Lab 12-2: Strings, Functions and Bioinformatics. . .

Due date: Monday, March 5, beginning of the lab period.

Lab Assignment

Assignment Preparation

This is Part 2 of **Lab 12.** You continue working on it with you **Lab 12-1** partner.

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.

Assignment

Please refer to the **Lab 12-1** handout for the background information on bioinformatics.

The second part of the lab consists of three programs.

Problem 4: GC-percentage computation.

GC-percentage. Given a DNA sequence $S = s_1 \dots s_N$, a GC-percentage of S is the percentage of characters in S that are either `'G'` or `'C'`.

For example, is $S =$`ATGGTTCTTA`, the *GC-percentage* of S is 0.3 (or 30%), as S contains two `'G'`s and one `'C'` out of 10 characters.

GC content refers to the percentage of DNA bases that contain either G (guanine) or C (cytosine). DNA is the genetic material of life that encodes instructions for making building blocks of living cells proteins. DNA is a linear polymer that contains information in the order/sequence of bases. Four bases found in DNA are G, C, T, and A. The frequency of bases in a DNA polymer varies among different organisms. GC content of entire genome (all DNA of one organism or species) can be used to distinguish two closely related species or to find a foreign gene in a genome (a gene that jumped species; for example, viruses can transfer genes from one organism/species to another).

In addition, GC content can vary between different functional regions within one genome. When analyzing a new genome, finding regions of high or low GC percentage can point to particularly interesting regions (e.g. protein coding genes, origin of replication). Most molecular biologists use GC content of small regions of DNA to choose primers (molecular tools) for DNA amplification by PCR.

Task. Write a program, `gcp.c`, that reads a DNA string in nucleotide alphabet from standard input, computes the GC-percentage of the string and prints it out.

For input we will use the same files (`dna-testNN` and `dna-fragmentNN`) as we used for the `Lab 12-1` programs.

Your program shall read the string, output it (prefacing it with the text `"Sequence :  "` (notice the two spaces). On the next line, your program shall print the text `"GC-content:  "` followed by the computed GC-percentage number, formatted to display two (2) digits after the decimal point.

Example. Here is an example of how the output of your program must look.

```
$cat dna-test04
ATGAAACCCGGGTGA
```

```
$ gcp < dna-test04
Sequence  : ATGAAACCCGGGTGA
GC-content: 53.33
```

Functions. You shall add a new function to you `genomics.c` function library (see **Lab 12-1** handout). As usual, the appropriate function declaration also needs to be added to the `genomics.h` file. The function prototype is

```
void updateGCCount(char s[], int * gc, int * at);
```

The function takes as inputs three parameters. The first one, `char s[]` is the DNA string whose GC-percentage needs to be computed. The second and third parameters are references to integer variables. One, `gc` represents the *current* count of `'G'` and `'C'` occurrences in the observed DNA strings. The other one, `at` represents the *current* count of `'A'` and `'T'` occurrences in the observed DNA strings.

The function shall sweep through the contents of `s` and update the `gc` and `at` counts appropriately.

Note: This function is a bit of an overkill for the task at hand. However, in many scenarios biologists need to find a combined GC-percentage of multiple DNA strings (e.g., DNA strings representing two different chromosomes of the same organism). `updateGCCount()` is handy in such situations. (Also, keeping both GC- and AT- counts is unnecessary, but convenient on occasion).

You must write the `int main()` of your GC-percentage computation program in a way that uses (calls) `updateGCCount()`.

Problem 5: Consensus Sequence.

This is the only program in this lab that will take a different type of input.

Definition. Given a set of DNA strings of the same length in the nucleotide alphabet (usually representing the same DNA fragment for different related species/organisms), a *consensus sequence* is a string in a nucleotide alphabet that has at each position the nucleotide that shows in the plurality of DNA strings at that position.

Example. Consider the following collection of three DNA strings:

```
GTTAC
GATAA
GAATC
```

The string `GATAC` is the consensus sequence for this set of strings. `G` appears in all three strings in position 1, `A` appears twice in position 2, `T` appears twice in position 3, `A` appears twice in position 4, and `C` appears twice in position 5.

Some sets of strings can have multiple consensus sequences. E.g., the set of four strings:

```
TTGCA
TTGGA
```

```
TTGCT
TTGGT
```

has four consensus sequences: `TTGCA`, `TTGCT`, `TTGGCT` and `TTGGT` - in the last two positions we have even number of `G` and `C`, and `A` and `T` nucleotides respectively.

Task. Your task is to write a program, `consensus.c`, which reads 10 nucleotide sequences from the standard input (all sequences will have the same length) and outputs a consensus sequence for these 10 sequences.

In case when there are multiple possible consensus sequences, your program shall report *any one* consensus sequence.

Example. — Here is an example of running this program.

```
$ cat seq-test02
ATCGATCGAT
ATCGATCGAT
ATCGATCGAT
ATCGATGGAT
ATCGATGGAT
ATGGATGGAA
ATGGATAGAA
ATGGATAGAA
ATGGATTGAA
ATGGATTGAA

$ consensus <seq-test02
ATCGATCGAT
```

Notes. Feel free to implement this program in any way you find convenient. There are no required functions for this program. You are allowed to define functions that help you simplify your `int main()`, but, please, declare and define them directly in the `consensus.c` file.

Program 6: Naïve Gene Predictor. **Gene prediction** is one of the most popular tasks biologists engage in when analyzing DNA sequences. Most of DNA does not carry any information. The important part of the DNA, coding sequences or genes contain information that is later transcribed into RNA and translated into proteins. Being able to find genes, therefore, is important.

A computer algorithm that analyzes a DNA sequence and finds DNA fragments that look like they can be genes is called a gene predictor. A variety of gene predictors has been developed in the past 15 years. Some of them are rather complex, but some use very simple observations about the nature of genes to base their predictions on.

Your task is to create a very simple gene predictor. It will use the following information about the structure of genes[1]:

[1]Technically, this gene predictor will only work for the DNA of prokaryotes — simple single-cell organisms like bacteria.

1. Most genes start with a Methionine (M) amino acid. The *codon* for Methionine is `ATG`. When Methionine serves as the beginning of a gene, it is commonly referred to as the `start codon`.
2. Genes end on one of the so called `stop codons`. There are three stop codons in the genetic code: `TAA`, `TAG` and `TGA`.
3. Genes can occur on either the positive (top) strand of the DNA molecule, or on the negative (bottom) strand.
4. Genes can occur on any reading frame of either the top or the bottom strand.
5. The `start` and `stop codons` of a gene must be on the same reading frame.

Write a program `geneSearch.c` which works as follows. It starts by reading a DNA string in a nucelotide alphabet from standard inputs. The program shall print the string read, and then, for each frame of the given DNA sequence, it shall perform the search for a suitable candidate gene using the observations stated above. Essentially, your program shall look for an occurrence of the `start codon ATG` on a given reading frame. Once found, your program shall look for the occurrence of one of the `stop codons` further "downstream" from the occurrence of the start codon.

Since the reading frame may contain multiple occurrences of both the `start` and the `stop codons`, use the following approach:

- Try to make your gene sequence as long as possible.
- For this, choose the earliest occurrence of the start codon,
- ...and the latest occurrence of the stop codon on the given reading frame.

This method allows you to predict at most one gene per reading frame for up to six predicted genes total.

Your program shall first search for the predicted genes on the three reading frames of the positive strand, and then – on the three reading frames of the negative strand (recall – the negative strand is the inverse complement of the positive strand).

For each prediction, your program shall output the frame number (from 0 to 5, as it so happens in this program), the starting and ending positions of the predicted gene sequence and the actual sequence of nucleotides forming the predicted gene. Both the start codon and the stop codon must be included.

Examples. Your code shall provide the same ouput as the instructor's code. Please refer to the examples below (as well as instructor's executables) for exact output format. Please notice the spacing and other features of the output.

In the first example, the gene is on the first reading frame (a.k.a., frame 0) of the positive strand (start and stop codons are highlighted in the input string for your convenience, the highlights are not part of the terminal session).

```
$ cat genetest01
AAAATGAAACCCTAGAACCC
   ^^^      ^^^

$ geneSearch < genetest01
```

```
AAAATGAAACCCTAGAACCC
Positive strand:
=> Frame 0:  [3, 15]
           ATGAAACCCTAG
Negative strand:
```

In the second example, the start and stop codon are found on the first frame of the negative strand (frame 3):

```
$ cat genetest03
AAACTAGGGTTTCCCTTTCATAAA
   ^^^            ^^^
$ geneSearch < genetest03
AAACTAGGGTTTCCCTTTCATAAA
Positive strand:
Negative strand:
=> Frame 3:  [20, 2]
           ATGAAAGGGAAACCCTAG
```

As the negative strand is read right to left, it will contain the inverse compliment of `CAT` (see underlined above), which is `ATG`, i.e. our start codon on the first negative reading frame. Further down on the same reading frame, there is an inverse complement of `CTA`, whose inverse complement is `TAG`, i.e., the `stop codon`.

Note how the output is organized in this case. The coordinates of the predicted gene are shown in the inverted order, but they reference the positive strand, hence the start codon has a higer position

Our final example shows how the predictor handles multiple start and stop codons on the same reading frame.

```
$ cat genetest04
AATGTTTATGGGGTAGCCCTAATTTGGG
 ---   ~~~   ~~~    ---
$ geneSearch < genetest04
AATGTTTATGGGGTAGCCCTAATTTGGG
Positive strand:
=> Frame 1:  [1, 22]
           ATGTTTATGGGGTAGCCCTAA
Negative strand:
```

Here, two start codons and two stop codons are seen on the second positive frame (frame 1). The codons underlined with `"---"` are used for gene prediction by the program.

Functions. My program uses a number of functions already defined in `genomics.c`, but does not introduce any new functions. Your program can include functions for certain parts of the task. Put all functions you define for this program (if any) in the `geneSearch.c` file.

Testing. The inputs to your program are the same `dna-testNN` and `dna-fragmentNN` test files used for other **Lab 12** programs. Additionally, we constructed a few extra tests in the files names `genetestNN`.

Submission.

Files to submit. Each pair submits one set of files from one account. In addition to files specified in **Lab 12-1** description, each team needs to submit the following files:

The following files are mandatory:

`gcp.c`, `consensus.c`, `geneSearch.c`

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows. Ssh to `unix1`, `unix2`, `unix3`, or `unix4`. The `handin` command is:

```
> handin dekhtyar lab12 <your files go here>
```

Notice that there is only one `handin` directory for both parts of **Lab 12**.