February 8, 2018

Take-Home Exam

Academic Integrity Policy

Honesty and integrity are two of the most important values of the university in its pursuit and dissemination of truth and knowledge. Faculty and students share the responsibility for maintaining the probity of the educational experience and preserving high standards of excellence. Academic dishonesty – cheating and plagiarism – is unacceptable behavior morally, ethically and legally; and it cannot be justified or tolerated.

Rules for this Take-Home Exam

Carefully read the instructions and questions. Take your time to make sure you fully understand what is being asked. You may only ask questions to the class instructor, not your classmates. Furthermore, questions may only be asked during the presential part of the exam.

- E-mail your solutions to the exam to leila.taher@fau.de by **12:00 p.m. (*12:00 Uhr*) on February 19, 2018**. If this is impossible for you, we can arrange in advance and by request a hard copy submission. Late submissions will not be accepted.
- You may use class notes, class slides, Perl books or the internet (you are **not allowed** to submit questions to internet discussion groups, though!). If you find a solution of a test problem in a book or online, cite it in your submission and do not copy it as-is, but make changes which demonstrate you understand what you are writing. Otherwise we may not accept it.
- The exam must be taken **completely alone**. Showing it or discussing it with anybody is forbidden, including (but not limited to) the other students in the course in current or previous years. It is also forbidden to use any solutions to similar problems from previous years as reference material.
- **We will check for plagiarism using special software, the internet and our own knowledge**. If we suspect any form of malpractice, we will ask both the person who copies and the person who facilitates the copying to come in to office hours and answer some questions so that we have a better perspective. If we feel that the evidence of cheating is strong enough, we may skip this step and give the involved parties a 0 for the assignment.
- You may answer this exam in German or English.
- Please, be aware that while you may work on a Windows PC, your programs should also run (and will only be tested) under UNIX/Linux.
- Finally, if you have technical problems, please e-mail the instructor (leila.taher@fau.de).

General instructions

- Use the pragmatic modules `strict` and `warnings`, but **do not** use any other modules unless explicitly told to.
- Programs must be consistently indented and commented so that a reasonable person can understand them. Moreover, if you do not know how to write a working Perl script, try including some pseudocode (`https://en.wikipedia.org/wiki/Pseudocode`). Depending on your understanding of the problem, we may award partial credit.
- There will be a penalty if your program does not follow directions. For example, programs must not require inputs other than those specified. Also, if the exercise requires you to ask the user for input, the input should not be hard-coded in your program.
- If you use sub-routines, all inputs should be passed as arguments.
- Try to use consistent descriptive variable names (e.g., do not call your variables `$var1`, `$var2`, etc, but rather `$sequence_file`, `$sequence_length`, etc).
- Call your programs `BioPerlEx1.pl`, `BioPerlEx2.pl` and `BioPerlEx3.pl` and add your initials as a prefix (for example, Leila Taher's programs would be called `LT_BioPerlEx1.pl`, `LT_BioPerlEx2.pl` and `LT_BioPerlEx3.pl`).
- Please, do not include your input (or output) files. Only your three Perl scripts.

Grading

- the correctness and robustness of your scripts (60%). Your program should produce correct answers on all *legal* input, i.e., not just on the (provided) example input files, and produce comprehensible error messages on invalid input (even if not explicitly stated!).
- the clarity of your scripts (20%). By reading through your code and comments, we should be able to figure out how your code is organized and why it works (or fails). If we cannot, we reserve the right to deduct points. You should be able to solve the exercises without resorting to functions/procedures that were not explained in class. However, if you do, explain carefully what the functions do. Even if you do so, we will most likely deduct points. For example, if you "have" to search for a particular string in a list (array) with the Perl `grep` function we will conclude: i) that you copied this code from somewhere else (this function was never addressed in class); and ii) that you do not know how to loop through an array and use regular expressions (because that would be the *standard* solution to this problem).
- the design and efficiency of your scripts (20%). All of your data structures should have the expected asymptotic time and space complexity, but the constants will not generally matter.

Exercise 1 100/3 points

The file `repeats.txt` contains pairs of overlapping regions in the human genome. The regions are specified as genomic ranges or intervals (chromosome, start and end positions). Columns 1 to 3 contain the coordinates of regions that are evolutionary conserved (`https://en.wikipedia.org/wiki/Conserved_sequence`) across multiple mammalian species. Columns 4 to 7 contain the coordinates of transposons (`https://en.wikipedia.org/wiki/Transposable_element`). Column 8 indicates the transposon class. For example:

```
chr1    2166503 2170482 chr1    2166470 2166782 SINE
chr1    2166503 2170482 chr1    2167853 2168143 SINE
chr1    2166503 2170482 chr1    2168846 2168888 LINE
```

`chr1:2166503-2170482` is an evolutionary conserved region in the human genome that overlaps with SINE (`chr1:2166470-2166782` and `chr1:2167853-2168888`) and LINE (`chr1:2168846-2168888`) transposons.

The file contains 4,725 evolutionary conserved regions. Regions that do not overlap with any transposon (of any class) are indicated as in the following example:

```
chr1    1717278 1717315 .       -1      -1
```

- Write a Perl program that computes the fraction of each sequence that overlaps with DNA, LINE, LTR and SINE transposons. For example:
  - 14.35% of `chr1:2166503-2170482` overlaps with SINE transposons;
  - 1.08% of `chr1:2166503-2170482` overlaps with LINE transposons.
- The name of the input file should be a read as a command line arguments of your program.
- The output should include the coordinates of the conserved region (as a genomic range) and the fraction overlapping with DNA, SINE, LTR and LINE transposons, as percentage with two decimals:

  ```
  chr1    2166503 2170482 14.35% SINE
  chr1    2166503 2170482 1.08% LINE
  chr1    2166503 2170482 0.00% DNA
  chr1    2166503 2170482 0.00% LTR
  ```

- The output should be written to a file with a name consisting of the name as the input file (e.g., `repeats.txt`) followed by a dot, the name of the program (e.g., `LT_BioPerlEx1.pl`) and the extension ".txt" (e.g., `repeats.txt.LT_BioPerlEx1.pl.txt`). This should work even if the user decides to rename the Perl script or if the input file is not called `repeats.txt`.

Exercise 2 100/3 points

Gene ontology (GO, `https://en.wikipedia.org/wiki/Gene_ontology`) is a major bioinformatics initiative that aims to:

- maintain and develop a controlled vocabulary for gene and gene product attributes;
- annotate genes and gene products; and
- provide tools for easy access of the data provided by the project.

The GO annotation terms are structured as a graph (`https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)`). In the graph, the GO annotation terms are nodes and the edges are the relations between them. Just as each term is defined, so the relations between GO annotation terms are also categorized and defined.

The *is a* relation forms the basic structure of GO. If we say A *is a* B, we mean that node A is a subtype of node B. For example, "mitotic cell cycle" *is a* "cell cycle", or "lyase activity" *is a* "catalytic activity". The *is a* relation is transitive, which means that if A is a B, and B is a C, we can infer that A is a C.

- Download the file `http://purl.obolibrary.org/obo/go/go-basic.obo`.

  Every GO term has a term name, e.g. mitochondrion, glucose transport, amino acid binding, and a unique zero-padded seven digit identifier prefixed by GO (id), e.g., GO:0005125 or GO:0060092. In addition, a *namespace* denotes which of the three sub-ontologies (cellular component, biological process or molecular function) the term belongs to. A *definition* provides a textual description of what the term represents, plus reference(s) to the source of

the information. Finally, each term has one or more relationships to other terms in the ontology. The relations included are: *is a* (`is_a`), *part of* (`part_of`), *regulates* (`regulates`), *negatively regulates* (`negatively_regulates`) and *positively regulates* (`positively_regulates`). (see `http://geneontology.org/page/ontology-relations` for more information). The terms in this file form an acyclic graph. For example:

```
[Term]
id: GO:0000001
name: mitochondrion inheritance
namespace: biological_process
def: "The distribution of mitochondria, including the mitochondrial genome,
into daughter cells after mitosis or meiosis, mediated by interactions
between mitochondria and the cytoskeleton." [GOC:mcc, PMID:10873824, PMID:11389764]
synonym: "mitochondrial inheritance" EXACT []
is_a: GO:0048308 ! organelle inheritance
is_a: GO:0048311 ! mitochondrion distribution
```

- Write a Perl program that prompts the user for a GO id (e.g., GO:0000001) and prints its name, namespace and all *is a* relationships for that GO term, including transitive relations. For example, GO:0000001 (mitochondrion inheritance) *is a*:

  GO:0006996, GO:0007005, GO:0008150, GO:0009987, GO:0016043, GO:0048308, GO:0048311, GO:0051179, GO:0051640, GO:0051641, GO:0051646 and GO:0071840.

  - Each "is a" relation of a given GO id should be printed exactly once.
  - The list of "is a" relations should be sorted according to the zero-padded seven digit identifier of the corresponding GO ids (e.g., GO:0016043 should be before GO:0044699 because 16043 < 44699).

Exercise 3 100/3 points

XML (`https://en.wikipedia.org/wiki/XML`) is an acronym standing for Extensible Markup Language. XML is a language similar to HTML, but it allows for more flexibility. XML files are structured text files that use tags to define objects and object attributes, and can be thought of as a text-based database.

The best way to understand XML is to look at an example. Below you can see three records from a movie database. Each record contains two fields: the title of a movie, and its genre.

```
<?xml version="1.0"?>
 <movies>
   <movie id="1">
      <title>The Ghost and Mr. Chicken</title>
      <genre>Comedy</genre>
   </movie>
   <movie id="2">
      <title>Gone with the Wind</title>
      <genre>Drama</genre>
   </movie>
   <movie id="3">
      <title>ThunderBall</title>
      <genre>Adventure</genre>
   </movie>
 </movies>
```

The first line contains a processing instruction known as the XML declaration. The remainder of the file is composed of XML elements. Each element consists of a start tag and an end tag. In the example, `<title>` is a start tag and `</title>` is an end tag. `<title>Gone with the Wind</title>` is an element.

Elements may contain:

- text, e.g., "The Ghost and Mr. Chicken"
- attributes, which contain data related to a specific element. For example, "id" is an attribute of "movie". Attributes must be quoted (e.g., `"1"`).
- other elements, e.g., the element "movie" contains the elements "title" and "genre".

- a mix of the above

As the data in XML format is not readable by general users, it may be necessary to convert it to some user friendly format, like CSV (Comma Separated Values, `https://en.wikipedia.org/wiki/Comma-separated_values`), and *vice versa*. CSV files can easily be opened using Microsoft Excel.

- Write a Perl program that takes a CSV data as input and produces an XML file as output. The CSV file that your program takes as input should look similar to `Ex3BioPerl.csv`. You can open the file with a text editor or with Excel. The first line contains the headers. Each further line contains information on a patient (this file contains only two records, but your program should work with any number of records). The fields in the file are separated by commas.
- Your XML file should always start with the lines:

  ```
  <?xml version="1.0"?>
  <ADT_GEKID>
   <Menge_Patient>
  ```

  and end with the lines:

  ```
   </Menge_Patient>
  </ADT_GEKID>
  ```

  In between, the XML file should contain a `Patient` element for each patient (row) in the CSV file, with the following structure:

  ```
  <Patient>
    <Patienten_Stammdaten Patient_ID="1">
      <KrankenkassenNr>100</KrankenkassenNr>
      <Patienten_Nachname>Duck</Patienten_Nachname>
      <Patienten_Vornamen>Donald</Patienten_Vornamen>
      <Patienten_Geburtsname></Patienten_Geburtsname>
      <Patienten_Geschlecht>M</Patienten_Geschlecht>
      <Patienten_Geburtsdatum>09.06.1934</Patienten_Geburtsdatum>
      <Menge_Adresse>
        <Adresse>
          <Patienten_Strasse>Staudtstr. 5</Patienten_Strasse>
          <Patienten_Land>Deutschland</Patienten_Land>
          <Patienten_PLZ>91058</Patienten_PLZ>
          <Patienten_Ort>Erlangen</Patienten_Ort>
        </Adresse>
      </Menge_Adresse>
    </Patienten_Stammdaten>
    <Menge_Meldung>
      <Meldung Meldung_ID="ER000001010000384" Melder_ID="ER000001">
        <Meldedatum>06.02.2018</Meldedatum>
        <Meldeanlass>Diagnose</Meldeanlass>
    </Menge_Meldung>
  </Patient>
  ```

- The headers in the CSV file correspond to the tags in the XML file, with few one exception:
  - the `Meldedatum` should be the date ("Datum") plus 5 days.
- The `Patient_ID` attribute of the `Patienten_Stammdaten` element should be the record (line) number in the CSV file.
- Ignore fields in the CSV file that are irrelevant to the XML file.
- All elements in the example should be included in the XML file. If some information is missing in the CSV file (e.g., no address is indicated), include no text in the corresponding element, e.g.:

  ```
  <Patienten_Strasse></Patienten_Strasse>
  ```

- Do not assume that the fields in the CSV file are always in the same order. Rather, rely on the headers.

- Warn the user if a certain field does not conform to the expected format (e.g., the "PLZ" should be a number with five digits).
- If necessary, re-format the dates so that they are in the standard European format (DD.MM.YYYY) in the XML file.
- The names of the input CSV and output XML files should be a read as a command line arguments of your program.