Grade: 90

Algorithms in computational Biology - HW #1

Submit:
- Ilay Anais,
- Hadar Pur

Problem #1:

numAlign(n,m) - the number of valid pairwise (global) sequence alignments of two sequences of length m and n.

We need to prove that:

$$((m+n)\ choose\ m)\ < numAlign(n,m)\ < ((m+n)\ choose\ m)^2$$

Proof:

→ Let's prove that: $((m+n)\ choose\ n)\ < numAlign(n,m)$

For example let's define S, T 2 sequences, S with length n and T with length m where:

S = TACG T = TC

So there are multiple alignments possible, $((m+n)\ choose\ m)$ relates to the ones with only gaps, no mismatches or matches.

In the sequences that we define n=4, m=2 so $((m+n)\ choose\ m)\ =\ 15$ in which a few possible alignments are:

S' =	- - TACG	- T - ACG	- TA - CG
T' =	TC - - - -	T - C - - -	T - - C - -

We now define a function f from $((m+n)\ choose\ n)$ # of alignments to the full set of alignments, which is 1-1 as no two alignments x and y map to the same alignment A unless they are the same alignment. Therefore since there are more total alignments than the number of alignments with only gaps
=> $((m+n)\ choose\ m) <\ numAlign(n,m)$

← Let's prove that: $numAlign(n,m)\ < ((m+n)\ choose\ m)^2$

As in the previous direction $(m+n)\ choose\ m)$ is the number of alignments with only gaps, that squared is the number of alignments if each gap we consider as a separate character (gap1, gap2, … s.t. The order matters)

For example let's define S, T 2 sequences, S with length n and T with length m where:

S = TACG T = TC

In the sequences that we define n=4, m=2 so $((m+n)\ choose\ m)^2 = 225$ in which a few possible alignments are:

S' =	TACG	TACG	TACG	TACG	T2A3CG	T1A4CG
T' =	T1C2	T2C1	T12C	T21C	1T2C45	2T2C53

We can see that there are far fewer than 225 alignment that includes gaps matches and mismatches as there is a 1-1 function F from the alignments to the set of $((m+n)\ choose\ m)^2$ s.t. F(alignment) = the same alignment with it's gaps in the numbering from lowest to highest in lexicographical order

F is 1-1 as each alignment is only mapped to 1 alignment with numbers, and no alignment with numbers is ever mapped to by two different alignments, assume towards contradiction that x ≠ y map to the same A, then it's gaps must be in different positions, but since all the gaps in A are supposed to be in the same places in both x and y so contradiction => x = y.

As such since the total number of alignments is less than the number of alignments with numbers instead of gaps

=> $numAlign(n,m) < ((m+n)\ choose\ m)^2$

Problem #2:

a) The main reason we cannot use the global alignment technique for local alignment is due to when we need to retrace the pointers, we cannot assume that the local alignment crosses the middle of the matrix at some point, so we cannot split it into two halves and run the process again on smaller sequences.

b) The new algorithm of linear-space for local alignment:

Input: Two sequences $S_{1...n}$, $T_{1...m}$ over the same alphabet Σ scoring function:

$$\sigma : (\Sigma \cup \{-\}) x (\Sigma \cup \{-\}) \rightarrow R$$

Initialization:

- Create two empty arrays of size equal to the smaller sequence
- Initialize $A[i] = B[i] = 0$ and for each $i = 0..n$
- M_S holds the max score of a suffix of $S_{1...i}$ and a suffix of $T_{1...j}$
- M_P holds the position of the max score seen so far

Main loop #1:

- For each i=0..n and j=0..m compute B[i] as follows:

$$B[i] = max\{\ 0\ ;\ A[i-1] + \sigma(S_i,\ T_j)\ ;\ A[i] + \sigma(S_i,\ -)\ ;\ B[j-1] + \sigma(-,\ T_j)\ \}$$

Once we reach i = n copy B to A

Keep a pointer to the “cell” that results in max value M_P

Flipping:

- Write sequences backwards until maximum cell placement, initialize like before M_P will be new max score position

Then once we finish the run and know that M_P is the maximum value cell, we then erase anything past the maximum cell, then we then we compute backwards

Main loop #2: (ascending order of columns/rows)

- For each i=0..n and j=0..m compute B[i] as follows:

$$B[i] = max\{\ 0\ ;\ A[i-1] + \sigma(S_i,\ T_j)\ ;\ A[i] + \sigma(S_i,\ -)\ ;\ B[j-1] + \sigma(-,\ T_j)\ \}$$

Then once we finish the run and know that M_P is the maximum value cell, we then erase anything past the maximum cell

We then run a linear space version of the global alignment on the subsequence cut on both sides, where we are guaranteed to be the shortest maximum score alignment in a matrix C and trace back from both of them

Output: the trace back from both sides

c) **Correctness:**

- If the path in the DP matrix representing a sub-alignment **A** passes through cell A[i , j] then the sub-alignment backwards for A^T will yield the same max score

  proof: simple observation

- If an optimal sub-alignment **A of** $S_{1...n}$, $T_{1...m}$ can be flipped backwards then there will be no sub-alignment with higher score.
  proof: by additivity of alignment score

Complexity:

- Initialization and main loop #1 of the algorithm takes O(nm) time and O(m) space
  Let's denote by Ctime and Cspace the constants behind these O's.
- Flipping the matrix takes O(nm) time and no extra space
- Main loop #2 – takes $C_{time}nm$ time (and $C_{space}m$ space)

Total time: $C_{time}nm$

Total space: $C_{space}m$

Problem #3 - A:

Let S be a genome sequence, and T be a mature mRNA aligned to S.
We treat Introns as subsequent gaps with no penalty to their length, just the number of introns must be penalized.

Let A be a matrix with positions $[i,j]$ s.t. each position is:

- Optimal alignment with a gap in T ($A[i,j|Intron]$)
- Optimal alignment with a match/mismatch ($A[i,j|Exon]$)

Claim #1: if A is an alignment that has max score among alignments of $S_{1...i}$, $T_{1...j}$ that end with gap in T, then the last column of A is $(S_i, -)$, which is preceded by one of the following:

- A max-score alignment of $S_{1...i-1}$, $T_{1...j}$ that ends with a gap in T (residual score of last column is $\varepsilon = 0$)
- a max-score alignment of $S_{1...i-1}$, $T_{1...j}$ that ends with match/mismatch or a gap in S (residual score of last column is δ)

Proof: if A ends with a gap in T, then the last column of A is $(S_i, -)$ and we need to consider the column before last:

Case A:
If it is $(S_{i-1}, -)$, then the alignment preceding the last column of A is an alignment A' of $S_{1...i-1}$, $T_{1...j}$ that ends with a gap in T.
Assume that there is another alignment A'' of $S_{1...i-1}$, $T_{1...j}$ that ends with a gap in T s.t. $\sigma(A'') > \sigma(A')$, and let A''' be the alignment of S and T that results from adding $(S_i, -)$ to A'', then:

$$\sigma(A''') = \sigma(A'') > \sigma(A') = \sigma(A)$$

In contradiction to the optimality of A.

Case B:
If it is (S_{i-1}, T_j) , then the alignment preceding the last column of A is an alignment A' of $S_{1...i-1}$, $T_{1...j}$ that ends with a match/mismatch.
Assume that there is another alignment A'' of $S_{1...i-1}$, $T_{1...j}$ that ends with a match/mismatch s.t. $\sigma(A'') > \sigma(A')$, and let A''' be the alignment of S and T that results from adding $(S_i, -)$ to A'', then:

$$\sigma(A''') = \sigma(A'') + \delta > \sigma(A') + \delta = \sigma(A)$$

In contradiction to the optimality of A.

Case C:
If it is $(-, T_j)$, then the alignment preceding the last column of A is an alignment A' of $S_{1...i-1}$, $T_{1...j}$ that ends with a gap in S.
Assume that there is another alignment A'' of $S_{1...i-1}$, $T_{1...j}$ that ends with a gap in S s.t. $\sigma(A'') > \sigma(A')$, and let A''' be the alignment of S and T that results from adding $(S_i, -)$ to A'', then:

$$\sigma(A''') = \sigma(A'') > \sigma(A') = \sigma(A)$$

In contradiction to the optimality of A.

Update:

$$A[i,j|Intron] = max\{ A[i-1,j|Exon] + \delta ; A[i-1,j|Intron] \}$$

Claim #2: if A is an alignment that has max score among alignments of $S_{1...i}$, $T_{1...j}$ that end with match or mismatch, then the last column of A is (S_i, T_j), and it is preceded by:

- A max-score alignment of $S_{1...i-1}$, $T_{1...j-1}$ (residual score of last column is $\sigma(S_i, T_j)$)

Proof: same as with global alignment with additive score

Update:

$$A[i,j|Exon] = max\{ A[i-1,j-1|Exon] ; A[i-1,j-1|Intron] \} + \sigma(S_i, T_j)$$

Algorithm:

Let S be a genome sequence, and T be a mature mRNA aligned to S which are defined as $S_{1...n}$, $T_{1...m}$

Perform Needleman-Wunsch on matrix A, with the following updates

$$A[i,j|Intron] = max\{ A[i-1,j|Exon] + \delta ; A[i-1,j|Intron] \}$$
$$A[i,j|Exon] = max\{ A[i-1,j-1|Exon] ; A[i-1,j-1|Intron] \} + \sigma(S_i, T_j)\}$$

With a score function of

- $\sigma(x, x) = 0$
- $\sigma(x, y) = \alpha, x! = y, \alpha < 0$
- *affine gap score* : $\delta = -1, \varepsilon = 0$

Complexity:
O(nm) from Needleman-Wunsch algorithm.

Correctness:
Let's define that K is the number of implied introns and L is the number of substitutions.
We need to prove that the alignment of the optimal scores minimizes the K's and L's based on α.
According to claim #1 and claim #2 the algorithm output $max\{\delta * K + \alpha * L\} = max\{-K + \alpha * L\}$, where $\alpha < 0$.

Problem #3 - B:
We do a similar procedure to the affine gap score that we saw in class, We force it to consider every gap as both an indel and an intron until we reach k introns, then we only consider the k largest ones to be introns and the others are considered indels. The update function is similar to 3.A's but we consider the parameter k as the maximum number of introns possible.

Let A be a matrix with positions $[i,j,k]$ s.t. each position is:

- Optimal alignment with a gap in T that allows k introns ($A[i,j,k|Intron]$)
- Optimal alignment with a match/mismatch that allows k introns ($A[i,j,k|Exon]$)

Proof: Similar to the first section on the problem.

Algorithm:

Let S be a genome sequence, and T be a mature mRNA aligned to S which are defined as $S_{1...n}$, $T_{1...m}$

Perform Needleman-Wunsch on matrix A, with the following updates:

$$A[i,j,k|Intron] = max\{A[i-1,j,k-1|Exon] + \delta ; A[i-1,j,k-1|Intron]; A[i-1,j,k|Exon] + \sigma(S_i, T_j)\}$$

$$A[i,j,k|Exon] = max\{A[i-1,j-1,k|Exon] ; A[i-1,j-1,k|Intron]\} + \sigma(S_i, T_j) ; A[i-1,j,k|Exon] + \sigma(S_i, T_j)\}$$

With a score function of

- σ
- $affine\ gap\ score: \delta = \sigma(x, -) = \sigma(-, y), \varepsilon = 0$

Complexity:
O(nmk) from Needleman-Wunsch algorithm.

Problem #4:

Link for google colab:
https://colab.research.google.com/drive/1L6TebaFqUcpBgj4K1Qis4q5qPKUYYne_?usp=sharing
We added the .py file just in case you can't open the Google collab link

Note that numbering start at 0 and (i,j) = column j row i

a) matrix_scores:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	2	0	2	0	0	0	0	0	0	0	0	2	0	0	0
0	0	0	0	0	4	1	0	0	2	0	0	0	0	2	0	0	0	0
0	0	0	0	2	1	6	3	0	0	0	0	0	0	0	4	1	0	0
0	0	0	0	2	0	3	4	1	0	0	0	0	0	0	2	2	0	0
0	0	0	0	0	0	0	5	6	3	2	0	2	2	0	0	4	1	0
0	0	0	0	0	0	0	2	7	4	5	2	2	4	1	0	2	2	0
0	2	2	2	0	0	0	0	4	5	2	7	4	1	2	0	0	4	4
0	0	0	0	4	1	2	0	1	2	3	4	5	2	0	4	1	1	2
0	0	0	0	1	6	3	0	0	3	0	1	2	3	4	1	2	0	0
0	0	0	0	0	3	4	1	0	2	1	0	0	0	5	2	0	0	0
0	0	0	0	0	0	1	6	3	0	4	1	2	2	2	3	4	1	0
0	0	0	0	2	0	2	3	4	1	1	2	0	0	0	4	1	2	0
0	2	2	2	0	0	0	0	1	2	0	3	0	0	0	1	2	3	4
0	2	4	4	1	0	0	0	0	0	0	2	1	0	0	0	0	4	5
0	0	1	2	2	0	0	2	2	0	2	0	4	3	0	0	2	1	2
0	0	0	0	0	4	1	0	0	4	1	0	1	2	5	2	0	0	0
0	0	0	0	2	1	6	3	0	1	2	0	0	0	2	7	4	1	0
0	0	0	0	0	4	3	4	1	2	0	0	0	0	2	4	5	2	0
0	0	0	0	0	2	2	1	2	3	0	0	0	0	2	1	2	3	0
0	0	0	0	0	0	0	4	3	0	5	2	2	2	0	0	3	0	1
0	2	2	2	0	0	0	1	2	1	2	7	4	1	0	0	0	5	2
0	2	4	4	1	0	0	0	0	0	0	4	5	2	0	0	0	2	7
0	0	1	2	6	3	2	0	0	0	0	1	2	3	0	2	0	0	4
0	0	0	0	4	4	5	2	0	0	0	0	0	0	1	2	0	0	1

b) The max_pos is (6, 8) lets trace back from there:
   (5, 7)
   (4, 6)
   (3, 6)
   (2, 5)
   (1, 4)
   (0, 3)

c) We got the local alignment with a subsequence of:
   - S' = ATAAGG
   - T' = ATA-GG

d) Lest observe the matrix and find another max_pos.
The max_pos2 is (22,18), lets trace back from there:
(21, 17)
(20, 16)
(19, 15)
(18, 15)
(17, 15)
(16, 14)
(15, 13)
(14, 12)
(13, 11)
(12, 10)
(11, 10)
(10, 9)
(9, 8)

We got the local alignment with a subsequence of:
S' = TGACCGTATTGCC
T' = TG-CGGTA--GCC

e) Another optimal alignment that overlaps with the alignment in c would be:
S' = ATAAGG
T' = AT-AGG

Due to the fact that it is another valid optimal alignment that has a gap in a different place.

f) We build a hirschberg algorithm for local alignment:

```
def hirschberg_scores(seq1, seq2, match_score, mismatch_score,
gap_score):
 #Create empty arrays A and B with cells indexed minimum of n or m
 #Initialize A and B to 0 values
 A = np.zeros(min(len(seq1)+1, len(seq2)+1), np.int)
 B = np.zeros(min(len(seq1)+1, len(seq2)+1), np.int)

 #pointer for highest score reached and it's position
 max_score = 0
 max_pos   = (len(seq1),len(seq2))

 #we make sure seq2 is the shorter one for looping purposes
 if len(seq1) < len(seq2):
   temp = seq1
   seq1 = seq2
   seq2 = temp

 print(' '.join(['{:4}'.format(item) for item in A]))

 #main loop O(mn) due to each step taking constant time and going over
all i and j is lengths of each sequence + 1
 for i, j in itertools.product(range(1, len(seq1)+1), range(1,
len(seq2)+1)):
```

```
   match = A[j-1] + (match_score if seq1[i - 1] == seq2[j - 1] else
mismatch_score)
   delete  = B[j-1] + gap_score
   insert  = A[j] + gap_score

   score = max(match, delete, insert, 0)

   if score > max_score:
     max_score = score
     max_pos = (i,j)

   if score == max_score and i+j < max_pos[0]+max_pos[1]:
     max_pos = (i, j)

   B[j] = score

   #if we reach end of row
   if(j == len(seq2)):
     #Copy B to A so that we only use two rows at each point
     A = B.copy()
     print(' '.join(['{:4}'.format(item) for item in A]))

 return A, max_pos, max_score
```

- We run hirschberg algorithm for local alignment on S and T,
- Then we cut both seq S and T and do a reverse, so we got S' and T'.
- Then running again hirschberg algorithm for local alignment on S' and T'.
- Then we cut both seq S' and T' and do a reverse, so we got S'' and T''.
- Then we ran the hirschberg algorithm again but for global alignment on S'' and T''.

```
def hirschberg_global_scores(seq1, seq2, match_score, mismatch_score,
gap_score):
 #Create empty arrays A and B with cells indexed minimum of n or m
 #Initialize A and B to 0 values
 A = np.zeros(min(len(seq1)+1, len(seq2)+1), np.int)
 B = np.zeros(min(len(seq1)+1, len(seq2)+1), np.int)

 #pointer for highest score reached and it's position
 max_score     = 0
 max_pos       = None
 middle_pos    = None

 #we make sure seq2 is the shorter one for looping purposes
 if len(seq1) < len(seq2):
   temp = seq1
   seq1 = seq2
   seq2 = temp
```

```
 print(' '.join(['{:4}'.format(item) for item in A]))

 #main loop O(mn) due to each step taking constant time and going over
all i and j is lengths of each sequence + 1
 for i, j in itertools.product(range(1, len(seq1)+1), range(1,
len(seq2)+1)):
   match = A[j-1] + (match_score if seq1[i - 1] == seq2[j - 1] else
mismatch_score)
   delete  = B[j-1] + gap_score
   insert  = A[j] + gap_score

   score = max(match, delete, insert)

   if score > max_score:
     max_score = score
     max_pos = (i,j)

   B[j] = score

   # got the middle section of the sub seq
   if(i == math.ceil(len(seq1)/2) and j == math.ceil(len(seq2)/2) ):
     middle_pos = (i, j)

   #if we reach end of row
   if(j == len(seq2)):
     #Copy B to A so that we only use two rows at each point
     A = B.copy()
     print(' '.join(['{:4}'.format(item) for item in A]))

 assert middle_pos is not None

 return A, max_pos, middle_pos, max_score
```

- After that we cut the 2 sequences S'' and T'' to halves, so we got S''' and T'''.
- Then run the regular global alignment for both halves

```
# Global alignment matrix scores
def matrix_scores_global(seq1, seq2, match_score, mismatch_score,
gap_score):
 #Create empty matrix A with rows indexed 0..n and columns indexed
0..m
 #Initialize A[i,0] = A[0, j] = 0 and for each i=0..n and j=0.. n
 A = np.zeros((len(seq1)+1, len(seq2)+1), np.int)

 max_score = 0
 max_pos   = None

 #main loop
```

```
 for i, j in itertools.product(range(1, A.shape[0]), range(1,
A.shape[1])):
   match    = A[i - 1, j - 1] + (match_score if seq1[i - 1] == seq2[j -
1] else mismatch_score)
   delete   = A[i - 1, j] + gap_score
   insert   = A[i, j - 1] + gap_score

   score = max(match, delete, insert)

   if score > max_score:
     max_score = score

   A[i,j] = score
   max_pos = (i,j)

 assert max_pos is not None

 return A, max_pos, max_score
```

- And then trace back each half matrix, and put it together so we will get the alignment S”” and T”” which is the local alignment that we found in in the beginning:
- S”” = ATAAGG
- T”” = ATA-GG

(NOTE: during the local alignments we printed the “matrix” by printing each row at a time for checking purposes, but we only ever keep two rows in memory so it is still in linear space)

Comment:
Q1 + Q4 Great! :)

Q2. c -3 The purpose here in proving the correctness is to show that indeed the algorithm manages to find the starting and ending points. You have not proved the correct claim.

Q3. b -7 It is not possible... you need k+1 parallel matrices (for each intron you need to save the best alignment).