Student Name: <Zuowen Wang>
Student ID : <1155123906>

Q1.
Score table with arrows

s \ *r*	C	C	T	C	C	Ø
G	1	← 3	2	-2	-6	-10
T	0	↖ ← 2	↖ ↑ ← 4	↑ 0	↑ -4	↑ -8
A	-1	↖ ← 1	↖ ← 3	↖ ↑ 2	↑ -2	↑ -6
C	-2	↖ ← 0	↖ ← 2	↖ ↑ ← 4	↑ 0	↑ -4
C	-6	↖ ← -4	↖ ← -2	← 0	↖ ↑ ← 2	↑ -2
Ø	-10	↖ ← -8	↖ ← -6	← -4	↖ ← -2	↖ ↑ ← 0

optimal alignment score: 1
all alignments:
r: _GTACC
s: CCT_CC

r: G_TACC
s: CCT_CC

r: GTACC
s: CCTCC

Q2.

s / *r*	C	C	T	C	C	∅
G	1	← 3	2	0	0	0
T	0	↖ ← 2	↖ ↑ ← 4	0	0	0
A	↑ 2	↖ 1	↖ ← 3	↖ ↑ 2	0	0
C	↑ 4	2	↖ 2	↖ ↑ ← 4	↑ 2	0
C	2	↖ 2	↖ 0	← 2	↖ 2	↖ 0
∅	0	↖ 0	↖ 0	0	↖ 0	↖ 0

alignments:

1. r: 2 TACC 5
   s: 3 T_CC 5
2. r: 4 CC 5
   s: 4 CC 5
3. r: 4 CC 5
   s: 1 CC 2

Q4.

(a)

	1	2	3	4	5	6
s \ r	C	C	T	C	C	∅
G				●	●	●
T					●	●
A						●
C	●					
C	●	●			↖	-2 ↑
∅	●	●	●		-2 ←	0

The entries marked with ● can never be involved in an optimal global alignment between any two length-5 sequences given the +2, -1, -2 scheme.

Proof:

We first consider the naive alignment of two length 5 sequences. Naive alignment simply means we align two length 5 sequences without any indel. Thus, in the worst case, it will give us an alignment score of -5, when every pair of character is a mismatch. The reason we use naive alignment as our criteria is that, an alignment score, which is higher or equals to naive alignment score, can always be achieved no matter which two sequences are given.

We then consider the **best case** if we introduce some indels. If the **best case** of introducing certain number of indels, still, gives us a worse score than the **worst case** of naive alignment, then we can conclude that this alignment will never be the optimal alignment given any two length 5 sequences.

We can see that if we introduce N indels in sequence r, then we must also introduce N indels in sequence s. So the number of indels in either sequence can only be 0,1,2,3,4,5. Notice that the number of indels should be smaller than 6 because otherwise we will have two indels in the same column, which violate the definition of alignment.

Let’s consider the case that we introduced 3 indels in each sequence for now. In the best case, despite that 3 indels, we can get 2 match for the left 2 pairs of characters. This give us an alignment score of (3+3) * (-2) + 2 * (+ 2) = -8. This score is lower than the worst naive alignment score.

If we have 2 indels in each sequences, then we will get (2+2)*(-2) + 2*(+2) = -4 in the best case, which is better than the worst naive alignment score. Thus we conclude that if we have introduced equals to or more than 3 indels in each sequence, we would, for sure, not get the optimal alignment for two length 5 sequences. (under this give scoring schema)

For convenience let’s name alignments with 3 or more indels in each sequence **‘non-optimal alignment’.** And name the region marked with blue dots ‘**non-optimal entries**’

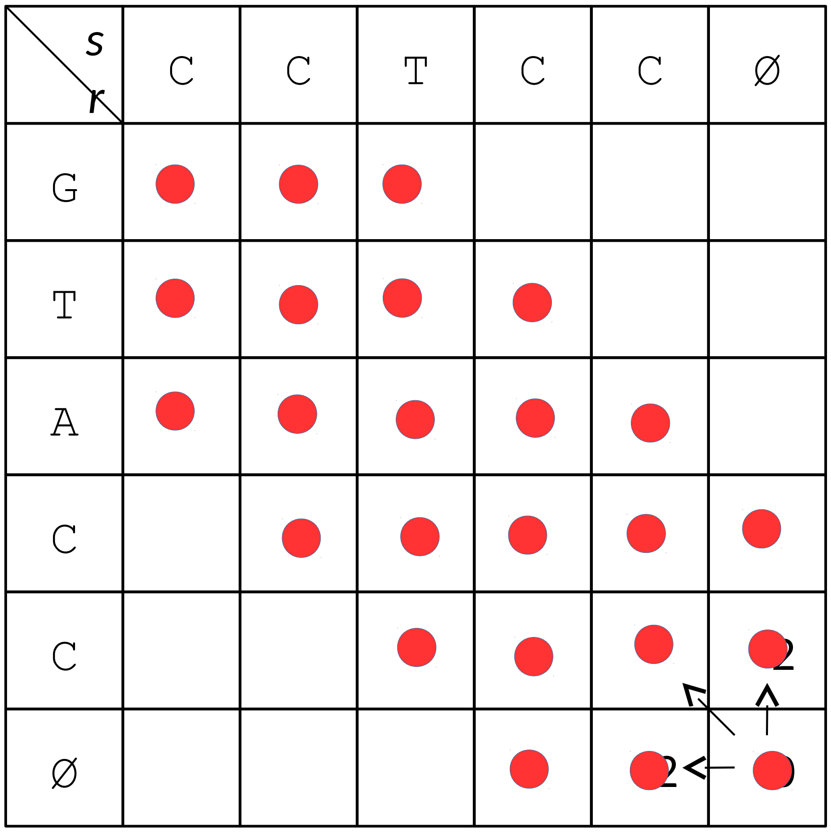
We want to prove that, **ONLY** traces of non-optimal alignments will contain non-optimal entries.
This is equivalent to prove
1. There exists trace of non-optimal alignment which contain non-optimal entries.
AND
2. There does not exist trace of (all possible alignments – (non-optimal alignment)) which contain non-optimal entries.

Statement 1 is very easy to prove. We can simply find an example like r: _ _ _ G T A C C, s: C C T C C _ _ _. It’s trace in dynamic programming is (6,6),(5,6),(4,6),(3,6),(2,5),(1,4),(1,3),(1,2),(1,1).

For statement 2, we have to check that for those alignments which have #indel = 2 or 1 or 0 in each sequence, will not enter the non-optimal entries. This is also easy to see because for each horizontal step or vertical step (corresponds to indel), we will make a transformation from (a,b) to (a, b-1) or (a-1, b). And for each diagonal step (corresponds to match or mismatch) we will have from (a,b) to (a-1, b-1). This indicates we can never reach (a,b) where ((a>b) & ((a-b)>= 3)) or ((a<b) & ((b-a)>=3)) which is the non-optimal entries.

Thus we proved that the non-optimal entries will only be entered by non-optimal alignments. And non-optimal alignments can never be optimal alignments. So the non-optimal entries are the region the task is asking for.

(b)Bounded Dynamic Programming will be used here.



r \ s	C	C	T	C	C	Ø
G	●	●	●			
T	●	●	●	●		
A	●	●	●	●	●	
C		●	●	●	●	●
C			●	●	●	● 2
Ø				●	● 2	● 0

Here we have k = 3 as the bandwidth.

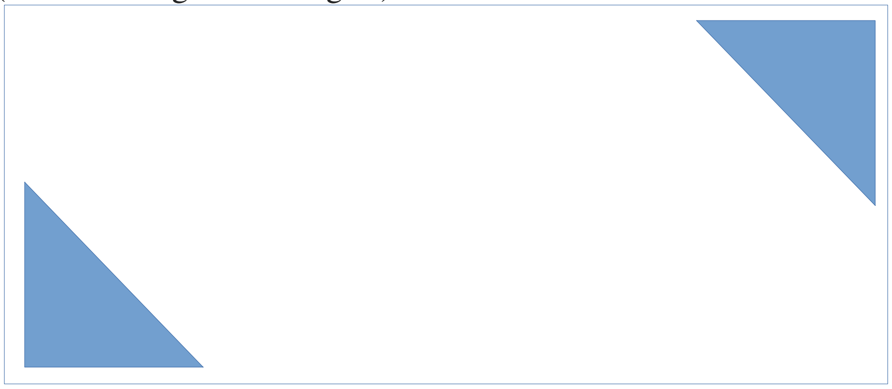
For initialization: initialize DP[i][5] and DP[5][j] for i,j > 5-k = 2 (in java array notation) using the scoring schema

For Iteration:

```
//Following is Pseudo Code, array starts from 0
  int k = 3
  for i in ( 4 … 0 ):
      for j in ( min (4, i + k -1) … max(0, i – k + 1)):
          DP[i][j] = max( DP[i+1][j+1] + s(r[i+1] , r[j+1]),   //diagonal
                          DP[i+1][j] + s(indel), if j>(i-k+1)   //vertical
                          DP[i][j+1] + s(indel), if i>(j-k+1)   //horizontal
                        )
```

some comments: the if in the max() is just take care of the boundary corner cases. We can also initialize the neighboring entries to score minus infinite. After the iteration we read DP[0][0] to get the optimal score as usual.

(c) the region is given as
(a, b) where
case 1: a>b and (a – b) > (n – X)
case 2: a<b and (b – a) > (m – X)
where X = ⌈ *n (Smismatch – Smatch) / (2*Sindel – Smatch)*⌉
(makred using blue triangles)



Q5.
(a)
r=ACGCGTCA

Length -2 subsequences	positions
AC	1
CA	7
CG	2,4
GC	3
GT	5
TC	6

(b)
s=CCGTGCAC

maximal diagonal runs:
1. r[2,3] = s[2,3]
2. r[4,6] = s[2,4]
3. r[3,4] = s[5,6]
4. r[1,2] = s[7,8]
5. r[7,8] = s[6,7]

(c)

```
class Record{
        int r1,
        int r2,
        int s1,
        int s2
}
//define an Record class, each Record object stores 4 integers, r1, r2, s1, s2.
// r1,s1 == start of maximal run, r2,s2 == end of maximal run


LinkedList<Record> maximalRuns = new LinkedList();
maximalRuns.add(0,0,0,0); //just a position taking object, to avoid bug in the
                          //thid for loop (null pointer)

for i in (1 … s.length() - 1):                    // the array starts from 1
        String subSeq = s.substring(i,i+1); // return the substring from i to i+1,
                                            // since we are using 2-mers here.
        for (int occur : lookup(subSeq):    // for each occurrence of subSeq in r
                int m = occur;  //this marks the start of subsequence in r
                for(Record re: maximalRuns):
                        if(re.s2 == i && re.r2 ==  m)  // the new 2-mer can be
                                re.s2++;                //appended to a previous record
                                re.r2++;                //update this record
                        else
                                maximalRuns.add(new Record(m,m+1,i,i+1))
                                                        //link a new record
boolean first = false;
for(Record run :  maximalRuns):
        if(first):   // this is just to skip the first record (a pseudo record)
                r1 = run.r1; r2 = run.r2; s1 = run.s1; s2 = run.s2;
                print the record using the pattern “r[r1,r2] == s[s1,s2]”;
                first = true;
```