DNA Analyzer Project: Requirements

***** Note: This is taken as-is from Moodle. It might require reshape *****

DNA Analyzer System

Goal

The goal of the system is to load, analyze, manipulate and save **DNA sequences**.

Description

DNA sequences are composed of four types of **nucleotides**;
The nucleotides are marked **A** (Adenine), **G** (Guanine), **C** (Cytosine) and **T** (Thymine).
A full DNA molecule usually consists of two strands, connected to each other in **base-pair** connections: **A**s with **T**s, and **C**s with **G**s.
Three successive nucleotides generate a **codon**, which might be chemically "read" in various ways.

The system will interact with the user through a **CLI** (Command Line Interface) that uses the standard I/O. Using that interface, the user will be able to load DNA sequences from files, to analyze them, to manipulate them (e.g., by extracting sequence slices or by modifying the sequence), and to store modified sequences and reports.

The commands are detailed in the following sections.

The Command Line Interface (CLI)

The command line interface allows interaction with the user. Throughout that interface, the user can enter their input and see the application's output. The prompt of the CLI is usually `> cmd >>>`; it might change when special type of input is required.

There are several groups of commands, which are detailed in the next chapters.

Sequences

The application's most important elements are **sequences**. These are DNA sequences (composed of the four characters `A`, `C`, `T` and `G`). Each sequence that is held in the app's memory has a **name** and **sequence number**. When referring to a sequence in commands, unless otherwise defined, it is possible to refer it either by its **ID** or by its **name**.

Reference to the **sequence number** is done using the ***hash* character** `#`

> *For example:*
>
> `#1` means sequence number *1*.
>
> `#107` means sequence number *107*.

Reference to the **sequence name** is done using the ***at* character** `@`

> *For example:*
>
> `@short-seq` refers to the sequence named *short-seq*.
>
> `@dolly-dna` refers to the sequence named *dolly-dna*.

Files

The application stores and reads DNA sequences from files. It will use raw DNA data, that is, files that contain the four letters and nothing else.

The default file extension for that app is `.rawdna`.

Common CLI markings

- **[argument]** - Words starting with "**[**", ending with "**]**" represent **optional** arguments.
- **<argument>** - Words starting with "**<**", ending with "**>**" represent **required** arguments.
- **arg1|arg2** - Pipe sign ("|") between words represents that **each one of them can be used.**

Example using bash command `cp`:

```
cp [-r|-f] <source> [<source2> [<source3> ...]] <destination>
```

Means that:

- `cp` can be used with flag -r or with flag -f, but they both are optional.
- After the flags (if they exist) must be the source files. At least one, but can be many.
- At the end must be the destination.

Legal examples:

```
cp x.cpp dna.cpp
cp -r project dnaProject
cp -f y.cpp dna.cpp
cp dna.cpp dna.h dnaProject
```

Sequence Creation Commands

The following commands are being used to **generate new sequences**:

new

```
> cmd >>> new <sequence> [@<sequence_name>]
```

Creates a new sequence, as described by the followed sequence.

If the `@<sequence_name>` is used, then this will be the name of the new sequence. Otherwise, a default name will be provided - `seq1` (or `seq2`, `seq3` and so on, if the name is already taken).
The new sequence, its name and its number (internal ID, starting with 1) are printed.

For example:

```
> cmd >>> new ATACTGCCTGAATAC @short_seq
```

will create that sequence;
if this is the first sequence, it will be numbered "1" and the following will be printed:

```
[1] short_seq: ATACTGCCTGAATAC
```

load

```
> cmd >>> load <file_name>  [@<sequence_name>]
```

loads the sequence from the file, assigns it with a number (ID) and a default name, if one was not provided (based on the file name, possibly postfixed with a number if the name already exists), and prints it.

For example:

```
> cmd >>> load my_dna_seq.rawdna
```

will load the sequence from the file `my_dna_seq.rawdna` and will print its assigned ID, its name and the sequence (no more than 40 chars; If there are more in the sequence, it prints the first 32, then an ellipsis, and then the last three ones). So, a typical output might be:

```
[14] my_dna_seq: AACGTTTTTGAACACCAGTCAACAACTAGCCA...TTG
```

dup

```
> cmd >>> dup <seq> [@<new_seq_name>]
```

duplicates the sequence.

If a new name is not provided, then it will be based on the name of `<seq>`, suffixed by `_1` (or `_2`, `_3`, ... if the name is already taken).

For example:

```
> cmd >>> dup #22
```

will result in

```
[23] conseq_1_1: ATACTGCCTGAATACAGCATAGCATTGCCT
```

Sequence Manipulation Commands

The following commands **manipulate existing sequences**:

Their default behavior is to modify the source sequence in-place (that is, the original ID and name of the sequence are left the same, only its content is modified).

- If a colon `:` appears after the command's argument, then the original sequence is left untouched, and a new sequence is generated with the manipulation results.
- If an argument of the form `@<new_seq_name>` is provided after the colon, then this is the name of the new sequence.
- Otherwise, if `@@` instead, then the name of the new sequence is automatically generated by the app.

Each command might generate a different default name.

When a sequence is required as a source, both ID (`#<seq_id>`) or name (`@seq_name`) are acceptable, unless otherwise defined.

slice

```
> cmd >>> slice <seq> <from_ind> <to_ind> [: [@<new_seq_name>|@@]]
```

Slices the sequence, so that starts in `<from_ind>` (0-based index) and ends in `<to_ind` (*inclusive*).

If `@<new_seq_name` is provided, the results will create a new sequence with that name. If `@@` is provided, the results will create a new sequence with auto-generate name, based on the name of the original sequence, with the suffix `_s1` (or, if that name is already occupied, with the suffix `_s2`, and so on).

For example:

Assuming that the former `short_seq`'s ID is 1, the following command:

```
> cmd >>> slice #1 4 9
```

will change the sequence to `TGCCT` (Letters at indices 4,5,6,7,8) and print the output:

```
[1] short_seq: TGCCT
```

If `@@` was provided to the same command, then sequence 1 would have not changed, and a new sequence would have been generated instead.

A typical call, then, might look like:

```
> cmd >>> slice #1 4 8 : @@
[19] short_seq_s1: TGCCT
```

replace

```
> cmd >>> replace <seq> <index> <new_letter> [: [@<new_seq_name>|@@]]
```

replaces the letter in the (0-based) index of `<seq>` by `<new_letter>`.

If `@<new_seq_name>` is provided, the original sequence is left untouched and the result is put in a newly created sequence with that name.
If `@@` is provided, the name is based on the original sequence, with the suffix `_r1` (or, if that name is already existing, `_r2` and so on).

The command might get more than a single replacement. In that case, after `<seq>` there will be more than one pair of `<index>` and `<new_letter>`.

For example:

```
> cmd >>> replace @short_seq_s1 0 A 3 A : @repl_seq
```

will result in the following output:

```
[20] repl_seq: AGCAT
```

concat

```
> cmd >>> concat <seq_1> <seq_2> [: [@<new_seq_name>|@@]]
```

concatenates `<seq_2>` at the end of `seq_1`.

If the colon `:` syntax is used, then both `seq_1` and `seq_2` are left untouched, and the result is being put in a new sequence, named either as provided by `<new_seq_name>`.
If the `@@` syntax is used, automatically generated to include the name of `<seq_1>`, an underscore, then the name of `seq_2` and finally the suffix `_c1` (or `_c2`, `_c3`, ..., if the name is already taken).

For example:

The result of:

```
> cmd >>> concat #19 @repl_seq : @@
```

will result in a new sequence and will print:

```
[21] short_seq_s1_repl_seq_c1: TGCCTAGCAT
```

Concatenation is also possible for more than two sequences.
In this case, the default name for the new sequence, in case that: `@@` is provided, will be `conseq_1` (or `conseq_2` and so on, if that name is already taken).

For example:

The following command:

```
> cmd >>> concat #1 #20 #20 #19 : @@
```

will result the following:

```
[22] conseq_1: ATACTGCCTGAATACAGCATAGCATTGCCT
```

If no "colon argument" is used, then the command modifies only the first sequence (e.g., in the last example - #1) and the other sequences are left untouched.

pair

```
> cmd >>> pair <seq> [: [@<new_seq_name>|@@]]
```

Create sequence `<seq>` with its pair sequence, that is, each `T` is replaced by an `A` (and vice versa), and each `C` is replaced by a `G` (and vice versa).

If an auto-generate name is required (`@@` is used), then it will be the original name, suffixed with `_p1` (or `_p2`, `_p3`, ... if the name is already taken).

For example:

```
> cmd >>> pair #21 : @@
```

will produce:

```
[24] short_seq_s1_repl_seq_c1_p1: ACGGATCGTA
```

Sequence Management Commands

This is a list of commands that **manage existing sequences** (without manipulating them).

rename

```
> cmd >>> rename <seq> @<new_name>
```

renames the name of the sequence to the new name.
If that name is already taken, an explanatory error message is printed.

del

```
> cmd >>> del <seq>
```

deletes that sequence.

Before deleting it, the user is asked to confirm that:
Confirmation is done by entering `y` or `Y`, Entering `n` or `N` cancels the deletion. Any other input will result in a message that asks the user again to confirm the deletion. Once confirmed, the sequence is deleted and a message is printed. Otherwise, a cancellation message is printed.

So, a deletion scenario might look like:

```
> cmd >>> del #23
Do you really want to delete conseq_1_1: ATACTGCCTGAATACAGCATAGCATTGCCT?
Please confirm by 'y' or 'Y', or cancel by 'n' or 'N'.
> confirm >>> x
You have typed an invalid response. Please either confirm by 'y'/'Y', or
cancel by 'n'/'N'.
> confirm >>> Y
Deleted: [23] conseq_1_1: ATACTGCCTGAATACAGCATAGCATTGCCT
```

reenum

```
> cmd >>> reenum
```

re-enumerates all the sequences, so that their numbers are 1..*n*, where *n* is the number of sequences. The original order is kept.

save

```
> cmd >>> save <seq> [<filename>]
```

saves sequence `<seq>` to a file.

If `<filename>` is not provided, the sequence name is being used.
The filename is suffixed by `.rawdna`.

Sequence Analysis Commands

len

```
> cmd >>> len <seq_id>
```

prints the length of the sequence.

> *For example:*
> If sequence #34 is `AAATGTGATG`, then it will look like this:

```
> cmd >>> len #34
10
```

find

The `find` command finds a sub-sequence within a sequence.

It has two flavors:

1. Takes an **expressed sub-sequence**:

```
> cmd >>> find <seq> <expressed_sub_seq>
```

> returns the (0-based) index of the first appearance of `<expressed_sub_seq>` in the sequence `<seq>`.

Thus, for example:

If sequence #11 is `AACCTTGGAATTCCGGAA` and we are looking for the sub-sequence `GG`, it will look like:

```
> cmd >>> find #11 GG
7
```

2. Refers an **existing sub-sequence**:

```
> cmd >>> find <seq_to_find_in> <seq_to_be_found>
```

Thus, for example:

If seq #11 is as appears above, and sequence #25 is `CTTGGA`, it might look like:

```
> cmd >>> find #11 #25
4
```

count

`count` works in a similar way to `find`, only it returns the number of instances of the sub-sequence within the larger sequence.

Like `find`, it has two flavors, one that takes an **expressed sub-sequence**, and one that refers an **existing sub-sequence**:

```
> cmd >>> count <seq> <expressed_sub_seq>
> cmd >>> count <seq_to_find_in> <seq_to_be_found>
```

findall

```
> cmd >>> findall <seq> <expressed_sub_seq>
```

```
> cmd >>> findall <seq_to_find_in> <seq_to_be_found>
```

work very similar to `find`, only they return all the indices where the sub-sequence appears.

> *Thus, for example:*
>
> Using the above sequence for sequence #11, it might look like:

```
> cmd >>> findall #11 GA
8 16
> cmd >>> findall #11 AA
1 9 17
```

Control Commands

The following commands control the application, rather than working on sequences:

help

```
> cmd >>> help [<command>]
```

If `<command>` is not provided, `help` lists all the available commands.
If a valid command is provided, it shows a short explanation of it.
If the value provided is not a valid command, a relevant message is printed.

list

```
> cmd >>> list
```

shows all the sequences in the system, by order.

For each sequence, it shows its number, its name and the sequence itself (up to 40 chars).
A sequence that is *connected to a file* (that is, either was loaded from a file, or was saved to a file at least once) and was not manipulated since last save, is prefixed by a `-` sign.
A sequence that was manipulated after last save, is prefixed by a `*` sign.
A sequence that is not connected to any file yet is prefixed by a `o` sign.

For example, a typical `list` output might be:

```
> cmd >>> list
- [1] short_seq: ATACTGCCTGAATAC
- [14] my_dna_seq: AACGTTTTTGAACACCAGTCAACAACTAGCCA...TTG
* [21] short_seq_s1_repl_seq_c1: TGCCTAGCAT
* [22] conseq_1: ATACTGCCTGAATACAGCATAGCATTGCCT
o [24] short_seq_s1_repl_seq_c1_p1: ACGGATCGTA
```

show

```
> cmd >>>show `<seq>` [<num_chars>]
```

Shows the sequence: Its ID, its name, its status and the sequence itself.

The status is either `up to date` (was not changed since last save), `modified` (changed since last save) or `new` (not yet connected to a file).
The ID, name and status are printed in the first line.
Then, the sequence itself is printed in the next lines, no more than 99 chars per line.
If `<num_chars>` is provided, then this is the number of chars to print (if the sequence is longer than that). Otherwise, `<num_chars>` defaults to 99.

Thus, it might look like:

```
> cmd >>> show #7 204
[7] lab_test_seq_feb_2015 modified
CCGTGCCTAGCATACGGATCGTATGCCTAGCATACTAGCATCCGTGCCTAGCATACGGATCGTATGCCTAGC
ATACTAGCATCCGTGCCTAGCATACGG
CGTATGCCTAGCATCCCGGATCGTATACGGATCGTAGTGCCTAGCATACGGAGCCTAGCATACTAGCATCCG
TGCCTAGCATTACTAGCATCCGTGCCT
AGCATA
```

quit

```
> cmd >>> quit
```

prints a goodbye message and exists the application.

If not all the sequences are `up to date`, it first requests for a confirmation.

So, for example, it might look like:

```
> cmd >>> quit
There are 3 modified and 2 new sequences. Are you sure you want to quit?
Please confirm by 'y' or 'Y', or cancel by 'n' or 'N'.
> confirm >>> qwerty
You have typed an invalid response. Please either confirm by 'y'/'Y', or
cancel by 'n'/'N'.
> confirm >>> Y
Thank you for using Dnalanyzer.
Goodbye!
```

Command Results Labels

Most of the CLI commands print to the screen their results. Usually, those results are one of:

1. A sequence (for example, in `load`, `dup` or `slice`)
2. An number (for example, in `find` or `count`)
3. A series of numbers (for example, in `findall`)

In the case of sequences result, they are also being named and added to the list of active sequences. In the same way, it is possible to save the results of *analysis commands* in a list of active labels. The syntax for that is very similar to that of sequence results - just add a `: @<label>` at the end of the command, and the result will be saved in that name.

For example:

```
> cmd >>> find #11 GG : @gg_ind
gg_ind: 7
```

or:

```
> cmd >>> findall #11 AA : @aa_locations
aa_locations: [1 9 17]
```

Show Labels

The command `labels` shows all the existing labels,

For example:

```
> cmd >>> labels
gg_ind: 7
aa_locations: [1 9 17]
```

Label Actions

Labels can be manipulated and analyzed using the following commands:

calc

The command `calc` allows simple calculations with labels.

The following calculations are supported:

Addition

Using the `+` operator, the sum of two operands is calculated. Each of the operands might be either a number or a label (the result, of course, might itself be labeled).

For example:

```
> cmd >>> calc 7 + @gg_ind
18
> cmd >>> calc @something + @gg_ind : @the_res
the_res: 25
```

Subtraction

The `-` operator works in a very similar way, but for subtraction.

For example:

```
> cmd >>> calc @the_res - @gg_ind
14
> cmd >>> calc 3 - 5 : @result
result: -2
```

Extreme Values

max

The command `max` finds the maximum number out of the given ones. It might get any number of arguments.

For example:

```
> cmd >>> max 7 12 @the_res 5 @gg_ind
25
```

min

The command `min` does the same, but for the minimum value.

For example:

```
> cmd >>> min @gg_ind 0 : @start_index
start_index: 0
```

Series Actions

On labels that are series of numbers, the following commands are defined:

size

Shows the size of the series, i.e. the number of elements in it.

For example:

```
> cmd >>> size @all_markers
37
> cmd >>> size @all_markers : @num_markers
num_markers: 37
```

atindex

Shows the *i*-th element in the series (zero based).

> *For example:*
>
> If the first number in the series `all_markers` is 5, then this will show `5`:

```
> cmd >>> atindex @all_markers 0
5
```

> or

```
> cmd >>> calc @num_markers - 1 : @last_marker
last_marker: 36
> cmd >>> atindex @all_markers @last_marker
12
```

Using References to Last Results

In order to make commands easier to use, the last result of each operation might be used in a special way.

Last Sequence Identifier

Whenever a sequence is required, the symbol `##` might be used to represent the last created (or manipulated) sequence.

For example:

```
> cmd >>> slice #1 4 9
[1] short_seq: TGCCT
> cmd >>> len ##
5
```

Last Label Identifier

Whenever a label is required, the symbol `__` might be used to represent the last calculated result.

For example:

```
> cmd >>> slice #1 4 9
[1] short_seq: TGCCT
> cmd >>> len ##
5
> cmd >>> calc __ - 2
3
> cmd >>> atindex ##
```

Batch Commands

Batch Creation

```
> cmd >>>batch <batch_name>
```

Batch mode allows the user to define a series of actions that will take place one after another.

In order to define a batch, the user enters the command `batch`, followed by the name of that new batch. Then, it enters into **batch mode**, where any command is not being activated immediately, but rather, entered into the batch.

The command `end` ends the batch mode.

For example:

```
> cmd >>> batch my_batch
> batch >>> load basil_dna.rawdna @basil
> batch >>> pair basil @basil_pair
> batch >>> find ## TGATTCTC : @start_slice
> batch >>> find ## TTTTAAAATTTTCCCC
> batch >>> calc __ + 4
> batch >>> slice @basil_pair @start_slice __ @basil_interesting_part
> batch >>> save ##
> batch >>> end
> cmd >>>
```

This batch (when run) will load a sequence from the file `basil_dna.rawdna` and will keep it under the name `basil`.
Then, it will create its pair and keep it under the name `basil_pair`.
After that, it will slice it from the first index of the sub-sequence `TGATTCTC` to four nucleodites after the first index of the sub-sequence `TTTTAAAATTTTCCCC` (that is, it will include the first four `T` of that sequence). That slice will be kept under the name

`basil_interesting_part` and then will be saved to disk using that name (with the `.rawdna` suffix).
When the batch mode ends, the batch is added to the list of active batches - nothing is being activated yet.

Running Batches

The command `run <@batchname>` runs a batch, that is, executes it as if the commands were entered manually.

Listing Batches

The command `batchlist` shows a list of all the batch names.

Showing a Batch

The command `batchshow <@batch_name>` shows the content of that batch.

Saving Batches

Saving a batch is done using the command `batchsave`, followed by the filename.

If the filename is omitted, then the batch name is being used as the filename, with the suffix `.dnabatch`
The batch is saved exactly as it is written in the CLI (without the prompt, of course).

> *Thus, for example:*

The above script will be saved as:

```
load basil_dna.rawdna @basil
pair basil @basil_pair
find ## TGATTCTC : @start_slice
find ## TTTTAAAATTTTCCCC
calc __ + 4
slice @basil_pair @start_slice __ @basil_interesting_part
save ##
```

Loading Batches

Loading a batch is done using the command `batchload`, followed by the filename to be loaded.

The loaded batch will have that name (without the `.dnabatch` suffix, if appears). If the command is followed by `: @<batch_name>`, then it will be kept as `batch_name`.