Grade: 95

Algorithms in computational Biology - HW #2

Submit:
- Ilay Anais,
- Hadar Pur

Problem #1:

non-contiguous substring - is a string obtained by deleting a subset of characters from S.

alignment with omissions (AWO) - is an alignment of two non-contiguous substrings of S and T.

A. A maximum score AWO will contain gaps if the scoring function gives gaps a positive score > 0.

   Lets define the scoring function as:

   - $\sigma(x, x) = 2$
   - $\sigma(x, y) = -2, x! = y$
   - $\sigma(x, -) = \sigma(-, y) = 1$

   An example for that is:

   S = ATGCTA

   T = AGCA

   Maximum AWO alignment:

   S' = ATGCA

   T' = A-GC-A

   So the score for this alignment will be: $\sigma(Alignment(S, T)) = 10$

B. Professor Crude suggested the following algorithm for computing a maximum score AWO:
   1. Run the Needleman-Wunsch algorithm to compute a max-score global alignment (S', T') of S and T.
   2. Eliminate all columns (S'_i, T'_j) in the alignment s.t. $(S'_i, T'_j) < 0$.
   3. Output the resulting alignment.

Claim: Professor Crude's algorithm always returns an AWO

Proof: Assume (in contradiction to the claim) that Professor Crude's algorithm doesn't always return an AWO.

1. We simply run Needleman-Wunsch and receive a max score global alignment, since a substring can also be the entire string we still have an AWO between S and T
2. We eliminate the columns of S' and T' which have a negative score, however since this is all we change, we still have an AWO as it remains being an alignment of substrings of S and T, as we can have omissions in substrings of S and T
3. We output the AWO we created in a and b so we remain with an AWO

In contradiction to our initial assumption that Professor Crude's algorithm doesn't always return an AWO.

C. Lets define 2 sequences:
   - S = AGTT
   - T = ATGT

   Lets define the scoring function as:

   - $\sigma(x,\ x) = 2$
   - $\sigma(x,\ y) = -2,\ x! = y$
   - $\sigma(x,\ -) = \sigma(-,\ y) = -5$

   So if we will run Crude's algorithm:
   - ❖ In step #1, we will get that S' = AGTT, T' = ATGT which are equal to S and T, and has the max score that equals 0.
   - ❖ In step #2, we will remove all columns (S'_i, T'_j) in the alignment that has a negative score.
   - ❖ In step #3, we will get S'' = AT, T'' = AT, with a score that equals 4.

     However, this is not the max score with AWO, which is S' = AGT, T' = AGT, with a max score of 6.

D. We modify the scoring scheme so that:
   - $\sigma'(x,\ x) = \sigma(x,\ x)$
   - $\sigma'(x,\ y) = \sigma(x,\ y),\ x! = y$
   - $\sigma'(x,-) = \sigma'(-,\ y) = 0$

   Then the algorithm is:
   1. Run Needleman-Wunsch on S, T with the scoring function defined above
   2. Remove all columns (S'_i, T'_j) in the alignment s.t. (S'_i, T'_j) less than or equal to 0
   3. Return this as the AWO

Time complexity:

- The time complexity for Needleman-Wunsch is **O(mn) –** computing the value in each cell involves three arithmetic operations and maximization. The traceback operation at the end (recovering the path in the matrix) takes **O(m+n)** additional steps.
- The time complexity for step 2, is in the worst case **O(m+n)** as we simply go over the sequences and remove any columns that have a score less than or equal to 0
- We output the AWO in **O(1)**.

So overall we will get time complexity of **O(mn)**.

Space complexity:

- The space complexity for Needleman-Wunsch is **O(mn) –** matrix A has mn cells and for each cell we hold a number and a pointer
- The space complexity for step 2, is in the worst case **O(2(m+n))** as we simply save the sequences S', T'.
- We output the AWO is **O(2(m+n))** from the previous step

So overall we will get space complexity of **O(mn)**.

E. Lets prove the correctness of the algorithm:
   - First, we need to show that the algorithm always returns a valid AWO A (based on Professor Crude's Algorithm):

     Claim: Our suggested algorithm always returns an AWO

     Proof: Assume (in contradiction to the claim) that our suggested algorithm doesn't always return an AWO.

     1. We simply run Needleman-Wunsch and receive a max score global alignment, since a substring can also be the entire string we still have an AWO between S and T
     2. We eliminate the columns of S' and T' which have a 0 or less than 0, however since this is all we change, we still have an AWO as it remains being an alignment of substrings of S and T, as we can have omissions in substrings of S and T
     3. We output the AWO we created in steps 1 and 2 so we remain with an AWO

     In contradiction to our initial assumption that our suggested algorithm doesn't always return an AWO.

- Second, we need to show that A is the maximum score AWO:

  Claim: AWO A has the maximum score.

  Proof: Assume towards a contradiction that our algorithm does not output the maximum score AWO A.
  Then there exists alignment A' s.t. the score of A' greater the score of AWO A with the same scoring scheme.
  Lets run the first step in the algorithm on S, T, so we will get alignment A with max score for global alignment, that guarantees us that the score will be equal to or greater than 0 (by the scoring function we will prefer gaps instead of having a negative score).
  Due to this algorithm being largely based on Needleman-Wunsch, we can use it's correctness to give the maximum scoring alignment.
  As such the scoring scheme and step 2 guarantees we only have positive columns.
  This is due to non-contiguous substrings not allowing you to add gaps to fill in.
  Due to this, A' must have a greater score than A, however, due to the scoring scheme we maximize the score in A by not penalizing gaps and keeping the scoring scheme as is otherwise, so $\sigma(A') = \sigma(A)$

  In contradiction to our initial assumption that A' has a greater score than A, so our algorithm returns the maximum score AWO.

F. Lets define the new scoring function as:
   - $\sigma(x, x) = 2$
   - $\sigma(x, y) = -2, x! = y$
   - $\sigma(x, -) = \sigma(-, y) = 0$

   Then the algorithm is:
   1. Run Needleman-Wunsch on S, T with the scoring function defined above
   2. Remove all columns (S'_i , T'_j) in the alignment s.t. (S'_i , T'_j) less than or equal to 0
   3. Return this as the longest common non-contiguous subsequences

   <u>Time complexity:</u>
   - The time complexity for Needleman-Wunsch is **O(mn) –** computing the value in each cell involves three arithmetic operations and maximization. The traceback operation at the end (recovering the path in the matrix) takes **O(m+n)** additional steps.
   - The time complexity for step 2, is in the worst case **O(m+n)** as we simply go over the sequences and remove any columns that have a score less than or equal to 0
   - We output the longest common non-contiguous subsequences in **O(1)**.

   So overall we will get time complexity of **O(mn)**.

Space complexity:

- The space complexity for Needleman-Wunsch is **O(mn) –** matrix A has mn cells and for each cell we hold a number and a pointer
- The space complexity for step 2, is in the worst case **O(2(m+n))** as we simply save the sequences S', T'.
- We output the common non-contiguous subsequences is **O(2(m+n))** from the previous step

So overall we will get space complexity of **O(mn)**.

Lets prove the correctness of the algorithm:

- First, we need to show that the algorithm always returns a valid common non-contiguous subsequences A:

  Claim: Our suggested algorithm always returns a common non-contiguous subsequence

  Proof: Assume (in contradiction to the claim) that our suggested algorithm doesn't always return a common non-contiguous subsequence.

  1. We simply run Needleman-Wunsch and receive a max score global alignment, since a substring can also be the entire string we still have an alignment between S and T
  2. We eliminate the columns of S' and T' which have a 0 or less than 0, however since this is all we change, we still have an common non-contiguous subsequences as it remains being an alignment of substrings of S and T, as we can have omissions in substrings of S and T
  3. We output the alignment we created in steps 1 and 2 so we remain with a common non-contiguous subsequence.

  In contradiction to our initial assumption that our suggested algorithm doesn't always return a common non-contiguous subsequence.

- Second, we need to show that A is the the longest common non-contiguous subsequence

  Claim: The common non-contiguous subsequence A is the longest.

  Proof: Assume towards a contradiction that our algorithm does not output the maximum score longest common non-contiguous subsequences A.
  Then there exists alignment A' s.t. the length of A' is longer than A.
  Lets run the first step in the algorithm on S, T, so we will get alignment A with max score for global alignment, that guarantees us that the length will be the longest common subsequence,due to the scoring function we will prefer gaps instead over mismatches.

In step 2 we have 3 options, due to the fact that the scoring scheme will make the algorithm not penalize gaps and penalize mismatches, as well as reward only matches:

1. A will contain only gaps, so A will be an empty alignment with a score of 0.
2. A will contain some gaps and matches, so after this step, A' will be a substring that contains only matches.
3. A will contain only matches, so A will be the alignment that is equal to the alignment that we got from the NW algorithm in step 1.

As such the scoring scheme and step 2 guarantees we only have matching columns.
Due to this, A' must have a greater number of matches than A as it is longer, however, due to the scoring scheme we maximize the number of matches in A by not penalizing gaps and rewarding matches only, so |A'| = |A|

In contradiction to our initial assumption that A' is longer than A, so our algorithm returns the longest common non-contiguous subsequences.

Problem #2:

This question deals with the problem of finding common substrings between two sequences S and T. We do this by considering alignments between the two sequences that do not contain gaps, but may contain mismatches. For two sequences, x and y, of the same length we define by d(x,y) the number of positions i in which $x_i \neq y_i$.

A. Lets define an algorithm that finds completely matching substrings with no mismatches.

Input:

Two sequences $S_{i..n}$, $T_{j..m}$ over the same alphabet Σ.

Initialization:

Initialize a value **max_length** = 0 to keep the length of the longest currently found common substring.
Initialize a temporary value **current_length** to keep the length of the current substring.
Initialize an empty string **A** to keep the longest common substring.
Initialize an empty string **B** to keep the current substring

Main loop:

For each $i = 0...n$:
    current_length = 0
    For each $j = 0...m$ and $k = i...n$:
        If $S_k == T_j$:
            Increase **current_length** by 1
            Add S_k to the end of **B**
        Else if **current_length** > **max_length**:
            Set **max_length** = **current_length**
            Set **A** = **B**
            B = “”
            current_length = 0
        Else:
            B = “”
            current_length = 0
    if **current_length** > **max_length**:
        Set **max_length** = **current_length**
        Set **A** = **B**
        B = “”
        current_length = 0

Output:

A which is the longest substring.

Time complexity:

- Initialize all the parameters in $O(1)$.
- The time complexity for the main loop is $O(mn^2)$ – running on both sequences and checking that each character is equal in both of them
- The time complexity for output is $O(1)$

So overall we will get time complexity of $O(mn^2)$.

Space complexity:

- The space complexity $O(m+n)$ – the string that keeps common substrings between 2 sequences.

So overall we will get space complexity of $O(m+n)$.

B. Lets define 2 sequences S and T:

- S = ACGATATATAAGTT
- T = ACGTAATTTAACTT

	A	**C**	**G**	**A**	**T**	**A**	**T**	**A**	**T**	**A**	**A**	**G**	**T**	**T**
A	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C	1	2	2	2	2	2	2	2	2	2	2	2	2	2
G	1	2	3	3	3	3	3	3	3	3	3	3	3	3
T	1	2	3	4	4	4	4	4	4	4	4	4	4	4
A	1	2	3	4	5	5	5	5	5	5	5	5	5	5
A	1	2	3	4	5	6	6	6	6	6	6	6	6	6
T	1	2	3	4	5	6	7	7	7	7	7	7	7	7
T	1	2	3	4	5	6	7	?	?	?	?	?	?	?
T	1	2	3	4	5	6	7	?	?	?	?	?	?	?
A	1	2	3	4	5	6	7	?	?	?	?	?	?	?
A	1	2	3	4	5	6	7	?	?	?	?	?	?	?
C	1	2	3	4	5	6	7	?	?	?	?	?	?	?
T	1	2	3	4	5	6	7	?	?	?	?	?	?	?
T	1	2	3	4	5	6	7	?	?	?	?	?	?	?

We cannot compute the ? due to the fact that we don't know how many mismatches we've had until now, and even if we did, we don't know from where to begin the sequence with 1 or 0 mismatches.
The matching substrings with at most 2 mismatches is actually:

- S' = ATATAAGTT
- T' = ATTTAACTT

C. Lets define an algorithm that finds matching substrings with at most 2 mismatches.

Input:

Two sequences $S_{i..n}$, $T_{j..m}$ over the same alphabet Σ.

Initialization:

Initialize an empty **matrix A** with rows index $i = 0...n$ and columns indexed $j = 0...m$ and initialize A[i][j] = 0
Initialize **mismatch** = 2
Initialize **current_mismatch** = 0
Initialize **max_length** = 0
Initialize **pointer_j** = 0
Initialize **pointer_k** = 0

Note: If we want to get the alignment. We need to save a pointer to the position i, j for the max length.
pointer_max_j = 0
pointer_max_k = 0

Main loop:

```
For each i = 0...n:
    current_mismatch = 0
    For each j = 0...m and k = i...n:
        If S_k == T_j:
            If k == 0 or j == 0:
                A[k][j] = 1
            Else:
                A[k][j] = A[k-1][j-1] + 1
                If A[k][j] > max_length:
                    max_length = A[k][j]
                    pointer_max_j = j
                    pointer_max_k = k
        Else if current_mismatch < mismatch:
            If current_mismtach == 0:
                pointer_j = j
                pointer_k = k

            current_mismatch += 1

            If k == 0 or j == 0:
                A[k][j] = 1
            Else:
                A[k][j] = A[k-1][j-1] + 1
                If A[k][j] > max_length:
```

```
                                        max_length = A[k][j]
                                        pointer_max_j = j
                                        pointer_max_k = k
                    Else if current_mismatch == mismatch:
                            j = pointer_j
                            k = pointer_k
                            A[k][j] = 0
                            current_mismatch = 0
```

Output:

max_length (and the pointers to **pointer_max_j, pointer_max_k** to get the longest substring with K mismatches).

The modification you had to apply to address the issue raised in (b):
We added pointers to where we started counting mismatches, and added a counter to see how many mismatches we got before we had to go back, we also changed how the matrix works to calculate the length of the sequence, and saved a pointer to the cell with the max length.

Proof of correctness:

- First, we need to prove that the suggested algorithm computes a common substring with 2 mismatches.

  Claim: Our suggested algorithm does compute a common substring with a maximum of 2 mismatches.

  Proof: Assume towards a contradiction that our algorithm does not compute a common substring with a maximum of 2 mismatches.

  So we either compute common substring with more than 2 mismatches or a non - common substring.
  - ➢ On the one hand, since we only go over both strings S, T and pull letters from them until we reach 2 mismatches this cannot be the case for us to compute common substring with more than 2 mismatches.
  - ➢ On the other hand, since we only go over both strings S, T and pull letters from them until we reach the end of those strings, this cannot be the case for us to compute a non - common substring.

  In contradiction to our initial assumption that our algorithm does not compute a common substring with a maximum of 2 mismatches.

- Second, we need to prove that the suggested algorithm computes the longest common substring with a maximum of 2 mismatches.

  Claim: Our suggested algorithm does compute the longest common substring with a maximum of 2 mismatches.

  Proof: Assume towards a contradiction that our algorithm does not compute the longest common substring with 2 mismatches.

  So we either compute common substring with more than 2 mismatches or a not the longest common substring.
  - ➢ On the one hand, the proof from the previous claim guarantees us that the algorithm computes a common substring with max of 2 mismatches.
  - ➢ On the other hand, since we only go over both strings S, T and save the max length that we compute in each step, this cannot be the case for us to compute a different substring than the longest common substring.

  In contradiction to our initial assumption that our algorithm does not compute the longest common substring with a maximum of 2 mismatches.

Problem #3:

When given a model for homology H with parameters θ_H,
$sumHomology(S,T) = \Sigma P(A|H,\theta_H)$ - where the sum is taken over all valid alignments A of S and T.

A. **Algorithm:**

Input:

Two sequences $S_{1..n}$, $T_{1..m}$ over the same alphabet Σ and parameters θ_H.

Definition of $sumHomology(S,T)$**:**

$sumHomology(S,T) = \Sigma P(A|H,\theta_H) =$
$= sumHomology(S',T') + sumHomology(S',T) + sumHomology(S,T')$

S' and T' are the prefix of S and T.

The probabilities:

- The probability of the last column (S_i, T_j) in the alignment is $sumHomology(S',T') = A[i-1,j-1] * P(S_i, T_j)$
- The probability of the last column $(S_i, -)$ in the alignment is $sumHomology(S',T) = A[i-1,j] * P(S_i, -)$
- The probability of the last column $(-, T_j)$ in the alignment is $sumHomology(S,T') = A[i,j-1] * P(-, T_j)$

Initialization:

Initialize an empty **matrix A** with rows index $i = 0...n$ and columns indexed $j = 0...m$ and initialize A[i][j] = 1 , when i or j equals to 0, otherwise initialize A[i][j] = 0.

Main loop:

For each $i = 1...n$ *and* $j = 1...m$:

$A[i][j] = sumHomology(S_i,T_j) = A[i-1,j-1] * P(S_i, T_j) +$
$A[i-1,j] * P(S_i, -) + A[i,j-1] * P(-, T_j)$

Output:

Sum of all valid alignments A of S and T.

Complexity:
Since we go over the matrix and compute for each cell A[i][j] the probability, the complexity will be O(mn).

Correctness:
Like NW algorithm, which is an algorithm that finds the max score alignment when given 2 sequences S and T, we create a matrix A m*n, and compute for each cell A[i][j] the max score alignment.

In our algorithm we do something similar, but instead of max score alignment, we find a sum of homology which is done by computing for each cell A[i][j] the probability of that column to exist in alignment A.

Both algorithms use a scoring function to compute A[i][j] by predicting match, mismatch or a gap in the alignment A.

B. Link for google colab:
https://colab.research.google.com/drive/1kchWri1KEx2SqHvsHH7Bx5zBRbu4Anf6?usp=sharing
We added the .py file just in case you can't open the Google collab link.

```
sumHomology is  8.748608389104158e-35 for
S =  ATAAGGCATTGACCGTATTGCCAA
T =  CCCATAGGTGCGGTAGCC
```

C. sumHomology:

```
sumHomology reached out to 0 after  144  times for
S =  TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTGATTAAGCCAAGGTTCCCCG
T =  CCCATAGGTGCGGTAGCC
```

global score:

```
score is  -431.0 for
S =  TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTGATTAAGCCAAGGTTCCCCG
T =  CCCATAGGTGCGGTAGCC
```

In computing the probability of the most likely alignment, we multiply the probabilities which means that when we do the log of them we can add them, however when we do sumHomology, since we multiply the parameters with the previous result, we would be doing logs of logs, meaning we wouldn't get a similar answer necessarily.

Comment:
Q1.a -1 a(x,y)>=0 for x xor y = "-".

Q2.c -4 you have to save 3 matrices for zero, one and two mismatches in order to get an optimal solution.