**Problem Set 1: Aligning and Modeling Genomes**

**საჭირო საინსტალაციო პაკეტები:**

1. გადმოწერეთ Gnuplot-ის საინსტალაციო შემდეგი ლინკიდან: <http://www.gnuplot.info/>. საინსტალაციო შეგიძლიათ გადმოწეროთ Google Classwork-დან. ინსტალაციის დროს მონიშნეთ ოფცია „Add application directory to your PATH environment variable”
2. Anaconda3-ის ინსტალაცია, Conda გარემოს გამართვა
3. Python 2-ის დაყენება Conda გარემოში (მიუთითეთ პროექტის სახელიც):

conda create - -name envName python=2.7

1. სასურველი გარემოს გააქტიურება: conda activate envName
2. გადადით პროექტის საქაღალდეში cd ბრძანებით
3. პირველი ამოცანის გაშვება: python ps1-seqalign.py human\_HoxA13.fa mouse\_HoxA13.fa
4. **Evolutionary distances of orthologs**\* **and paralogs**\*

In this problem, you will implement the **Needleman-Wunsch algorithm** for pairwise sequence alignment, apply it to the protein-coding sequences of related genes from several mammalian genomes, and use the results to learn about their evolution.

(a) In the Problem Set Folder, we have provided a python skeleton program **ps1-seqalign.py**, which you will complete. We provide a traceback routine**, but you will write the code to fill in the score and traceback matrices.** The skeleton program specifies a substitution matrix and gap penalty. Please submit (1) the portion of the code that you wrote; and (2) the score of the alignment of the human and mouse HoxA13 genes, which we also provide.

The command to run the program is:

python ps1-seqalign.py <FASTA 1> <FASTA 2>

The Hox cluster (Hox გენების კლასტერი) is a set of genes that are crucial in determining body plan formation during embryo development. They are found in all bilateral animals, in species as distant as the fruit fly (დროზოფილა). The fruit fly has one Hox cluster, while most vertebrates have four. It is thought that vertebrates have undergone two rounds of whole-genome duplication, giving rise to four Hox clusters from the ancestral one, although the hypothesis remains controversial.

In the remainder of this problem, you will use your **Needleman-Wunsch** alignment program to analyze the sequences of several Hox genes, and estimate the date of the most recent vertebrate whole-genome duplication. In particular, we are interested in using the N-W alignment score as a distance metric between two sequences.

The score of alignment is 2971.

(b) Make minor adjustments to your alignment program so that the score it computes can be interpreted as a distance metric. That is, the score of a sequence aligned with itself should be zero, all scores should be non-negative, and sequences that are more dissimilar should be given a score with a greater magnitude. Describe the changes you made in your handin; no code is necessary.

Firstly, I changed part in which we initialize dynamic programming table F. Instead of subtracting gap penalty, which would end up in negative number as gap penalty itself is positive, I add it. For example:  
initial code was this: F[i][0] = 0 – i \* gap\_pen . I changed it to this: F[i][0] = 0 + i \* gap\_pen

Then, I also changed the update rule of the matrix. Firstly, if the two bases match, then I’m using 0 instead of the value retrieved from substitution matrix, otherwise I’m using absolute value of the latter. This means, if the bases match, formula will be F[i-1][j-1] + 0, if they don’t match : F[i-1][j-1] + |subst\_matrix[seq1[i-1]][seq2[j-1]]|

This way, if the two bases mismatch is common, for example A and G, we know that the respective value in substitution matrix would be relatively larger negative value, so that when we would add it to the cost, it would end up bigger than for example T and G. But in this modification, we need to get lower value, so if we just take the absolute values of the substitution matrix, then in case of A and G, we’ll end up with smaller number. (Because if a>b and both of them are negative, then |a| < |b|).

Also, for the cases in which either of sequences are extended with gap, I add gap penalty instead of subtracting it. (same reason as in the initializing part change).

And, for the update rule, I take minimum of the 3 values computed, instead of maximum. So, basically modified algorithm is this:

def seqalignDP(seq1, seq2, subst\_matrix, gap\_pen):

F = [[0 for j in range(len(seq2) + 1)] for i in range(len(seq1) + 1)]

TB = [[PTR\_NONE for j in range(len(seq2) + 1)] for i in range(len(seq1) + 1)]

# initialize dynamic programming table for Needleman-Wunsch alignment (Durbin p.20)

for i in range(1, len(seq1) + 1):

F[i][0] = 0 + i \* gap\_pen

TB[i][0] = PTR\_GAP2 # indicates a gap in seq2

for j in range(1, len(seq2) + 1):

F[0][j] = 0 + j \* gap\_pen

TB[0][j] = PTR\_GAP1 # indicates a gap in seq1

for i in range(1, len(seq1) + 1):

for j in range(1, len(seq2) + 1):

fb = base\_idx[seq1[i - 1]]

sb = base\_idx[seq2[j - 1]]

sub = 0 if fb == sb else subst\_matrix[fb][sb] \* (-1)

both\_extended = F[i - 1][j - 1] + sub

gap\_seq2 = F[i - 1][j] + gap\_pen

gap\_seq1 = F[i][j - 1] + gap\_pen

scores = [(both\_extended, PTR\_BASE), (gap\_seq2, PTR\_GAP2), (gap\_seq1, PTR\_GAP1)]

scores\_sorted = sorted(scores, key=lambda x: x[0])

result = scores\_sorted[0]

F[i][j] = result[0]

TB[i][j] = result[1]

return F[len(seq1)][len(seq2)], F, TB

(c) Apply your modified program to compute a distance between the human HoxA13 gene and the mouse HoxA13 gene.

Distance is 197.

(d) The modern mammalian genes HoxA13 and HoxD13 arose from a single ancestral gene by whole-genome duplication, long before the human-mouse divergence. We provide the sequences of the human and mouse HoxD13 genes in the folder. Given that the fossil record shows that human and mouse diverged about 70 million years ago, use your distance metric and your results from part (c) to estimate the date of the whole-genome duplication that gave rise to HoxA13 and HoxD13. Make sure to state the assumptions underlying your estimate.

I computed distances between human\_hoxD13 and human\_hoxA13, which was 1145. By a simple proportion, I’m multiplying 70 million by 1145 and dividing it by 197 (distance between human\_hoxA13 and mouse\_hoxA13).

But I also did the same to mouse\_hoxD13 and mouse hoxA13, which was 1098, and also human\_hoxD13 and mouse\_hoxD13, which was 258.

So, in order to get better estimation, I think it would be better to average the two above results. First was 406 million, and the second was 297 million. That’s why my final guess would be around 350 million years.

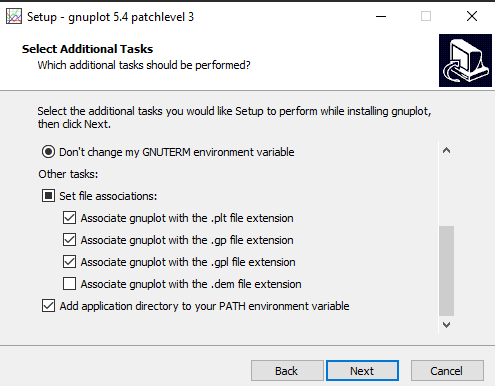
1. **Sequence hashing and dotplot visualization**

As you have seen in problem 1, sequence alignment is a quadratic time algorithm. Full sequence alignment is therefore only feasible for sequences near the length of a single gene. To align larger regions of a genome, heuristic approximations are typically used. In this problem, you will use hashing techniques to guide the alignment of a 1 megabase (1 million nucleotides) region surrounding the HoxA cluster in human (human-hoxa-region.fa) and mouse (mouse-hoxa-region.fa). You will use dotplots to visualize the performance of various hashing methodologies.

The code provided (ps1-dotplot.py) finds all 30-mers in the human that also appear in mouse. On a dotplot, each of these matches is represented as a single dot at (x, y), where x is a coordinate for the beginning of a 30-mer in human and y is a coordinate for the beginning of a matching 30-mer in mouse. We provide a plotting function that will produce dotplot images. The format of the image is determined by the file extension (\*.ps, \*.png, \*.jpg). There is also code for heuristically judging the specificity of the matches (the fraction of matches that occur near the diagonal of the dotplot).

1. Run the script unchanged to generate a dotplot for all exact matching 30-mers. It must be run in the same directory where it is located, since it also requires utils.py and plotting.py. This script also requires **gnuplot** software which should be installed.

გადმოწერეთ Gnuplot-ის საინსტალაციო შემდეგი ლინკიდან: <http://www.gnuplot.info/>. საინსტალაციო შეგიძლიათ გადმოწეროთ Google Classwork-დან. ინსტალაციის დროს მონიშნეთ ოფცია „Add application directory to your PATH environment variable”



The command to run the program is:

python ps1-dotplot.py <FASTA 1> <FASTA 2> <PLOT FILE (\*.ps, \*.png, \*.jpg)

Describe what you see. How many hits are there and what percentage fall near the diagonal? Do you observe any structure in the off-diagonal hits? What types of genomic elements could cause such a pattern? Why are matches that are close to the diagonal more likely than off-diagonal matches to represent “correct”, or orthologous\*, alignments?

There are 62829 hits, and 24.70197% of them fall near the diagonal.

The pattern I see in the off-diagonal hits is that they seem to be aligned on the same row and column on the plot, hence making rectangle-like shape. If we stay on the same row and move only along the column, it means that the location of the genes in one of the sequences stays the same, and it changes only in the other sequence. And vice versa for columns. So, if we get several matches aligned on the same rows (and columns), this means that the same segment is matched several times in other genome. This can be caused by several types of genomic elements, for example segments that were duplicated. Duplication would cause them to appear in the genomes several times, hence creating matches in our case that would look like a rectangle because of the fact that on the same row there would be several matches in the other sequence (same goes for columns).

Matches close to the diagonal more likely represent “correct” alignments because being close to the diagonal means that the locations of two k-mers also match, which means they probably have same function and their match is not a result of accident or mutation.

1. Make the following modifications to the script and report how the plot changes qualitatively and quantatively (how many hits, what percentage are near the diagonal). Also briefly describe how you implemented each change.

i. Modify the script to find all exact matching 100-mers

I just changed the kmerlen variable in the script to 100.

There were 1198 hits and 100% of them are on diagonal.

ii. Modify the script to find all 60-mers that match every other base

I just changed the kmerlen variable in the script to 60 and added [::2] to the end of both keys so that it takes every second item from the lists.

Like this: key = seq1[i:i+kmerlen][::2]

There were 23933 hits and 38.74149% of them are on diagonal.

iii. Modify the script to find all 90-mers that match every third base

I just changed the kmerlen variable in the script to 90 and added [::3] to the end of both keys so that it takes every second item from the lists.

Like this: key = seq1[i:i+kmerlen][::3]

There were 8887 hits and 93.85619% of them are on diagonal.

iv. Modify the script to find all 120-mers that match every fourth base

I just changed the kmerlen variable in the script to 120 and added [::4] to the end of both keys so that it takes every second item from the lists.

Like this: key = seq1[i:i+kmerlen][::4]

There were 6044 hits and 82.13104% of them are on diagonal.

v. Modify the script to find all 100-mers that allow at most two mismatches in each contiguous block of six bases. Instead of producing a plot, focus on describing how you would implement this modification.

I would use sliding window approach to get matches between 100-mers so that in each contiguous block of six bases would allow at most two mismatches. But with this approach, we’ll only be able to get diagonal hits. However, this is not the big problem, because as we already saw, in 100-mers, 100% of them are diagonal hits anyways. So we won’t need to modify the approach to search in the whole genome, we’ll only need to adjust it so that the small margin around the diagonal also gets included.

1. Although parts a, b.ii, b.iii, and b.iv require the same number of matching bases (30 = 60/2 = 90/3= 120/4), one of them is more specific to the diagonal. Explain why this might be so.

The most specific was b.iii, and I think the reason may be that in that case we get every third base and as we know, each amino acid is coded by three consecutive nucleotides, called triplet. By taking every third base, we make sure to get one from every triplet, so the matches found with these parameters are more precise.

1. Explain the trade-off you see between number of hits near the diagonal (sensitivity) and the percentage of hits near the diagonal (specificity). How is the trade-off affected by the hashing parameters?

When we increase the length of k-mers, we consequentially get smaller number of hits, but the higher percentage of them being diagonal. This can be because in the larger segments, the chance of getting “accidental”, incorrect matches are significantly smaller, in other words, hits in the 100-mers for example are much more likely to be correct. Because of that, the number of hits in general reduces, hence the number of hits near the diagonal (sensitivity) also reduces, however, specificity increases, because above-mentioned “correct” hits often appear near the diagonals, because it means their locations also match, which can be a sign that their function is the same. So the contribution of diagonal hits is bigger in this case and their percentage increases.

1. Modify the script to also detect inversions. An inversion occurs when a stretch of DNA is spliced out

and reinserted in reverse orientation. For example,

CGT[GATT]AGA

⇓

CGT[AATC]AGA

To find inversions, we would need to change the rule of generating keys, but only in one of the sequences. Instead of getting part of it directly (slice of list), we first reverse it and then get the complementary bases. Just like in the above example, if the segment is GATT, our key would be AATC (reverse gets us TTAG, and then taking complements will give AATC).

After that, we need to choose parameters with greedy approach, by trying out all the possible lengths of keys and matching criteria. (which bases should be the same in order to allow it to count as “match”)

The **human-hoxa-region-modified.fa** file contains a version of the Hox region with an artificial inversion. Use the dotplot to locate the inversion in human. (Note: ignore the sensitivity measure, and only test all sizes necessary to detect the inversion.)

1. **HMMs for GC-rich regions: State durations and limitations**

An important use of HMMs is to decode or parse a genome into its biological components: exons, introns,

regulatory regions, etc. In this problem, we will examine how the accuracy of HMM predictions is affected by certain inherent properties of the model.

In this problem, we will use GC content (the fraction of letters that are a C or a G) to classify the genome

into high-GC regions (on average 60% G or C) and Low-GC regions (on average 60% A or T). These have different melting temperatures, different replication times across the cell cycle, and different gene density. They have also been hypothesized to have different evolutionary origins (see isochores), but this hypothesis remains controversial.

Our simple model requires only two states. We have provided a program, viterbi.py, which you will complete and use to decode several artificial genomes, and then compare the resulting predictions of High-GC and Low-GC regions to a provided (correct) annotation. More details about this program are included at the end of the problem.

(a) In most HMMs, the self-loop transition probabilities are large, while the transition probabilities between different states are small. Once a Markov chain with these transition probabilities enters state , it tends to stay in state for a while. The state duration is the total number of consecutive steps at which the Markov chain stays in the same state, before switching to another state (e.g. transitioning into state and then transitioning out to a different state is a state duration of 1). What is the expected (mean) state duration of state as a function of the transition probability ? What is the distribution of state durations ?

Expected state duration of state k can be computed like this:

If the duration is d, it means that for (d-1) steps, we stayed in same state and on the d-th step we moved to any other one.

Probability of this event is like this: (1-) \* ^ (d-1)

Mean can be computed by taking the sum of the above expression multiplied by d, where d gets values from 1 to infinity.

So like this: E=sum(d \* (1 – ) \* ^ (d-1)) (d = 1 to ∞)

This sum converges to E=1/(1-)

The distribution of state durations is actually written above already. P( (1-) \* ^ (d-1). As I’ve already mentioned, we have to stay in the same state for (d-1) times (^ (d-1)), and then transition to any other one on the final step. ( multiplication by (1-) )

(b) Complete the implementation of the Viterbi algorithm in **viterbi.py.** Based on the HMM parameters hard-coded into the program, what are the expected state durations for High-GC and Low-GC regions? Apply the finished program to the data file hmmgen, which was generated using the same HMM, and verify that your program achieves ∼83% accuracy.

83.31% accuracy.

for both of them is 0.99, so expected state duration is 1/1-0.99=100.

(c) Now apply your program to the files mystery1, mystery2, and mystery3. How do the (correct) state duration distributions in the mystery sequences differ and what do they have in common? What accuracy levels does your HMM achieve on these sequences? How does each Viterbi-predicted state duration distribution differ from the correct distribution? (You do not need to include the plots in your solutions.)

Mystery1: accuracy 71.96%

Mystery2: accuracy 68.8%

Mystery3: accuracy 67.72%

State durations actually differ from Viterbi-predicted ones. The latters seem to be larger than the correct distribution, but if the correct region lengths of High-GC and Low-GC are close to one another, so are the ones predicted by our algorithm. Also, base compositions seem to be pretty close to correct data.

(d) Would re-training the HMM parameters according to the procedure described in lecture, using the correct annotations as training data, improve the accuracy of the Viterbi annotation for the mystery sequences? Why or why not?

(Extra credit) Try to make the decoder perform better by adjusting the hard-coded model parameters. If you succeed, can you explain why?

Theoretically, yes. By using the correct annotations as training data, we are going to have more accurate and precise estimations for probabilities used in algorithm. These are emission and transition probabilities. This is why improving them by re-training using the correct annotations will most likely improve our accuracy in the end.

(e) As you are now aware, the length distribution of genomic elements can strongly affect the predictive accuracy of an HMM used to decode them. Unfortunately, most elements in real genomes do not follow the length distribution you derived in part (a).

**Details about viterbi.py**

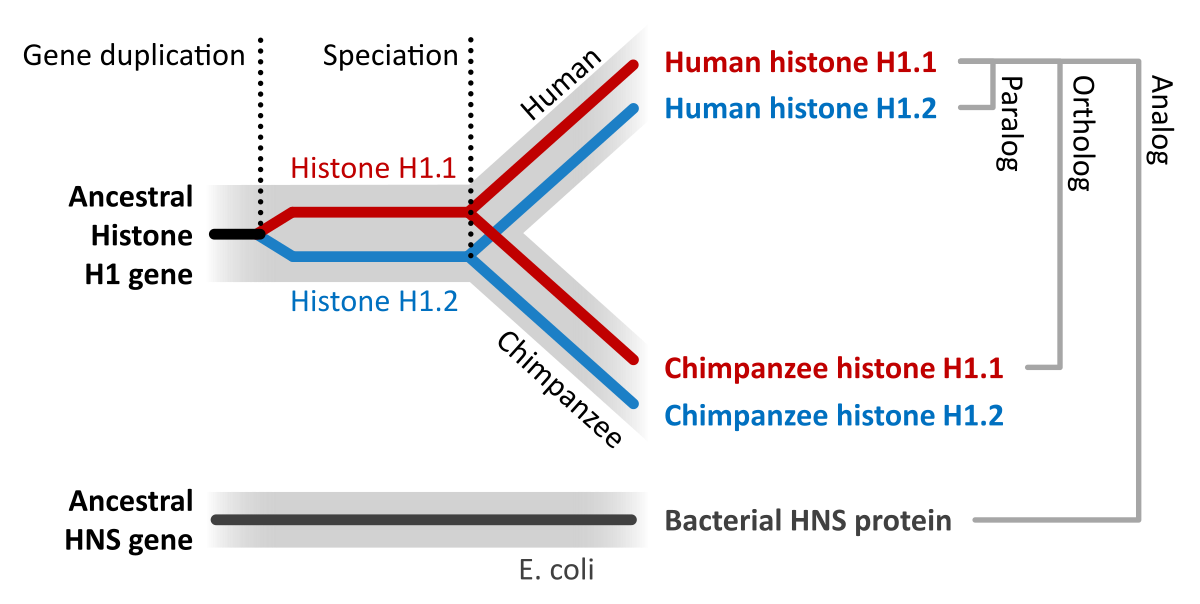
Note that like in problem set 1, the plotting portion of this code relies on gnuplot. Therefore, you should run this on athena after running “add gnu” if you want plotting to work.

The nearly complete program **viterbi.py** performs the following:

* Reads in a data file containing a DNA sequence and an authoritative (correct) annotation, consisting of a string of pluses and minuses, specifying where the High-GC and Low-GC regions are, respectively.
* Calculates the base composition of the High-GC and Low-GC regions, calculates the mean length of High-GC and Low-GC regions, and plots a histogram of the lengths of the High-GC and Low-GC regions. (All with respect to the authoritative annotation.)
* Performs Viterbi decoding on the DNA sequence, using a hard-coded HMM designed to detect High-GC and Low-GC regions. (This is the part you will complete.)
* Calculates the base composition of the High-GC and Low-GC regions, calculates the mean length of High-GC and Low-GC regions, and plots a histogram of the lengths of the High-GC and Low-GC regions. (All with respect to the Viterbi annotation.)
* Calculates the accuracy of the Viterbi decoding, defined as the percentage of predicted plus and minus states that match the authoritative annotation.

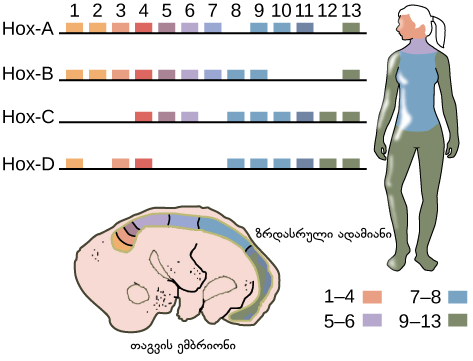
**დამატებითი ინფორმაცია**

**Orthologs** are homologous genes in different species that diverged from a single ancestral gene after a speciation event and **paralogs** are homologous genes that originate from the intragenomic duplication of an ancestral gene.



***Hox* გენები**

ხერხემლიანებში, მაგალითად, ადამიანებსა და თაგვებში, *Hox* გენები დუპლიცირდა ევოლუციური ისტორიის განმავლობაში და დღეისთვის მსგავსი ტიპის გენების ოთხ კლასტერად იყოფა, A-დან D-მდე:



ზოგადად, განსხვავებული კლასტერების გენები ერთად მოქმედებენ, რათა სხეულის სეგმენტების იდენტობები განსაზღვრონ თავისა და კუდის ღერძის სიგრძეზე. ეს ნიშნავს, რომ კლასტერის დასაწყისში მყოფი გენები — თანრიგობით 1-თან ახლოს დიაგრამაზე — როგორც წესი, განსაზღვრავენ ორგანიზმის თავის ნაწილის სტრუქტურებს, ხოლო კლასტერის ბოლოს არსებული გენები — თანრიგობით 13-თან ახლოს დიაგრამაზე — სხეულის კუდის ნაწილის სტრუქტურებს განსაზღვრავენ.

მიუხედავად ამისა, გენის დუპლიკაციის მეშვეობით ზოგიერთმა *Hox* გენმა უფრო სპეციალიზებული როლები მიიღო. მაგალითად, კლასტერის ბოლოში მყოფი მრავალი *Hox* გენი სპეციფიკურად ხერხემლიანების კიდურების — ხელების, ფეხების ან ფრთების — განვითარებაში მოქმედებს, როგორც ეს ზემოთ მოცემულ დიაგრამაზეა ნაჩვენები ქალის მაგალითზე. ადამიანებში *HoxD13* გენის მუტაცია იწვევს გენეტიკურ მდგომარეობას, რომელსაც სინპოლიდაქტილია ეწოდება და რომლის დროსაც ადამიანები იბადებიან ხელის ან ფეხის დამატებითი, ზოგჯერ შერწყმული თითებით.



*Hox* კლასტერი შესანიშნავი მაგალითია იმისა, თუ როგორ შეიძლება, ორგანიზმის განვითარებისთვის საჭირო გენები შემოინახოს და მოდიფიცირდეს ევოლუციის განმავლობაში, მეტადრე მაშინ, როდესაც ისინი დუპლიკაციის გზით კოპირდება. *Hox* გენები აგრეთვე გვიჩვენებს, თუ რამდენად ძლიერი შეიძლება იყოს ორგანიზმის განვითარებისთვის საჭირო გენი, განსაკუთრებით, როდესაც ის ტრანსკრიფციის ფაქტორია, რომელიც მრავალი სამიზნე გენის ჩართვა/გამორთვითაა დაკავებული, რათა განსაზღვრული გენეტიკური „პროგრამა“ გაააქტიუროს.