA TCG AT

In translating the DNA sequence in three frames, any nucleotides that are not part of a codon are ignored. The remaining codons are translated into amino acids using the **genetic code table**:

Frame 1:	TCT	TAA	TCG	AAT	CGA	T
	S	**Stop**	S	N	R	

Frame 2:	T	CTT	AAT	CGA	ATC	GAT
		L	N	R	I	D

Frame 3:	TC	TTA	ATC	GAA	TCG	AT
		L	I	E	S	

So, the same DNA sequence, may give rise to three different sequneces of amino acids: "S**Stop**SNR", "LNRID" and "LIES"[4], depending on which frame is actually correct.

Three more frames. DNA molecule has two strands. Given a DNA sequence that comes from one of the strands, it is possible that either this sequence, **or the corresponding sequence from the other strand** participates in describing a protein. Therefore, given a DNA sequence, we may need to convert both it, and **its reverse complement** to the amino acid sequences. Each of the two sequences (direct and the reverse complement) can be converted using three frames. Therefore, the total number of frames, i.e., plausible amino acid sequences represented by a DNA fragment is six: three for the fragment itself, and three for its **reverse complement**.

Example(continued). Consider again, the DNA sequence

TCTTAATCGAATCGAT

The example above shows the three frames of translation of this fragment into amino acid sequences. We now complete the full list of translations, by using the reverse complement of this fragment.

The reverse complement of the fragment is

Sequence:	TCTTAATCGAATCGAT
complement sequence:	AGAATTAGCTTAGCTA
reverse complement:	ATCGATTCGATTAAGA

The three frames for the reverse complement are:

Frame 4:	ATC GAT TCG ATT AAG A
Frame 5:	A TCG ATT CGA TTA AGA
Frame 6:	AT CGA TTC GAT TAA GA

The translation to the sequences of amino acids is:

Frame 4:	ATC	GAT	TCG	ATT	AAG	A
	I	D	S	I	K	
Frame 5:	A	TCG	ATT	CGA	TTA	AGA
		S	I	R	L	R
Frame 6:	AT	CGA	TTC	GAT	TAA	GA
		R	F	D	Stop	

[4] This was not intentional.

Combining the results of the two examples, here are the six amino acid sequences that may be encoded by the given DNA fragment:

S<Stop>SNR
LNRID
LIES
IDSIK
SIRLR
RFD<Stop>

Assignment

Your assignment for this lab is to develop software solutions for five problems from the field of bioinformatics.

Problem 1: The Dot Plot

One of the most common problems studied by genome scientists is that of matching different DNA sequence fragments. The fragments may have come from the same DNA molecule (e.g., the actual DNA sequencing problem is the problem of combining multiple short DNA fragments into a longer one by arranging them in an appropriate order) or from DNA molecules of different spcecies (e.g., when trying to find how similar the DNA of two different species is).

In both cases, a construct called a **dot plot** can be utilized by the genome scientists to obtain a quick visualization of the similarity between two DNA fragments.

Dot Plot. Given two DNA sequences $S_1 = l_1 \dots l_N$ and $S_2 = t_1 \dots t_M$, where l_i and t_j come from **either** the nucleodite alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$ **or** the amino acid alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}, \mathsf{K}, \mathsf{L}, \mathsf{M}, \mathsf{N}, \mathsf{P}, \mathsf{Q}, \mathsf{R}, \mathsf{S}, \mathsf{T}, \mathsf{V}, \mathsf{Y}, \mathsf{W}\}$, the **simple dot plot** of S_1 vs. S_2 is a two-dimensional table $D = \{d_{ij}\}$, $i = 1 \dots N$, $j = 1 \dots M$, where

$d_{ij} = 1$ if $l_i = t_j$;
$d_{ij} = 0$ otherwise.

Example. Consider two DNA sequences $S_1 =$ TCAAGA and $S_2 =$ TCGATT. Their **simple dot plot** is

	T	C	G	A	T	T
T	1	0	0	0	1	1
C	0	1	0	0	0	0
A	0	0	0	1	0	0
A	0	0	0	1	0	0
G	0	0	1	0	0	0
A	0	0	0	1	0	0

or

	T	C	G	A	T	T
T	1				1	1
C		1				
A				1		
A				1		
G			1			
A				1		

A general notion of a **dot plot** involves setting $d_{ij} = 1$ if **sequences around** l_i **and** t_j **are similar**. Examples or such *similarity are*:

- $l_{i-1}l_il_{i+1} = t_{j-1}t_jt_{j+1}$;
- $l_{i-2}l_{i-1}l_il_{i+1}l_{i+1} = t_{j-2}t_{j-1}t_jt_{j+1}t_{j+2}$;

Your task. Write two C programs that take as input two DNA sequences and output two different **dot plots**. Both programs should work in exactly the same manner, except for the part that determines whether or not $d_{ij} = 1$.

Your programs will output PPM files that contain the image of the appropriate dot plot.

Program names. Name your programs `dotplot01.c` and `dotplot02.c`.

Input. The input to your dot plot programs will consist of three numbers followed by two strings, each on its separate line. The first two numbers represent the lengths of the two input strings. The third number represents the *size in pixels of a single "dot" on the dot plot image.* The strings can be either in the nucleotide or amino acid alphabet, but both strings will always be in **the same alphabet**.

A sample input is shown below:

```
10 12 3
ATCTTGCAAG
ATATCAACGA
```

Input restrictions. The size of each string cannot be larger than 400. Let the three numeric inputs be called L1, L2 and DotSize. Then, the following restrictions hold:

$$\mathsf{L1} \cdot \mathsf{DotSize} \leq 1200.$$

$$\mathsf{L2} \cdot \mathsf{DotSize} \leq 1200.$$

Your programs should check the inputs for compliance withthe restrictions above. If the inputs violate the restrictions, your programs shall terminate with an appropriate error message.

Output. If the three numeric inputs are given names L1, L2 and DotSize, then, the output of your programs is a PPM image with $\mathsf{L1} \cdot \mathsf{DotSize}$ rows and $\mathsf{L2} \cdot \mathsf{DotSize}$ columns. This image will represent the L1 rows and L2 columns of the dot plot between the two input strings. Each row/column intersection of the dot plot is represented by a $\mathsf{DotSize} \times \mathsf{DotSize}$ square area. The key to building the image is in the rule below:

> If your program detects a match between position i in the first input string and position j of the second input string, then the PPM image area corresponding to the d_{ij} position of the dot plot shall be painted. You can choose the paint color (red, blue, dark green, will all work).
>
> If your program does not detect a match, the area corresponding to d_{ij} shall be painted `white`.

Match detection. dotplot01.c The first program shall detect a match, i.e., set $d_{ij} = 1$ (i.e., paint the appropriate area of the PPM image) **iff**

> *the ith character of the first input string is the same as the jth character of the second input string.*

Match detection. dotplot02. The second program shall detect a match, i.e. set $d_{ij} = 1$ (i.e., paint the appropriate area of the PPM image) **iff**

> *the three-character sequence starting at position $i-1$ of the string* ***exactly matches*** *the three character sequence starting at position $j-1$ of the second string.*

Functional decomposition. Your programs will declare, implement and use the following functions:

- `int match(char s1[], char s2[], int i, int j).` This function checks whether the two strings generate a dot-plot match at the location i, j. The check is done according to the rules outlined above and is different for each program.
- `int drawDot(char image[][1200][3], int i, int j, int dotSize).` This function takes as input the image array, and paints a "dot" representing a match on the dot plot for the dot plot position d_{ij}, with the size of the dot being `dotSize`. (Please note, that in your program the constants 1200 and 3 must be `#define`d).

You are encouraged to use other functions in your program as you see fit. If both `dotplot01.c` and `dotplot02.c` use the same functions, implement them only once, put them in a `dotplot.h` header file[5] and `#include "dotplot.h"` in both programs.

Comments. Note, that you will not know the exact image size at compile time, as it is determined by the input parameters. You do know the maximum possible image size, 1200×1200, so this allows you to create appropriate arrays for storing the image data. However, only part of the array will be used to store image data. Therefore, you need to be careful when outputing the colors from the array to form the PPM image. Note, also, that because of this, it is not feasible for you to use the `image.h` library of functions. However, you can implement some of them (e.g., `blankImage()`, `putHeader()`, `drawImage()`) for this application.

Problem 2: The Reverse Complement

Write a program that takes as input a single DNA sequence in nucleotide alphabet and outputs the **reverse complement** of this string.

Program name. Name your program `icomplement.c`.

[5]In this case, you can simply implement the functions directly in the header file.

Input. The input to the program is a single string consisting of letters `A`, `T`, `C` and `G`. The string will be terminated with a `newline` character.

The maximum length of an input string is 400.

Output. The output of the program is a single string (terminated with a new line character) representing the **reverse complement** of the input string.

Functions and header files. Create a header file `genomics.h`. Use this file to declare **and define** the following functions:

- `char compl(char c)`. This function takes as input a character in the nucleotide alphabet {A,T,C,G} and returns its complement.
- `void complement(char s[], char compl[], int length)`. This function takes as input two strings, `s`, as well as *initially empty* string `compl`, and the number `length`, representing the length of the string `s`. The function puts the complement of string `s` (i.e., the string that consists of the complements of individual characters of `s` in the same order) into string `compl`.
- `void reverse(char s[], char inv[], int length)`. This function takes as input two strings, `s`, and an *initially empty* string `inv`, as well as the number `length`, representing the length of the string `s`. The function **inverts** `s` and put the reverse of `s` into string `inv`.

Your `main()` function for this program must produce the reverse complement of the input string using the functions described above. (It goes without saying that your program should `#include "genomics.h"`).

Example. Consider the following input string:

```
ATTCCATGG
```

The output of your program will be:

```
CCATGGAAT
```

Problem 3: Conversion of Nucleotide Sequences into Amino Acid Sequences

Your third program, will apply the **genetic code** to convert a DNA sequence in a nucleotide alphabet into **all possible** amino acid sequences it represents.

Program name. Name your program `dna2amino.c`

Input. The input to this program is the same as the input to the `icomplement.c` program: a `newline`-terminated string of characters in a nucleotide alphabet.

Output. The output of the program is six `newline`-terminated strings in the amino acid alphabet. Each string represents a translation of the input DNA sequence into an amino acid sequence on **one of the six possible frames**. The frames are output in the order from Frame 1 (actual DNA sequence, starting at the first nucleotide) to Frame 6 (reverse coplement sequence, starting at the third nucleotide).

No other output is to be generated.

In the output, the **stop codons** that encodes nucleodtide sequences TAA, TAG and TGA is to be represented using the character `'#'`.

Example. If the input to the `icomplement.c` program is

```
TCTTAATCGAATCGAT
```

then the program shall output the following:

```
S#SNR
LNRID
LIES
IDSIK
SIRLR
RFD#
```

(see excamples above for the translation).

Functions and header files. Your program shall use the same `genomics.h` header file as the `icomplement.c` program does. It will wind up using some of the same functions. In addition, more functions will be added to the `genomics.h` library.

Because your program needs to construct the reverse complement of the input in order to determine frames 4–6 of the output, your program will use the `complement()` `reverse()` (and `compl()`) functions defined in `genomics.h` for the `icomplement.c` program.

Additionally, the following functions shall be declared and defined in `genomics.h`.

- `char geneticCode(char c1, char c2, char c3).` This program takes as input three `char` values representing a triple of nucleotides in a DNA sequence (i.e., a codon). It outputs the *1-letter symbol* representing an amino acid, encoded by the input nucleotides, **according to the genetic code table**.

  For the start codon `ATG`, `geneticCode()` shall return `'M'` (note, that this is the only codon encoding Methionine as well, and therefore, there will not be an ambiguity about it).

  For the stop codons `TAA`, `TAG` and `TGA`, `geneticCode()` shall return `'#'`.

  For all other nucleotide triples, the output of `geneticCode()` is uniquely determined by the **genetic code table** (Table 2).

- `convert2Amino(char s[], char amino[], int length, int frame).` This function takes as input a string `s` containing a nucleotide sequence of length `length`, an *initially empty* string `amino`, and the conversion frame represented by the `frame` parameter.

> The function translates the input nucleotide sequence into an amino acid sequence in the specified frame. The result of the translation is stored in the `amino` parameter.
>
> **Note:** `frame` can take values of `1`, `2` and `3`, i.e., this function translates into the amino acid sequence **only the actual DNA sequence presented**, and not the reverse complement.

Problem 4: Finding Palindromes

In linguistics, a **palindrome** is a string (sentence of text) that spells the same when read from left-to-right and from right-to-left. "Meaningful" palindromes exist in essentialle every human language. Some examples of English palindromes are:

```
eye
Anna
tenet
rotator
redivider
madam
madam I'm Adam
never odd or even
murder for a jar of red rum
rats live on no evil star
God saw I was dog
Doc, note: I dissent. A fast never prevents a fatness. I diet on cod.
```

Palidromes in genomics. In genomics, a version of palindromes plays an important role in determining the 3D structure of a DNA molecule.

> A DNA palindrome is a sequence of characters in the nucleotide alphabet $\{A, C, G, T\}$, such that **it is equal to its reverse complement.**

Example. Consider a sequence

AGTACT

It's complement is TCATGA. It's reverse complement is AGTACT, i.e., the original string.

The biological significance of the palindromes in DNA sequences is illustrated in Figures 2 and 3. Essentially, because if a palindrom is present in a DNA sequence on some strand, the DNA strand may become *entangled* in this place, as the nucleotides in the palindrome sequence may get "paired" with their complements in the palindrome sequence, rather than with the nucleotides on the second strand. Such entanglements, called hairpins are common in DNA sequences, and biochemists want to be able to find where they occur in the DNA.

Your assignment. You will write a program that takes as input a DNA sequence string in nucleotide alphabet, looks for palindromes in it, and reports the longest palindrome it finds.

To simplify the problem, you will be looking for only one type of palindrome in the input strings: the palindromes *formed around the center of the string.*

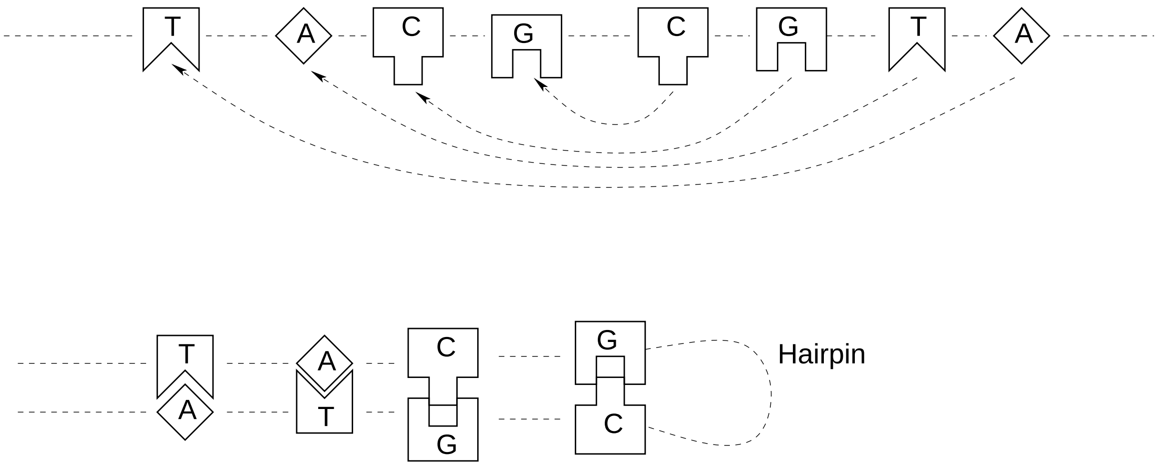

Figure 2: Palindromes in DNA sequences form *hairpins.*

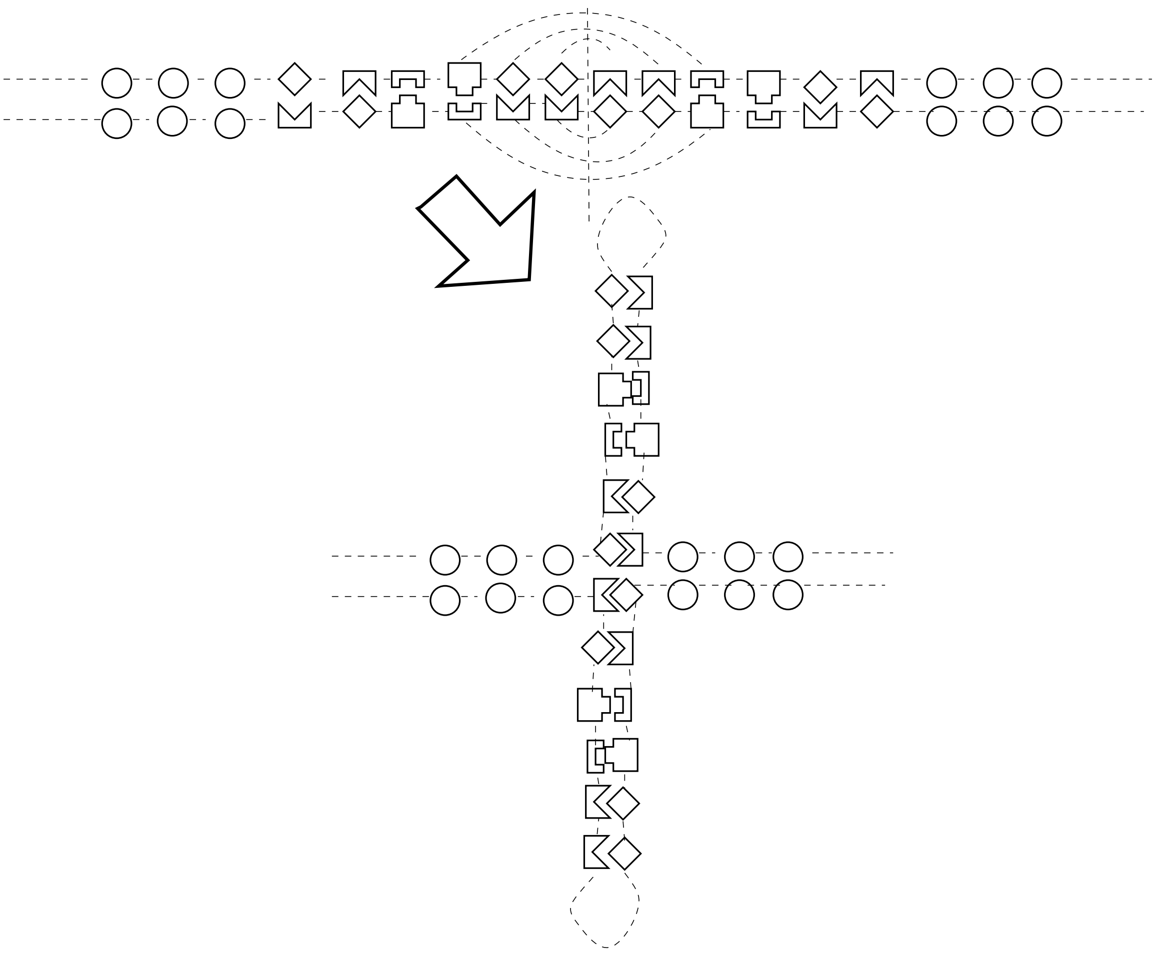

Figure 3: Palindromes in DNA sequences form *hairpins.* (part 2)

Program name. Name your program `palindrome.c`.

Input. The input to this program is the same as the input to `icomplement.c` and `dna2amino.c`.

Output. The program shall output the following information:

- Length of the longest palindrome found.
- The palindrome itself, printed out in the following way:
  - The first line of the output, contains the entire palindrome.
  - The second line of the output contains the reverse of the second half of the palindrome, formatted so that each character is directly below its complement in the first half.

For example, let the input to the program be

```
AATCGATT
```

The longest palindrom, around the center of this sequence is `ATCGAT`. The output of the program will be:

```
Longest palindrome: 6 characters long

ATCGAT
TAG
```

Finding palindromes. A sequence can have palindromes anywhere. You are only responsible for detecting the longest palindrome centered at the midpoint of the input string. There are two cases to consider:

1. Input string has **even length**. In this case, the mid-point of the string lies *between* two nucleotides. You detect the palindrome by comparing the appropriate characters on both sides of the midpoint.

   For example, if the input string `AATCGATT` is stored in a string variable

   ```
   char s[8] = {'A','A','T','C','G','A','T','T'}
   ```

   the midpoint occurs between `s[3]` and `s[4]`: the fourth and the fifth character in the string.

2. Input string has **odd length**. In this case, the mid-point falls onto a nucleotide. Your program will omit this nucleotide and will start matching the nucleotides immediately before and after it, effectively excluding the central nucleotide from consideration.

   For, example, the sequence `AAT` will be considered a palindrome by your program. The midpoint, the second `A` nucleotide is not used, but the nucleotides on both sides of it, `A` and `T` are compared to each other. By the same token, `ATT`, `ACT` and `AGT` are all palindromes of length 3, and `ATTAT`, `ATAAT`, `ATCAT` and `ATGAT` are palindromes of size 5.

Functional decomposition. Feel free to use functions from the `genomics.h` library that you have developed for `icomplement.c` and `dna2amino.c`. There is no requirement for you to create any specific functions for this program, however, you may choose to do so. If you do, *declare and define* these functions in the `genomics.h` header file.

Problem 5: Multiple Sequence representation

The final problem of the assignment finds you working on PPM images again. You will write a program that takes as input 10 DNA sequences in the nucleotide alphabet, and builds an image representing the composite of these sequences.

The problem of comparing multiple DNA sequences to each other occurs in genomics quite commonly. Scientists study similar DNA locations for multiple related species and want to find out what the differences are. Consider, for example the following ten short DNA fragments:

```
ATATC
ATTTC
AATTC
ATGTC
ATTTC
ATCTC
AAAAC
AAGAC
ATATC
AATTC
```

All fragments have matching first and fifth nucleotides. In 80% of fragments the fourth nucleotide is T, while in the remaining 20%, it is A. The second nucleotide is evenly split between T and A, while the third nucleotide shows the least amount of consistency with the T: 40%, A: 30%, G: 20% and C: 10% distribution.

Your program will accept 10 input DNA sequences **of the same length**, compute the distribution of nucleotides in each position, and output a PPM image that represents, in color, the computed distributions.

Program name. Name your program seq2ppm.c.

Input. The input to this program is ten `newline`-terminated DNA strings of the same length. (Your program is not responsible for verifying that all strings have the same length, but it is responsible for determining the lengnth of the strings). The maximum length of a string is 400.

Output. The output of your program is a PPM image. The image shall have 400 rows. The number of columns in the image is determined by your program based on the input size — see considerations below. The top and the bottom portions of the image shall be painted some neutral color — white or light gray. The central 100 rows of the image will contain the color-coded representation of the input sequences. See specifications below to learn how to color this region.

Image size determination. The first task of your program is to determine the size of your image. The color-coded representation of the DNA sequences is similar in nature to the USA elections histogram that you have built in lab 5. Essentially, each nucleotide position in the DNA sequences is going to be represented horizontally, by a few columns of pixels. The key to determining the size (i.e., the total number of columns) of the image is to establish how many pixels to use to represent a single nucleotide position.

Use the following rules for computation.

- The max. number of columns is 1200. Therefore, if you have a input strings of size 400, you can afford 3 pixels per nucleotide base.
- The minimum number of columns is 400 (to obtain a square image). Therefore, if your input strings have size 1, you will use 400 pixels per nucleotide base.
- For as long as you can afford more than 20 pixels per nucleotide base, your image can be 400 pixels in width.
- If you cannot afford 20 pixels per nucleotide base with 400 pixels, but you can – with 1200 pixels, your image width shall be $20 \cdot \mathsf{Length}$, where Length is the length of the input strings.
- If $20 \cdot \mathsf{Length} > 1200$, then you reserve $PPN = floor\left(\frac{1200}{\mathsf{Length}}\right)$ pixels per nucleotide, and the width of the image is $PPN \cdot \mathsf{Length}$.

Declare and define a function `int getWidth(int length)` which takes as input the length of the input strings and outputs the width of the image.

Each position of the input string will be represented by a $100 \times PPN$ rectangle on the PPM image, where PPN, i.e., *pixels per nucleotide* is the number of columns used to represent each nucleotide in the sequence. We refer to these $100 \times PPN$ bloacks as *nucleotide blocks* below.

Statistics. For each position in the given DNA sequences (from 1 to Length), you need to compute the percentage of occurrence of every nucleotide. You can elect to either store these distributions in an array (or collection of arrays), or to simply employ a group of variables to store the distribution for the current position on the string.

Based on the distribution, your program will determine the color of each nucleotide block.

Color determination. The color of each nucleotide block is determined as follows. We start by asigning a base color to each nucleotide:

Nucleotide	Abbreviation	Color	RGB	Name
Adenine	A	green	(0, 255, 0)	AColor
Cytosine	C	blue	(0, 0, 255)	CColor
Guanine	G	black	(0,0,0)	GColor
Thymine	T	red	(255,0,0)	TColor

The color of a nucleotide box is the **weighted average** of the four base colors, where the weights come from the distribution of nucleotides at the given position.

More formally, consider position i of the input strings, and let P1, P2, P3, P4 be, respectively the percentages of Adenine, Cytosine, Guanine and Thymine occurrences this position. Then the color of position i, denoted Color(i) is computed as follows:

$$\text{Color(i)} = P1 \cdot \text{AColor} + P2 \cdot \text{CColor} + P3 \cdot \text{GColor} + P4 \cdot \text{TColor}.$$

This computation is performed separately for every color component of the color.

Example. Consider the ten input strings in the example above. Let us compute the color of the second nucleotide box.

The distribution of nucleotides at position 2 is:

Nucleotide	Percent of occurrence
A	$50\% \sim 0.5$
C	0%
G	0%
T	$50\% \sim 0.5$

The red component of the color for position 2 will be:

$$\text{Color(i)(Red)} = \text{AColor[red]}*0.5+\text{CColor[red]}*0+\text{GColor[red]}*0+\text{TColor[red]}*0.5 = 0*0.5+0*0+0*0+255*0.5 = 127.$$

(remember, this is an integer computation, so $255 * 0.5 = 127$.)

Similarly, the green and the blue components are computed as follows:

$$\text{Color(i)(Green)} = \text{AColor[green]}*0.5+\text{CColor[green]}*0+\text{GColor[green]}*0+\text{TColor[green]}*0.5 = 0*0.5+0*0+0*0+0*0.5 = 0.$$

$$\text{Color(i)(Blue)} = \text{AColor[blue]}*0.5+\text{CColor[blue]}*0+\text{GColor[blue]}*0+\text{TColor[blue]}*0.5 = 255*0.5+0*0+0*0+0*0.5 = 127.$$

Thus, Color(i) = (127,0,127), i.e., dark(ish) magenta.

Functions. Besides the function listed above, your program should use functions to do the following tasks:

- Compute the distribution of nucleotides given a position in the string.
- Compute the color of the nucleotide box given the distribution of nucleotides at its position.

Unlike other cases, *you get to declare and define your own functions* here. Depending on your approach to solving this problem, you may wind up selecting different input and output parameters for these functions.

Submission.

Files to submit. Each pair submits one set of files from one account. The following files are mandatory:

`team.txt`, `dotplot01.c`, `dotplot02.c`, `icomplement.c`, `dna2amino.c`, `palindrome.c`, `seq2ppm.c`, `dotplot.h`, `genomics.h`.

If you have developed other files (extra .h file, for example), submit them as well.

`team.txt` file shall contain the name of the team and the names of all team members in each pair, and the Cal Poly IDs of each. E.g, if I were on the team with Dr. John Bellardo, my `team.txt` file would be

```
Go, Poly!
John Bellardo, bellardo
Alex Dekhtyar, dekhtyar
```

Submit `team.txt` as soon as you form your group.

Files can be submitted one-by-one, or all-at-once.

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows:

Section 01:

```
> handin dekhtyar-grader lab07-01  <your files go here>
```

Section 09:

```
> handin dekhtyar lab07-09  <your files go here>
```

`handin` is set to stop accepting submissions 24 hours after the due time.

Testing

Tests are available on the course web page. Please note, that test input files for the `icompliment.c`, `dna2amino.c` and `palindrome.c` are the same. While some data is "toy", to help you debug your programs, a portion of the test cases will contain real and long(ish) DNA sequences that may present actual interest to the students of bioinformatics courses.