COMP561 Assignment 1

by Jingyuan Wang
ID: 260860682

<Question 1>

▵ substitution matrix

M:

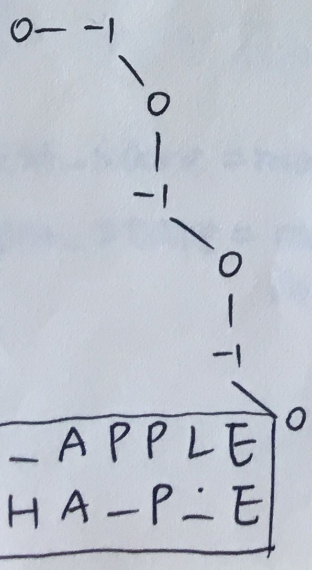
	A	P	L	E	H
A	+1	-1	-1	-1	-1
P	-1	+1	-1	-1	-1
L	-1	-1	+1	-1	-1
E	-1	-1	-1	+1	-1
H	-1	-1	-1	-1	+1

▵ insertion/deletion penalty = -1 * L.

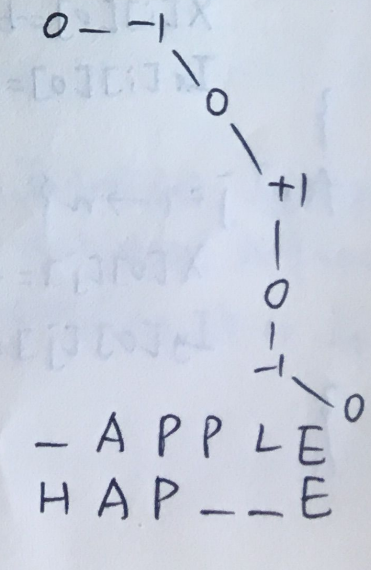
▵ Perform Needleman-Wunsch alg.

∘	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	-1
L	-4	-4	-3	-1	-1
E	-5	-5	-4	-2	0

∴ → Solution 1



→ Solution 2



<Question 2>

Suppose pair of sequences : S: A , T: AAA.

there are 3 good alignments, which are:

① A__ ② _A_ ③ __A

AAA AAA AAA.

a) If we use linear gap penalty cost(L) = -1 * L,

all 3 alignments have score of score(match) - 2,

i.e. ~~we will possibly choose~~ all three alignments are optimal.

b) If we use affine gap penalty cost(L) = -2 - 0.5 * L.

alignment ①&③ will achieve score of score(match) - 3

whereas alignment ② has score of score(match) - 5.

That is alignment ①&③ are optimal.

<Question 3>

a) constraint : $|m-n| \leq \min(m,n)+1$

b) ▵ Input: M, b, S, T.

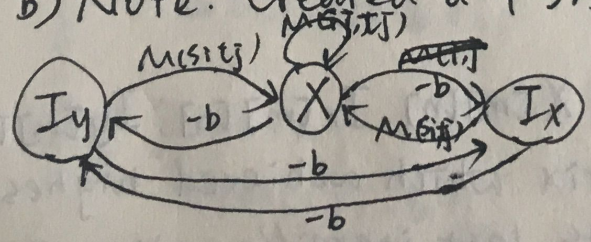
▵ Optimal global alignment(M, b, S, T):

```
StateX, StateIx, StateIy [m+1][n+1] // to store FSA state with each condition
for i = 1 → m {
        X[i][0] = -b*i;
        Ix[i][0] = -b*i;
}
for j = 1 → n {
        X[0][j] = -b*j;
        Iy[0][j] = -b*j
}
```

```
for i=1→m{
    for j=1→n{
        //X is highest score given Si aligned to tj
        X[i][j]=max(X[i-1][j-1]+M[i][j],
                    Ix[i-1][j-1]+M[i][j],
                    Iy[i-1][j-1]+M[i][j]);
        if(X[i][j]=X[i-1][j-1]+M[i][j]) store X→X;
        if(X[i][j]=Ix[i-1][j-1]+M[i][j]) store X→Ix;
        if(X[i][j]=Iy[i-1][j-1]+M[i][j]) store X→Iy;
        //store x→X means StateX[i][j]=X
        //store x→Ix means StateX[i][j]=Ix
        //Store x→Iy means StateX[i][j]=Iy.
        //Ix is highest score given Si aligned to '_'
        Ix[i][j]=max(X[i-1][j]-b, Iy[i-1][j]-b);
        if(Ix[i][j]=X[i-1][j]-b) StateIx[i][j]=X;
        if(Ix[i][j]=Iy[i-1][j]-b) StateIx[i][j]=Iy;
        //Iy is highest score given tj aligned to '_'
        Iy[i][j]=max(X[i][j-1]-b, Ix[i][j-1]-b);
        if(Iy[i][j]=X[i][j-1]-b) StateIy[i][j]=X;
        if(Iy[i][j]=Ix[i][j-1]-b) StateIy[i][j]=Ix;
    }
}
highest_score=max(X[m][n], Ix[m][n], Iy[m][n]);
current_state=matrix which achieved highest_score
              in its last item;//i.e. X or Ix. or Iy.
```

```
// trace back with states (similar to FSA)
i=m; j=n;
while i>0 & j>0 {
    if(current_state = X){
        current_state = State X[i][j];
        align si with tj ;
        i-=1 ; j-=1;
    }
    if (current_state = Ix) {
        current_state = State Ix [i][j];
        align si with '_';
        i-=1;
    }
    if(current_state = Iy){
        current_state = State Iy [i][j];
        align '_' with tj
        j-=1;
    }
if(i=0){ while(j!=0){align tj with '_'; j-=1;}
if(j=0){ while(i!=0){align si with '_'; i-=1;}
print (alignment, highestscore)
```

Q3. b) Note. Created a FSA like:



Q3. c) Please see align.py for code.
and optimal-alignment.txt for optimal alignment and scores.

<Question 4>

Let $X_{i,j}$ be the length of longest common sequence of $s_1 \cdots s_i$ and $t_1 \cdots t_j$.

a) Use Smith Waterman alg to calculate $X_{i,j}$ each time

$$X_{i,j} = \max \begin{cases} X_{i-1,j-1} + M(s_i, t_j) \\ X_{i-1,j} + c \\ X_{i,j-1} + c \\ 0 \end{cases}$$

where we input parameters of this formula.

① $M(s_i, t_j) = \begin{cases} 1 & , \text{where } s_i = \$ t_j \\ -\min(\text{len}(S), \text{len}(T)) & , \text{where } s_i \neq t_j. \end{cases}$

(assign a constant value to $M(s_i, t_j)$ where $s_i \neq t_j$, the value is less or equal to ~~$-\min(\text{len}(S), \text{len}(T))$~~ -2 to ensure $X_{i-1,j-1} + M(s_i, t_j)$ is not larger than 0 when $s_i \neq \$ t_j$)

② $c = 0$

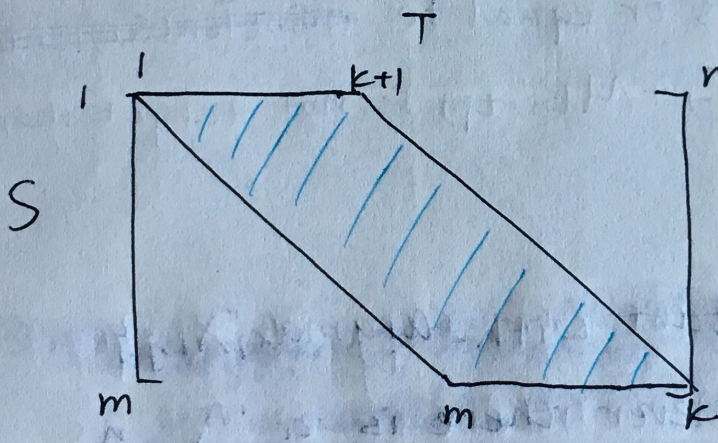
b) When trace back, start with $\arg\max_{i,j}\{X_{i,j}\}$ count s_i or t_j whenever there is a "↖" as part of the common sequence.

c) Other steps are identical with Smith Waterman alg.

<Question 5>

Similar to N-W alg, use $X_{i,j}$ to denote best score of alignment for $s_1 \cdots s_i$ and $t_1 \cdots t_j$.

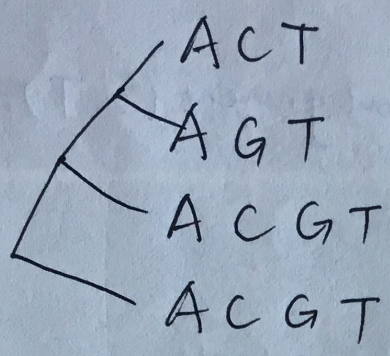
```
for j = 0 → k        // initialization.
    X_{0,j} = -b*j   // b is linear gap penalty.
for i = 1 → m
    for j = i → k+i
        X_{i,j} = max { X_{(i-1,j-1)} + M(s_i, t_j)
                      { X_{(i,j-1)} - b
        // and give "↖" or "←" to the matrix.
// traceback same with N-W.
```



After performing algorithm above, we are recording $X_{i,j}$ inside this parallelogram, which gives base k, height m, and proves the alg is in time $O(m*k)$

<Question 6>

⇒ Set of DNA sequences:



⇒ Alignment by the alg

```
A C _ T
A G _ T
A C G T
A C G T
```

OR

```
A _ C T
A _ G T
A C G T
A C G T
```

score: both -7

⇒ optimal alignment

```
A C _ T
A _ G T
A C G T
A C G T
```

score: -6.

<Question 7>

S: AAAAA CCCCC GGGGG TTTTT

T: AAAAT CCCCA GGGGC TTTTG

< Bonus Question >

To create such an algorithm, we can use Divide & Conquer scheme to find one element of alignment at one time, with following function

```
func divide-and-conquer-alignment(S, T):
    m = len(S);
    n = len(T);
    if(m<=1 or n<=2){
        use Needleman Wunsch alg,
        return alignment
    }
    Perform ForgetPast-NW on (S, T[1:⌊n/2⌋]);
    Perform Backward-ForgetPast-NW on (S, T[⌊n/2⌋+1:n]);
    Suppose the result of last 2 steps is f and g,
    find q that maximize f(i,j)+g(i,j).
    add q into a global alignment table P.
    // begin divide & conquer
    divide-and-conquer-alignment(S[1:q], T[1:⌊n/2⌋])
    divide-and-conquer-alignment(S[q:m], T[⌊n/2⌋:n])
    return P
}
```

Notes: ① Backward-ForgetPast-NW is to consider the [m,n] be the start point, and perform the same alg in the opposite direction

② each time, q is the optimal alignment point we need

③ the size of P is at most $\underline{m+n}$, which takes up space usage of this alg (when no match in nucleotide)