Biocomputing II – Coursework guidance

I refer to the database layer as DB, the middle (business logic) layer as BL and the front end graphical interface with CGI scripts as (FE).

Standardized file headers

All code (including HTML, CSS and test code) should have a header documenting the function of the code, how to use it (if it's run from the command line), who wrote it, the date, the version number and a change log.

Standardized function headers

Trying to document calls to subroutines outside the API may not be maintainable. Do it in the code and generate a document automatically if you wish.

All subroutines should contain a docstring that explains how to use the routine and is formatted in a standard way.

For example:

```
Input:   STRING      select   SQL Select statement
Input:   STRING      from     SQL From statement
Return:  STRINGARRAY          Array of result tuples
```

There are programs such as doxygen and Pydoc that can generate web pages of documentation if you format parameters in a standardized way:

- https://stackoverflow.com/questions/34331088/how-to-comment-parameters-for-pydoc
- https://www.youtube.com/watch?v=COaSNOrFGQM

Config file

Avoid all hardcoding (absolute URLs, DB contact info, etc) – place all of this in a config file (e.g. config.py or use a text file – in which case you need some code to read the file). The same config file should be accessed by all layers of the code.

Code layers

The database end needs to...

1. Parse and read the data – you must parse this yourself and not make use of libraries such as BioPython. We want you to experience the complexity of parsing these types of data.
2. Store the data into the database – you only need to store the data required for the front end to be functional.
3. Provide an access layer that allows basic information to be extracted from the database without knowing the schema.

The middle layer needs to:

1. Identify where the coding regions are in the genomic DNA
2. Generate the coding DNA sequence from the DNA
3. Align the protein sequence translation with the DNA coding sequence
4. Identify RE sites

5. Provide a list of available REs to the front end
6. Identify whether an RE has restriction sites within the coding region
7. Count codon usage in a gene
8. Calculate codon usage across all coding regions (perhaps use the subroutine developed for counting codon usage in a single gene) - store this somewhere; perhaps back in the database or in a text file.
9. Extract information from the database layer such as the complete gene list or individual gene information. (Some of these may simply be wrappers to database layer code)

Note that there should not be any HTML in the output from this layer.

Note also that some BL functions may simply call the DB function and return the data unmodified.

The front layer needs to:

1. Take data from the middle layer and format it for presentation.
2. Allow searches to be specified through forms
3. Perform all HTML/CSS formatting, generate tables, etc. (Note that libraries such as Bootstrap, JQuery, etc., **are allowed** to make your web site more attractive.)
4. **Ensure all HTML and CSS is validated!** validator.w3.org jigsaw.w3.org/css-validator

Using GitHub

- One member of the team should create a GitHub repository, clone the demonstration repository as explained in 'Practicalities' and grant read/write access to other members of the team.
- Each team member should then clone the repository
  `git clone git@github.com:`*`RepositoryOwner/RepositoryName`*`.git`
  where *`RepositoryOwner`* is the GitHub account name of the person who created the repository and *`RepositoryName`* is the name of the shared repository.
- Each team member will then create their own directory within the repository
- Remember to add each file that you create to the repository with
  `git add `*`filename`*
- Remember to commit changes you make to your files:
  `git commit -a -m "`*`A comment describing the change`*`"`
  Typically you should be doing this every time you have a small addition to the code that works, or that you expect to work. It is much better to commit small changes very regularly than big changes every day or two, so **I would expect this to be done every 30 minutes to 1 hour.**
- Remember to push the changes back to GitHub.
  `git push`
  Do this *at least* once a day, but there is no harm in doing it every time you do a commit.
- At the start of each day of work, pull down any changes that may have been made by other members of the team.
  `git pull`
  You might want to do this more often – especially if you have been discussing changes that they plan to make and, even more importantly, if they have informed you of API changes that will affect your code.

Work plan

Week 0 is the day the project is given to you.

- Week 0-1: Everyone – take a look at the data files, their documentation and the project requirements. Use grep or write small scripts to see what happens in the data. Meet to discuss findings including problems identified and what is needed for the requirements. Create your shared GitHub repository based on the provided sample.
- Week 1-2: Everyone – consider what needs to be stored in the database to satisfy the front end requirements. Consider any problems in the data and consider suitable database designs. Meet to start thinking about the APIs. i.e. If you are the front end developer think about what you need from the BL layer (you should just be formatting data and sending off queries) and discuss a suitable API with the BL developer. If you are the business layer developer, work out what you are going to need from the database - you will not write any SQL or any HTML. If you are the database developer start to think about a suitable database design to provide what the BL will need and discuss the API with the BL developer.
- Week 2-3: Discuss and fix the specification for your APIs with the understanding it *may* change later – but you wish to avoid this…
    - Document your API with agreed specifications of how to call the functions and agreed specifications of what each function will do. Ensure this goes into GitHub.
    - **Immediately implement dummy versions of all the APIs which return some data of the correct form.** The parameters passed into the calls may be ignored by this dummy code – you just want to ensure that you have routines that return an answer of some sort in the agreed format. (Note that the DB developer does **not** need to have a database at this stage – just return some static data.) Again see the sample code to see how gthis should be done and ensure this code goes into GitHub. **We cannot emphasize strongly enough how important this is!!!!!**

At this stage the FE should be able to call the BL API and get something that could be displayed. e.g. if the FE wants a list of gene IDs with protein names etc., then a set of text should be returned by the middle layer in the agreed data structures. Ideally this should be picked up from dummy data provided by the DB API.

Similarly, the DB API should be implemented with dummy code returning what you have agreed, in the format you have agreed. *The underlying database design does not matter* – it is up to the DB layer designer to provide 'wrappers' that encapsulate the necessary SQL to extract the required data from the database and return the data in the agreed format for use by the middle layer. If the database format changes, this will have *no effect* on the BL developer as the DB API, as seen by the BL, will not change.

You must have the API agreed and dummy code implemented for the project discussion in Week 3. You may think this is unnecessary and a waste of time – you'd rather dive in and write proper code. However, every group who has not done this regrets it later!

Each team member should then be able to work largely independently, modifying their dummy code into real code and pushing their code to GitHub as soon as it is working. **Announce this to the rest of the team so they can test their code.** Thus the dummy code gradually evolves into the real code and by testing your code all the time there should be no need for any last minute integration of the different layers as you will be working with the code from different layers all the time, but at the same time, you are not dependent on other layers since there will be dummy routines present in case anybody is unable to complete their pieces of code. As the code gets updated, the functionality will appear.

The team will clearly need regular meetings to discuss any needed changes to the APIs (should be minimal if you have spent enough time discussing it at the start!) and will need to come together in the final couple of weeks to sort out any remaining issues.

If changes are needed to the API (for example, the FE ends up needing something from the BL that hasn't been specified), this must be communicated to the other developer and the documentation updated. Make sure all team members know and that they all do a `git pull` and **test their code** once such changes are made.

Tips Based on Previous Years

1. Again, **define your BL and DB APIs and provide a dummy implementation** based around the demonstration git repository as your first step. Document your APIs and what is expected from the code. You **will** regret it if you don't do this.
2. **Ensure your API documentation is sufficiently detailed**. Don't just say something like "Returns a sequence" – instead it should be something like "Returns a protein sequence as a string" or "Returns a DNA sequence in FASTA format as a string".
3. **Your API documentation should be in docstrings in your (dummy) code**. There are automated programs for extracting this, or you could write a simply Python script to extract the documentation; you could use Markdown format (as used for README.md files in Git) to format the documentation within your code. This saves problems with keeping 2 sets of documentation in sync and up to date if there are any changes.
4. **Someone needs to take charge as project manager.** If people in the team aren't pulling their weight, please report back to Andrew or Adrian so we can try to find out what's going on.
5. **Agree interim deadline**s within the team and stick to them. Make sure everyone starts early and isn't leaving everything until the last 2 weeks.
6. **One person needs to create the GitHub repository** and then give the other members of the team read/write access so everyone is committing to the same repository.
7. **Use GitHub** from the start – you will regret it if you don't! We have had people lose code they had written by accidentally deleting or over-writing it. If you use git/github properly this is never a problem as you can always get older code back. Do a commit every time you change something (probably every 30mins or so). Push to GitHub when something works (or at least once a day).
8. Make sure you **use a configuration file** rather than hard-coding things like URLs and database connection information into your Python. This makes it much easier to port your code from machine to machine (including from where you may be working at home to Hope). Suppose Andrew and Adrian are working on different machines and you need to install on Hope. Create three different config files: `config_andrew.py`, `config_adrian.py`, `config_hope.py` and store these in GitHub. Do not store `config.py` in GitHub, but create that as a symbolic link to the relevant config file on each machine.
9. **Don't make things too complicated** – at least in the first instance. This applies especially to the front end. If you are planning on using JavaScript and AJAX, make sure you have a working version using the CGI script to generate your HTML before you try to do anything more complex.
10. **There is no need to use XML or JSON for this project** – no data markup is needed. The 3-layer design is simply implemented as a) CGI scripts, b) a Python module that provides

the API for the BL layer, c) a Python module that provides the API for the DB access layer. The only place where you *might* use XML or JSON is in the CGI should you wish to provide a more complex JavaScript-based graphical interface that uses AJAX.

11. The BL code does need to highlight things within sequences – where the coding regions are in the DNA and where the restriction sites are. The BL code must *not* do this by using special characters that change colours or by including HTML markup. You could do this using some simple token which is then replaced by appropriate markup in the FE CGI script. For example:
    `ATCGTCGGTTATGT{GCTGGCAT}GTTGCGTGTGTGCTAGTTGCTA`
    where the { and } mark the start and end of a region of interest. Alternatively, provide a list containing ranges of bases to be highlighted (in this case 14-21 – counting from zero), or start positions and lengths (here 14 and 8).
12. In the past, people have dumped the complete gene list from the database to a static XML file that is loaded by the FE. You should not be doing this, but should be obtaining data from the DB API via the BL API.
13. **Do not under-estimate the complexity of parsing the Genbank file**. This is quite an involved and complex task to get right and may take a lot longer than you expect.
14. Don't forget the specification says you can skip any Genbank entries with unusual `join=` records. i.e. those that contain references to other Genbank entries or characters such as '<' or '>'. Also where there are multiple `join=` records for alternative splicing, just take the first one.
15. In the past people have reported that there may be odd characters (i.e. not the usual a-z, A-Z, 0-9 and punctuation) in the Genbank file meaning that they need to set the database encoding to UTF-8. I find this surprising in the parts of the file that actually need to be stored. Another approach would be to ensure that the database parser replaces any non-standard characters.
16. **Note that you won't be able to load things like images, JavaScript or CSS from the cgi-bin directory.** These need to live in your HTML directories. Also you won't be able to load them using a 'file://xxxx' URI if you want this to work over the web – you need to use 'http://xxxx'. Wherever possible use URIs of the form '~user/biocomp2/xxxx' (specified in your config file) rather than 'http://student.cryst.bbk.ac.uk/~user/biocomp2/xxxx'. Again this makes portability easier.
17. Don't forget that you can use the `<pre>` tag to display pre-formatted text. This may be a useful way of showing sequence data and alignments. Another way would be to use a table.
18. **You need to be prepared for multiple entries being returned by searches**. Some of the things (like protein names) may not be unique. Identify which search really is a unique search and assume everything else is not. Thus all these searches will return lists of results (which may contain one or more items). Consequently your web site should probably return a table of results for all searches (use the same code you use to display the full dataset) and a link for each item to get the relevant unique result which will take you to the detail page for that item.
19. **When you replace dummy code with real code, announce it to the team, so the layers above can test their code avoiding last-minute integration issues.** Do **not** leave integration to the end! By taking this approach of defining your APIs with dummy code at the beginning, you integrate early and test early – and keep doing so throughout the project.
20. There are some differences between MariaDB (on CentOS7) and MySQL (CentOS6.7 on Hope) so be careful!

21. **Do not rely on unusual libraries** (e.g. bs4 beautifulsoup for XML/HTML parsing) rather than the ones you have been taught and know work on Hope. These libraries may not be available on Hope.
22. **Ideally work on Hope (nomachine).** If you cannot do this, then ensure you test your code on Hope very regularly. This will save you a lot of pain later!
23. Ensure you specify a particular version of Python with a full path to that version in your shebang line rather than relying on:
    `#!/usr/bin/env python3`
    This will avoid version compatibility problems.
24. Since the database loading is essentially a one-off task, there is no major advantage in using a PEP249 library (PyMySQL). You can simply write your SQL to a file and then load this into MySQL using:
    `mysql –u username –p password database_name < file.sql`
    Of course you will need a PEP249 library (PyMySQL) for the DB API code.
25. **Plan for failure!** In other words, ensure that functions you write return some special value to indicate a failure and that you check for this in calls to these routines.
26. The `cgitb` library is useful in the FE CGI code such that errors get reported back to the web browser during debugging.
27. Typical development cycle:
    - Develop the APIs and 3-layer architecture:
      - Define BL API and DB API
      - Implement dummy code for these
      - Start to implement FE code (HTML and CGI scripts) – ensure the CGI obtains (dummy) data from the BL API
      - Ensure the BL API obtains its (dummy) data from the DB API
    - Now the actual development can begin in parallel:
      - The DB developer writes code to populate the database and modifies the DB API code to use real data from the database. (Note that the API as presented to the BL will not change, but the implementation of the code will change from dummy data to real data.) This real code gradually replaces the dummy code. The BL layer magically starts to see real data.
      - The BL developer gradually replaces the dummy code with real code – the FE layer magically starts to see real data.
      - The FE developer continues development
    - If everything works as it should, the integration of the layers is happening from the start and the whole project should simply come together and work!
28. Once the APIs have been agreed, there should be minimal need to discuss anything between members of the group unless new functions are needed or errors are found. How often do you meet with developers of the Python Pandas library? You don't – you simply use it.
29. Note that the API can be separate from the code that actually implements it. Suppose you have some complex code in the BL that needs lots of separate functions. Those could be contained in a separate module imported by the BL API module. This would ensure that the FE isn't tempted to access them directly.

30. Don’t leave old versions of code lying around in your repository. Don’t forget that deleting them means that they can still be recovered using git. If you want to keep them visible in the repository, put them in a directory called ‘ATTIC’ or ‘OLD’ (or something similar) – this makes marking much easier!

Getting Started – practicalities

1. Clone the demonstration repository and make it your own:
    - Create a new empty repository on GitHub (please make it private) – let's assume your GitHub account is called `JohnSmith` and your repository is called `Group1`. Now type the following commands:
    - `git clone git@github.com:AndrewCRMartin/biocomp2.git`
    - `mv biocomp2 Group1`
    - `cd Group1`
    - `git remote set-url origin git@github.com:JohnSmith/Group1.git`
    - `git push -u origin master`
2. Ensure everyone in your group has read/write access to this repository by adding them as collaborators.
3. Now edit the `install.sh` script and `cgi-biocomp2/config_demo.py` files to reflect the directories and URLs for where you are installing it.
4. Type `./install.sh` to install the demo website and view it with a browser.

Reflective Essay (2-5 pages)

1. Explain how you set about the project detailing:
    - how you interacted with the other members of the group
    - how you (as a group) came up with the requirements for the overall project
    - how you (as an individual) came up with the requirements for your contribution
2. How well did the development cycle work within the group?
3. You should document the development process - what stages did you go through and what steps did you take? Did you go through several iterations or did you spend a long time in design and then a relatively short development stage?
4. What strategies (if any) did you take for testing your code - particularly if other people's code wasn't ready?
5. Are there any known issues or bugs, or anything that doesn't work as specified?
6. You should discuss what worked well and what could have worked better - both with your code and the group interaction. Were there any particular problems? Conversely were there any solutions or ideas that you were particularly proud of?
7. Were there alternative strategies you could have used? (Either for the project design or implementation)
8. What do you think you gained out of the project? How has this experience helped you? What experience or insights have you gained?

Consequently I suggest headings of the form:

1. Approach to the project
    - Interaction with the team
    - Overall project requirements
    - Requirements for my contribution
2. Performance of the development cycle

3. The development process
4. Code testing
5. Known issues
6. What worked and what didn't - problems and solutions
7. Alternative strategies
8. Personal insights

Code documentation (1-2 pages per tier)

Note that this should not simply be a re-printing of the code with the comments in bold (or something similar).

For the DB layer

- Instructions on how to run the program(s) to parse Genbank and populate the database
- Documentation for the DB access API – how to call each function, what parameters are required and in what format, and what is returned and in what format. Provide examples.
- The call tree (how one routine calls another) so people can navigate through the code.
- Provide table definitions and UML diagrams (or equivalent) detailing indexes, primary and foreign keys, and any constraints.

For the BL layer

- Documentation for the BL API – how to call each function, what parameters are required and in what format, and what is returned and in what format. Provide examples.
- The call tree (how one routine calls another) so people can navigate through the code.

For the FE layer

- A brief description of how to use the web site. This may be a separate document or suitable help information on the web page itself.
- Instructions on how to call the CGI scripts.
- The call tree (how one routine calls another) so people can navigate through the code.

Overall

- Explain how to install the software on a different machine (e.g. what needs to be done to the configuration file, is there anything else that needs to be edited/changed).
- How to populate the database should have been explained with the DB layer, but is there anything else that needs to be run – e.g. to calculate and store the information on chromosome codon usage. If so, how are these programs run?

Marking criteria

In general we will be taking a loose, "holistic" approach to marking so we are able to give credit where it has been earned rather than penalizing people for not doing enough in one area. You are expected to stretch yourselves!

However, criteria will include:

- How well does the web server meet the required brief?
- How easy is it to use?
- Is the 3-layer design properly implemented?
- Quality of the HTML
    - is it compatible with XHTML (or later) standards?
    - is all formatting done with CSS?
    - does it make appropriate use of semantic markup?
- Quality of code
    - is code properly laid out?
    - is it readable and understandable using suitable variable names?
    - does it use appropriate error and code checking?

        - is it properly commented? This should include a header at the top of each file and a docstring comment for each function. Comments must include author information.
        - are things like hard-coded pathnames, database details, etc., appropriately placed to make installation on a different machine straightforward? i.e. in a configuration file wherever possible (the possible exception being the HTML file).
- Quality of documentation
    - is all code (especially the APIs to the code) appropriately documented?
    - is the documentation clear and easy to understand?
    - is the web interface documented – either with separate (brief) documentation, or with appropriate help information on the web pages?
- Commentary/Reflection – are the following items addressed in a balanced and constructive manner?
    - How well did the development cycle within the group work?
    - What were the challenges?
    - What might they have done differently?
    - What have you gained from this experience in terms of becoming a better programmer?