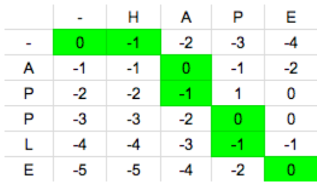
Comp 561 Assignment 1

Jonathan Pearce, 260672004

October 4, 2017

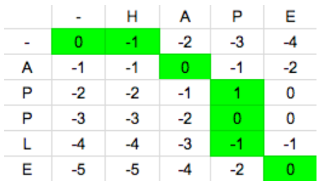
Problem 1. There are 2 optimal alignments

(a) Path 1



	-	H	A	P	E
-	0	-1	-2	-3	-4
A	-1	-1	0	-1	-2
P	-2	-2	-1	1	0
P	-3	-3	-2	0	0
L	-4	-4	-3	-1	-1
E	-5	-5	-4	-2	0

(b) Path 2



	-	H	A	P	E
-	0	-1	-2	-3	-4
A	-1	-1	0	-1	-2
P	-2	-2	-1	1	0
P	-3	-3	-2	0	0
L	-4	-4	-3	-1	-1
E	-5	-5	-4	-2	0

Figure 1: Matrix of optimal alignment with both paths highlighted in green

Path 1:

$$HA{-}P{-}E$$
$$-APPLE$$

3 matches, 3 indels
score $= 3(+1) + 3(-1) + 0(-1) = 0$

Path 2:

$$HAP{-}{-}E$$
$$-APPLE$$

3 matches, 3 indels
score $= 3(+1) + 3(-1) + 0(-1) = 0$

Problem 2.

$$S = ATCG, T = AGTGCTAG$$

Linear Gap Penalty: Optimal Alignment

$$A{-}T{-}C{-}{-}G$$
$$AGTGCTAG$$

4 matches, 4 indels
score $= 4 + (-4) = 0$

Affine Gap Penalty: Same Alignment

$$A{-}T{-}C{-}{-}G$$
$$AGTGCTAG$$

4 matches, 3 indel gaps
score $= 4 + (-2.5 + (-2.5) + (-3)) = -4$

Affine Gap Penalty: Optimal Alignment

$$A{-}{-}{-}{-}TCG$$
$$AGTGCTAG$$

3 matches, 1 indel gap, 1 mismatch
score $= 3 + (-4) + (-1) = -2$

The optimal alignments are different depending on which penalty is applied

Problem 3a. Solution attached below

Problem 3b. Alignment Score produced by my code: 2068

Problem 3c. The subsequence in the question is 32 characters long and in my alignment from (b), this subsequence starts at the 2332nd character in the Mouse BRAC1 gene. Starting at the 2332nd character in the alignment of the Human BRAC1 gene, we see the following subsequence.

$$TTGGTACCTGGTACTGATTATGGCACTCAGGA$$

This subsequence is the same as the one mentioned in the question except for 7 substitutions

Problem 4. Using the Smith-Waterman algorithm with the following scoring scheme and modified trace-back procedure you could find the longest sequence X that is a subsequence of both S and T. The scoring scheme required is as follows: A match is +1, insertion, deletion, transversion and transition mismatches are all 0. The traceback procedure would only need to be modified such that only matches would add a letter to X (the letter would be the same in S and T since it is a match). This scoring system ensures the maximum number of matches are made and thus the longest sequence X is found.

Problem 5. Solution attached below

Problem 6. Solution attached below

Problem 7. The idea is to have two sequences that are identical except for every 5th piece of DNA is different so that no substring of length 5 is the same.

$$AAAAAAAAAAAAAAAAAAAA$$
$$AAAATAAAATAAAATAAAAT$$

Problem Bonus. First, call ForgetThePast-NW and have it traverse up to the m/2 row. Now reverse the DNA sequences and pass them into next ForgetThePast-NW and have it traverse up to the (m/2 - 1) row (this is equivalent to calculating the optimal scoring array in reverse, i.e. starting at position (m,n) and working towards (0,0)). At this point we have found the optimal scores for rows m/2 (starting from (0,0)) and (m/2)+1 (starting from (m,n)). Now we can calculate which crossing from row m/2 to row m/2 + 1 produces the highest alignment score (this is also the final alignment score), and thus we will know the middle 2 pieces of DNA in each alignment. Now that we know the middle pieces of DNA in the final alignment we can split S and T into 2 parts, 1 part is the DNA sequence that occurs before the crossing we just found and 1 part is the DNA after. Using these substrings of S and T we can call ForgetThePast-NW twice as before and find where the optimal path crosses rows m/4 and (m/4)+1 and we can call ForgetThePast-NW twice again and find where the optimal path crosses rows 3m/4 and (3m/4)+1. It now becomes clear that if we recursively find where the optimal path crosses 2 rows in the optimal scoring array (with the method described at the beginning), then divide the DNA sequences based on where that crossing occurs and then repeat this process with our new substrings of DNA we will eventually find all m row crossings which is equivalent to the optimal alignment.

Each recursive call only ever has 2 rows in memory, and each recursive call occurs one after another. Therefore this algorithm requires only $O(m + n)$ space.

3)a)
```
a ← input
b ← input
m = length(S)
n = length(T)

K = [m][n]   // Dynamic programming arrays
X = [m][n]
y = [m][n]
X3 = [m][n]
y3 = [m][n]

①  for i = 0 to m :   // intialize first column
       if i<3 :
           z = i*b
       else :
           z = (i%3)*b + (i - i%3)*a
       K[i][0], X[i][0], y[i][0], X3[i][0], y3[i][0] = z.

   for i = 0 to n   // intialize first row
       if i < 3 :
           z = i * b
       else :
           z = (i%3)*b + (i - i%3)*a
       K[0][i], x[0][i], y[0][i], x3[0][i], y3[0][i] = z

②  for i = 1 to m
       for j = 1 to n
           c = MS(S_i, T_j)   // match score of S_i and T_j
           M[i][j] = max { M[i-1][j-1] + c, X[i-1][j-1] + c,

                 y[i-1][j-1] + c, X3[i-1][j-1] + c, y3[i-1][j-1] + c
```

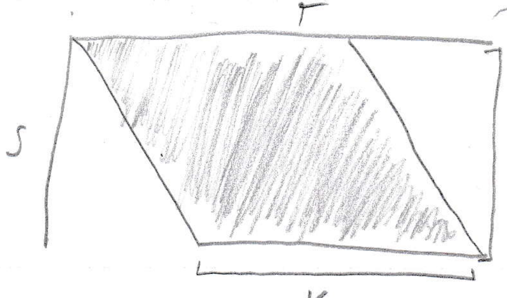
```
x[i][j] = max{ M[i][j-1] + b, x[i][j-1] + b, y[i][j-1] +
                   x3[i][j-1] + b , y3[i][j-1] + b }

y[i][j] = max{ M[i-1][j] + b, x[i-1][j] + b, y[i-1][j] +
                   x3[i-1][j] + b , y3[i-1][j] + b }

if j >= 3 :
   x3[i][j] = max { M[i][j-3] + 3a, x[i][j-3] + 3a, y[i][j-3] + 3
                   x3[i][j-3] + 3a , y3[i][j-3] + 3a }
else :
   x3[i][j] = -infinity
if i >= 3 :
   y3[i][j] = max { M[i-3][j] + 3a, x[i-3][j] + 3a, y[i-3][j] + 3
                   x3[i-3][j] + 3a , y3[i-3][j] + 3a }
else:
   y3[i][j] = -infinity

i = m, j = n
Salign = "   ", talign = "   "

③ while i > 0 or j > 0 :
      bScore = max { M[i][j], x[i][j], y[i][j], x3[i][j], y3[i][j] }

      if bScore == M[i][j] :
         Salign = S_i + Salign
         talign = t_j + talign
         i--
         j--
```

```
else if bScore == x[i][j]:
    Salign = "-" + Salign
    talign = $t_j$ + talign
    j--

else if bScore == y[i][j]:
    Salign = $S_i$ + Salign
    talign = "-" + talign
    i--

else if bScore == x3[i][j]:
    Salign = "---" + Salign
    talign = $t_{j-2}$ + $t_{j-1}$ + $t_j$ + talign
    j -= 3

else:
    Salign = $S_{i-2}$ + $S_{i-1}$ + $S_i$ + Salign
    talign = "---" + talign
    i -= 3

print Salign

print talign
```

Time : $\textcircled{1} + \textcircled{2} + \textcircled{3}$

$= (m+n) + 5(mn) + (m+n)$

$= O(mn)$



5) Algorithm: Reduced NW
- no ↓ move
- Max K → move
- Stay within Shaded region

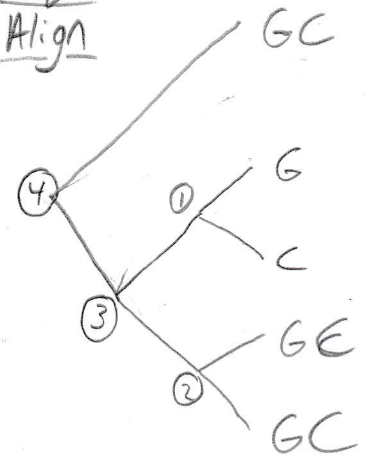
▨ - Section of Matrix optimal score will be calculated

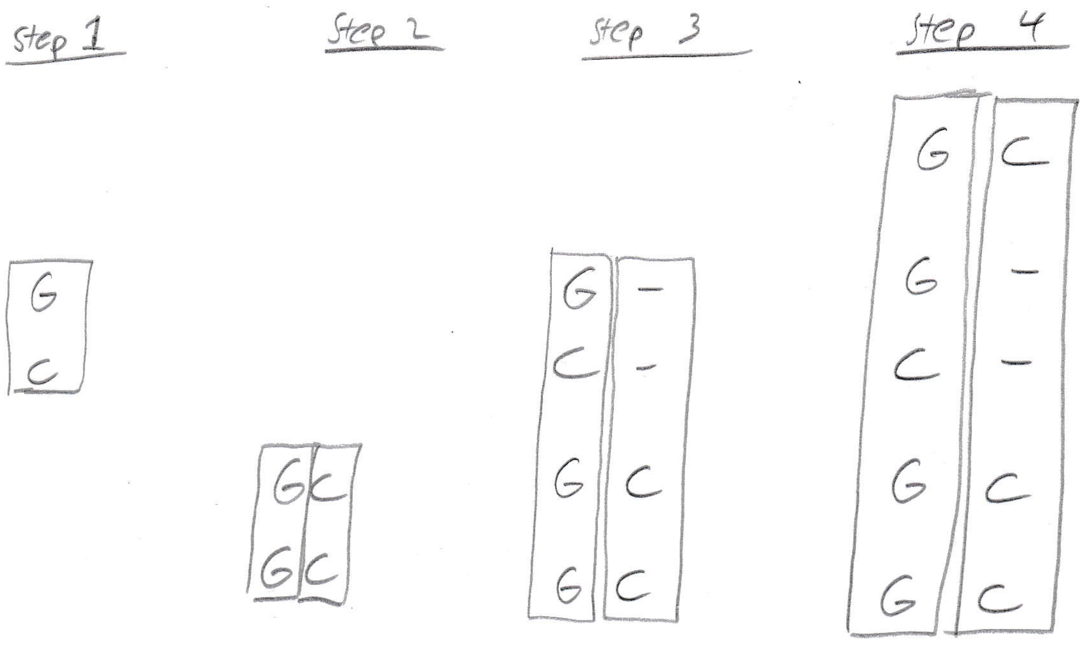
```
C ← input
X = [m][n]

① for i = 0 to K:
      X[0][i] = i · c

② for i = 1 to m:
      for j = i to (i+K):
         if i == j:
            X[i][j] = X[i-1][j-1] + M($S_i$, $T_j$)
         else:
            X[i][j] = max{ X[i-1][j-1] + M($S_i$, $T_j$), X[i][j-1] + c }

Salign = "  ", Talign = "  "
i = m , j = n

③ while i > 0 or j > 0:
      if i == j OR X[i][j] == X[i-1][j-1] + M($S_i$, $T_j$)
         Salign = $S_i$ + Salign
         talign = $t_j$ + talign
         i --
         j --
      else:
         Salign = "-" + Salign
         Talign = $t_j$ + talign
         j--
```

Time = ① + ② + ③ = $O(K) + O(mK) + O(n+m) = O(mK)$

6) Assuming a match is scored + 1

Progressive Seq Align





Scoring

GE	-1	-4	2	2
G –	-2	-1	-1	
C –	-4	-4		
G C	2			
G C				

⇒ Progressive Seq. Alignment Score = -11

Optimal Alignment

G C	-1	-1	2	2
G –	-4	-1	-1	
– C	-1	-1		
G C	2			
G C				

⇒ optimal seq. Alignment Score = -4

⇒ This is an example where progressive multiple Sequence Alignment produces a non optimal result.