ACLS — Programming, Algorithms and Data Structures

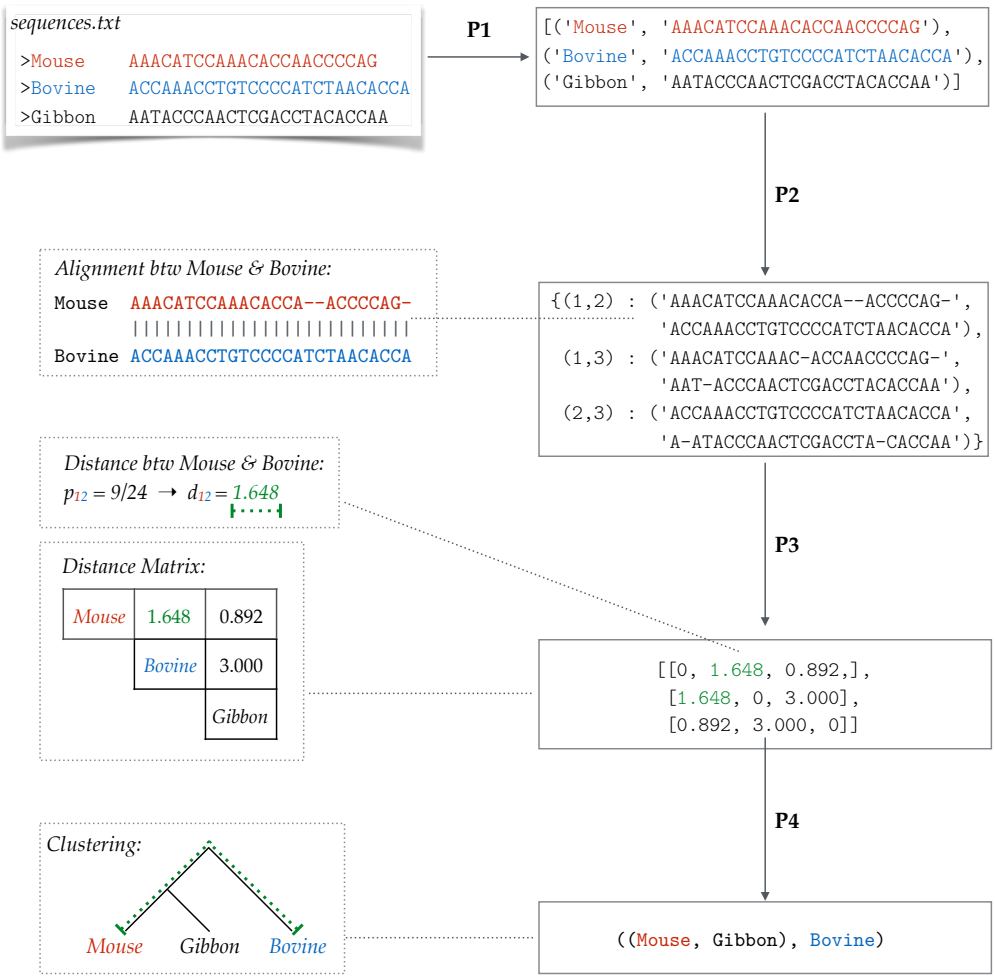
Programming Project 2019

Manuel Gil

Last update: October 21st 2019

In this project you will implement a program that reads genomics sequences from a file and clusters them in a tree according to their pairwise evolutionary distances. No prior knowledge on genomics is necessary. The project has four main parts (P1—P4). P1 and P3 are relatively straightforward. P4 is a little bit more difficult. P2 is challenging. Each part should be encapsulated in a module, with respective names P1.py, P2.py, P3.py, P4.py. In this document you find the details about the individual parts, about the submission, and about the scoring and marking.

Here is the overall structure of the project (please refer to the corresponding sections for explanation):



P1. Parsing a Sequence File

Prototype: `list(tuple(string, string))` **`ParseSeqFile`**`(string)`

Input: Path and filename pointing to a sequence file
Output: List of pairs, where each pair consists of a label and the corresponding sequence

Description
Here is an example sequence file:

```
>Mouse     ACCAAA CATCCAAACA CCAACCCCAG CCCTTACGCA ATCATAC AAAGAATATT
>Bovine    ACCAAACCTGTCCCCATCTAACACCAACCCACATATACAAGCTAAACCAAAAATACC
> Gibbon    ACTATACCCA CCCAACTCGA CCTACACCAA TCCCCACATA GCACACAGAC CAACAACCTC

>Orangutan  ACCCCACCCG TCTACACCAG CCAACACCAA CCCCCACCTA CTATACCAAC CAATAACCTC
>Gorilla   ACCCCATTTA TCCATAAAAA CCAAC ACCAA CCCCC ATCTA ACACACAAAC TAATGACCCC

>  Chimp   ACCCCATCCA CCCATACAAA CCAACATTAC CCATCCA ATATACAAAC ACCTC
>Human ACC C CA CTC A C C C A T A C AAA   CCA A C A C C A CTCTCCACCTAATA TACAAA T AC CTC
```

Each line in an input file is either empty, or it starts with a ">" character. If a line starts with ">" then it contains data. One data item consists of a Label (e.g. `Mouse`, or `Chimp`) followed by the corresponding nucleotide sequence (terminated by a *newline*). Between a label and the sequence there can be an arbitrary number of white space characters or tabulators ("tabs"). No white spaces or tabs are allowed inside a label. The sequence can contain an arbitrary number of white spaces; the nucleotides can only be `A`, `C`, `G`, `T`. Note that the sequences do generally not have the same length, due to insertion deletion events during evolution.

Write a function that parses such sequence files. If the file violates the above format definition, it should throw an exception with the message "malformed input".

P2. Alignment of Sequences

Prototype:

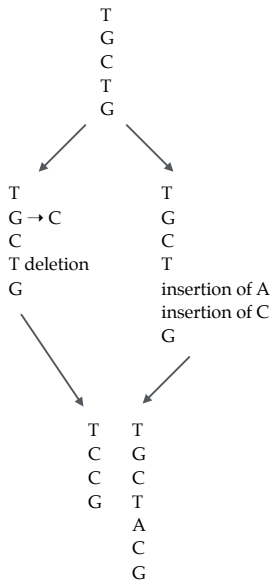
```
dict(tuple(int, int) -> tuple(string, string))
AlignByDP(list(tuple(string, string)))
```

Input: The input format corresponds to the output format of P1. But the function should still be considered self-contained, i.e if the input violates the above type, it should throw an exception with the message "malformed input".

Output: A dictionary. The keys should be tuples of integers (i,j). A key maps to the alignment of the corresponding sequences from the input. The alignment should be represented as a tuple (<Aligned Sequence i>, <Aligned Sequence j>).

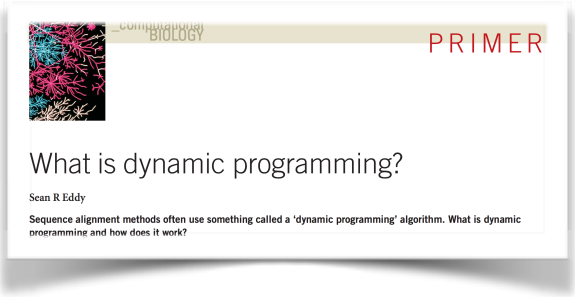
Description

Genomic sequence evolve, among other mechanisms, through substitutions (where a particular nucleotide is mutated into some other), and insertions/deletions of nucleotides, referred to as indels. Consider the evolutionary scenario on the right picture, where a sequence TGCTG diverges into two sequences. The one on the left is affected by one substitutions (G → C) and one deletion (of T). The right one gets two insertions. The ancestral sequence and the whole process is hidden from us. We observe the present day sequences TCCG and TGCTACG. Because of the indel process, genomic sequences have to be aligned, i.e. homologous characters have to be determined —a procedure termed sequence alignment. The alignment of the two sequences is:




```
TCC---G
TGCTACG
```

The goal of P2 is to align each pair of the input sequences —for *n* sequences there are *n(n-1)/2* pairs— according to the dynamic programming (DP) algorithm described in the following primer (to be found on Moodle):



Use the scoring system from the primer. Gaps should be represented by a dash "-". The resulting two strings in a pair of aligned sequences have the same length.

P3. Pairwise Distance Matrix

Prototype:

```
list(list(float))
ComputeDistMatrix(dict(tuple(int, int) -> tuple(string, string))
```

Input: The input format corresponds to the output format of P2. But the function should still be considered self-contained, i.e if the input violates the above type, it should throw an exception with the message "malformed input".

Output: A symmetrical matrix with zeroes in the diagonal. Each entry i,j corresponds to the evolutionary distance between sequences i and j.

Description

Consider the scenario on the right, where a sequence TTCAAGAC diverges into two sequences. The one on the left is affected by 7 substitutions (indicated by the short arrows), the right one by 3, thus 10 substitutions in total. When we compare the two resulting sequences (the evolutionary process is effectively hidden from us) we find 2 (out of 8) positions that differ. Thus, most substitutions are hidden from us.

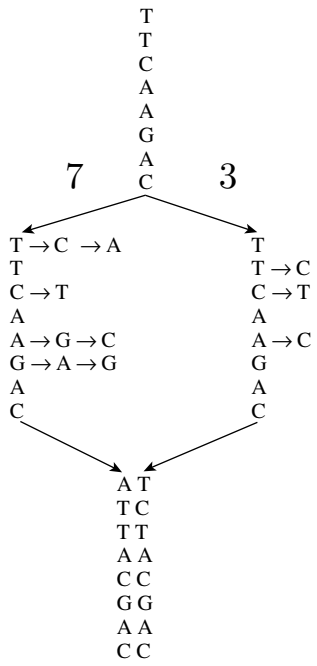


Figure from Yang, Computational Molecular Evolution, 2006

The p-distance is the proportion p of nucleotide sites at which two sequences being compared are different. It is obtained by dividing the number of nucleotide differences by the total number of nucleotides compared, $p=2/8$ in our example. However, the actual number of substitutions per site is $(7+3)/8 = 1.25$.

An estimate of the actual number of substitutions —i.e. an evolutionary distance— is obtained by correcting the p-distance for the hidden substitutions using the following formula:[1]

$$d_{ij} = -\frac{3}{4} \ln \left(1 - \frac{4}{3} p_{ij} \right)$$

For $p_{ij} \geqslant 3/4$ the sequences have diverged too much and look like unrelated random sequences, so that d_{ij} is undefined or infinity. In this case your function should set d_{ij} to 30. The goal of P3 is to compute the evolutionary distance for each alignment of P2, and to store it in a symmetric matrix of pairwise evolutionary distances. Note that the distances capture substitutions and ignore any indel information. The alignments that you get from P2 contain gaps. Exclude all gap positions in the computation of p. For example $p=1/4$ for the following alignment:

```
TCC---G
TGCTACG
```

[1] In our case we obtain $d = 0.3$, which is a gross underestimate of 1.25. Note, however, that the example is quite unrealistic; it contains a lot of hidden substitutions for illustration purpose.

P4. Clustering

Prototype: `string` **`Cluster`**`(list(list(float)), list)`

Input: The input format corresponds to the output format of P3. But the function should still be considered self-contained, i.e if the input violates the above type, it should throw an exception with the message "malformed input".

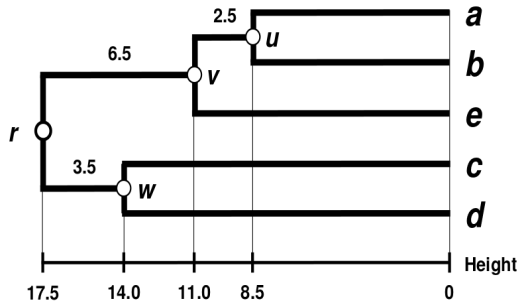
Output: A binary tree in string format

Description

In this part, you should cluster the input sequences according to the distance matrix obtained in P3. To do so, implement one of the most basic clustering methods (called WPGMA). The algorithm, together with an example, is described on Wikipedia:

https://en.wikipedia.org/wiki/WPGMA

The final tree structure should be represented as nested parentheses and be returned as a string. The labels should be the labels from the input sequence file. The tree below (from the Wikipedia entry) has the representation `((a,b),e),(c,d))`:You only



have to return the topology of the tree, without any branch lengths. Note that the WPGMA tree is a crude estimate of the evolutionary tree relating the input sequences.

Hint: Internally to your code, you could represent the distance matrix as dictionary to facilitate the reduction of the matrix. An even more straightforward approach are 2D dictionaries. For instance:

```
from collections import defaultdict

d = defaultdict(dict)
d['a']['b'] = 1
d['a']['c'] = 2
for i in d:
    for j in d[i]:
        print(i, j, d[i][j])
del d[‘a']['b']
del d['a']['c']
del d[‘a’]
```

Modalities

Deadline: 20. Dec 2019

The deadline is binding. I recommend that you plan. In particular, start early and distributed the four parts during the semester.

Submission

Upload a ZIP-file named <name>_<surname>.zip on Moodle. It should extract into a directory <name>_<surname> containing P1.py, P2.py, P3.py, P4.py. Each file should contain the main function according to the prototypes given above. For functions that you could not tackle, provide at least an empty function returning `None`.

Independence

You *are* allowed to have discussions with your peers. You *are not* allowed to code in groups, nor copy code from others or the net. You have to come up with each line of code yourself.

Paradigm

The whole program can be written in procedural form. However, use of any paradigm (e.g. OO, or functional) is allowed. Paradigm will not influence the mark.

Scoring

I will score each part according to functionality, modularity, and clarity. On the last page you can find the scoring sheet I will use.

Functionality

I will apply Unit Tests to check for correctness. You get the max points if your code passes all tests, and deductions for failed tests. A reasonable computational complexity is relatively straightforward, as the algorithms are basically defined by the tasks. However, if your code is inefficient (e.g. cubic instead of quadratic) I will deduct points.

Design

Your code should conform to the required specification (modules and main function's prototypes.) Further, I will score your design ("well thought out", "partially planned", "designed at the keyboard"); basically you should pay attention that your code is modular, by defining functions appropriately to break down the problems. In case of doubt, use more functions.

Clarity

Your code should be understandable ("understandable", "hard to follow", "incomprehensive"). This is achieved by:

i. Expressive variable and function naming
ii. Well formatted code
iii. Concise, meaningful, well-formatted comments (including docstrings). Not wordy, redundant or unnecessary ones.

Put yourself into the perspective of a person reading a hardcopy of your code.

Marking

The mark contributes 30% to the final mark. I think it is a personal decision, how much energy a student invests in a particular lecture. The point system should guide you, so that you can work towards a particular mark. However, bear in mind, that you could have undetected bugs, which reduce the points.

The maximum number of points is 140. I *plan* to apply a linear scale to mark (0pts $\rightarrow$ 1, 135pts $\rightarrow$ 6), but *may adapt it* according to the actual distribution of points of the class. Thus, for the time being, the mark can be *estimated* as

$$\text{Mark} = 1 + \text{Points}/27$$

For instance, 81 points would give a 4; 108 points a 5.

Scoring Sheet

The actual scoring sheet may differ in some details, but will essentially be structured as follows:

Name:		**MARK**	

Programming Project: Scoring Sheet

P1	**Parsing a Sequence File**	(29)	
	Functionality	(15)	
	Design	(5)	
	Clarity	(9)	
P2	**Alignment of Sequences**	(39)	
	Functionality	(25)	
	Design	(5)	
	Clarity	(9)	
P3	**Pairwise Dissimilarity**	(29)	
	Functionality	(15)	
	Design	(5)	
	Clarity	(9)	
P4	**Clustering**	(43)	
	Functionality	(29)	
	Design	(5)	
	Clarity	(9)	
		TOTAL (140)	