CAP 6515 MIDTERM

1 Solution to Question No: 1

1.1 Question I

The steps to build a suffix tree of the string ACAGCTCACAGCTC using Ukkonen's algorithm are attached after the first page of the assignment.

1.2 Question II

If a suffix tree (not implicit suffix tree) is given that stores all the possible suffixes of a string and the original string is unknown then it is possible to retrieve that original string from the given suffix tree in linear time. Here, the length of the string is equal to the number of leaves. The Suffix tree for the string 'banana$' is shown in Figure 1.

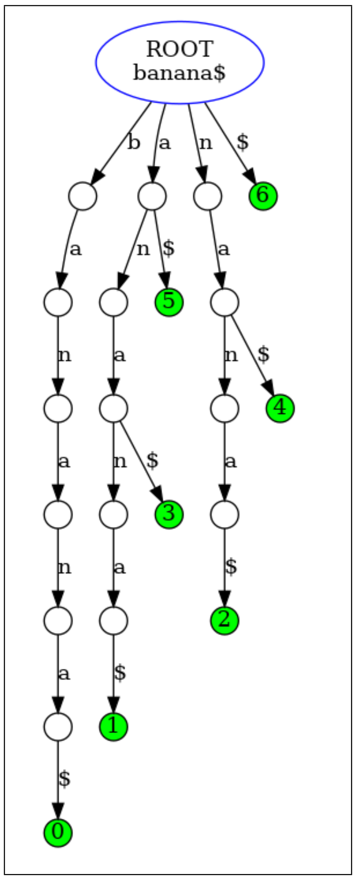


Figure 1: Suffix tree for string 'banana$' (not implicit)

If we traverse the tree starting from root using Depth First Search (DFS) then complexity will be O(V+E) [V= number of nodes, E=number of edge]. In this process, every time we start from root, we can keep a count (at root count = 0) which will keep track of the depth of the path it is traversing currently. Whenever we will come back to root while traversing, we will reset count to 0 and will start traversing a new path. If we find a path that's depth is equal to the length of the string (at it's leaf position) then we know for sure that the characters on this path form our string. So, we don't need to traverse the tree any longer after we find that path. This might reduce the complexity even more. The linear time algorithm is given below [Algorithm 1]:

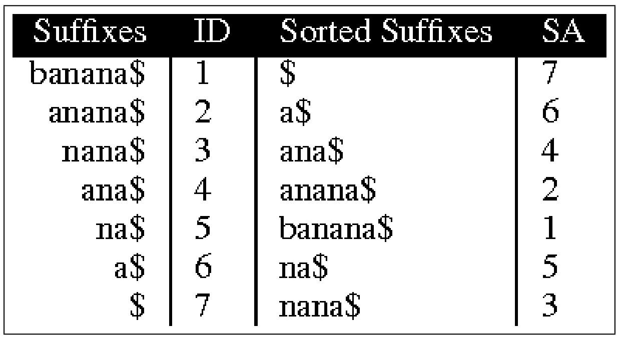
Algorithm 1 Linear Algorithm to Retrieve Original String from Suffix Tree

```
N ← Number of leaves in the Suffix Tree                    ▷ Original string length
S ← NULL                                                   ▷ Unknown Original String
Stack ← Root
Count ← 0
while Stack ! = Empty() do
    Node ← Stack.top()
    if Node = Root then
        Count = 0
    if Node.LeftChild = NULL and Node.RightChild = NULL then
        Stack.pop()
        Count−−
    if Node.LeftChild ! = NULL then
        Stack.push(Node.LeftChild)
        Count++
        Continue
    if Node.RighttChild ! = NULL then
        Stack.push(Node.RightChild)
        Count++
        Continue
    if Count = N then
        S = Current Path
        Break
return S
```

1.2.1 Retrieving Original String from Implicit Suffix Tree:

For retrieving the original string from an implicit Suffix tree, first we have to build the Suffix array from the given Suffix tree in linear time. Then we have to find out which character should be placed at which index of the original string with the help of the Suffix array and Suffix tree. The whole process is described below along with pseudocode.

Suffix Array: A Suffix array holds the starting indexes of all the possible suffixes of a given string in lexicographic order. For example, if the given string is ‘banana$’ then the Suffix array will be build using the process shown in Figure 2 (suppose indexing starts from position 1).



Suffixes	ID	Sorted Suffixes	SA
banana$	1	$	7
anana$	2	a$	6
nana$	3	ana$	4
ana$	4	anana$	2
na$	5	banana$	1
a$	6	na$	5
$	7	nana$	3

Figure 2: Process of Building Suffix array for string ‘banana$’

Building Suffix Array from Suffix Tree: A Suffix array can be built from a Suffix tree in linear time. If we apply Depth First Search (DFS) on the Suffix tree starting from root and in lexicographic order, then it will give us the Suffix array as output. DFS takes linear time O(m+n) where m represents number of nodes and n represents number of edges. It means, we can build the Suffix Array for a string from it’s Suffix tree in linear time.

Suppose we are given a Suffix tree for the string ‘banana$’ as shown in Figure 3. In Figure 3, the values shown in the leaves represent the index of the starting character of the path that leads towards this leaf. By applying DFS in lexicographical order on this tree, we will get the Suffix array (As shown in the 4th column SA of Figure 2) as output in linear time.

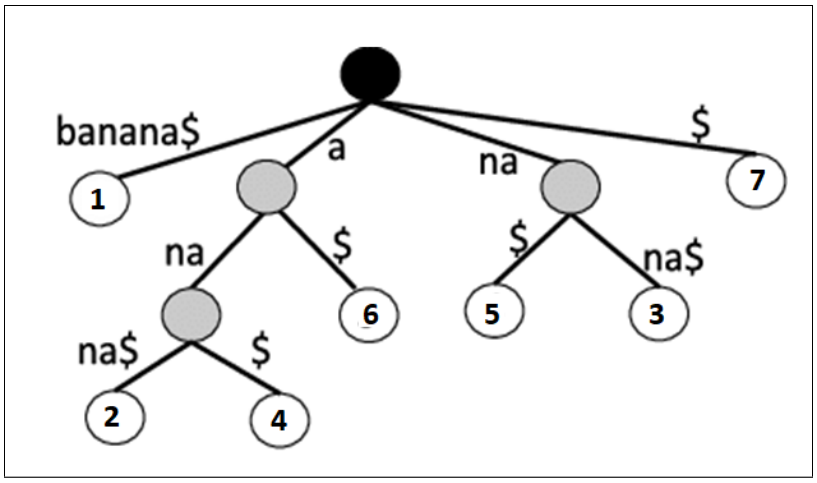


Figure 3: Process of Building Suffix Tree for string ‘banana$’

Retrieving Original String using Suffix Array and Suffix Tree: Suppose the original string is unknown but we know the length of the string from the Suffix tree as number of leaves in the Suffix tree is equal to the length of the string. The values in the Suffix array represents the starting positions of each suffix in a lexicographical sorted order. The leaves of the tree also hold the starting index of each suffix. Now for some value (n) in Suffix array, if we go to the leaf that holds the same value (n), then they will both represent the same suffix. If we subtract that value (n) from the length of the string (N), we will get p = (N-n) = (length of that suffix -1). Now, if we go up p positions starting from the nth leaf, the character at (p+1) position will be the character that exists in the nth index of the original character.

Consider the example of ‘bananna$’. The Suffix array of this string contains the value 4 in the 3rd index. Here, N = 7, n = 4, p = 3. If we now go up 3 positions starting from the leaf of the Suffix tree (Figure 3) holding the value 4, the 4th postion holds the character ‘a’ ($ ← a ← n ← **a**). Which means, the 4th index of ‘banana$’ should hold ‘a’ which is true. The peseudocode is given below:

Algorithm 2 Linear Algorithm to Retrieve Original String from Implicit Suffix Tree

```
 1: N ← Number of leaves in the Suffix Tree                         ▷ Original string length
 2: S ← NULL                                                        ▷ Unknown Original String
 3: Suff_Arr ← NULL                                    ▷ Suffix Array for the given Suffix Tree
 4: Apply DFS on Suffix Tree starting from root in lexicographic order
 5: Suff_Arr ← Output of DFS
 6: for i ← 1 to N do
 7:     p ← N - Suff_Arr[i]
 8:     Move to the leaf whose value is Suff_Arr[i]
 9:     Move up to p positions (characters) starting from the current leaf
10:     C ← Character at (p+1)th position                           ▷ Starting from the current leaf
11:     S[i] ← C
12: return S
```

Time Complexity Analysis Suppose, in the Suffix tree, n is the number of nodes and m is the number of edges. Then the complexity for applying DFS is O(n+m). If the length of the string is N, then it will take O(N) time only to retrieve N characters of the string S (as finding the character at p+1 position takes only O(1) time).
So, Total time complexity = O(n+m) + O(N) ; which is linear

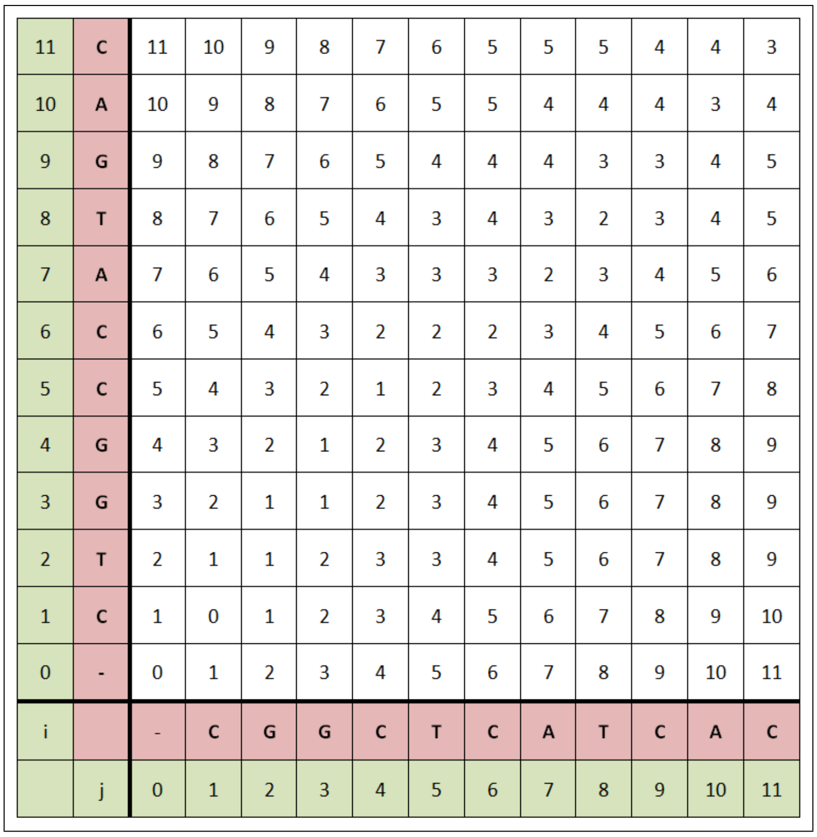
2 Solution to Question No: 2

2.1 Question I

Two strings CTGGCCATGAC and CGGCTCATCAC are given. We have to construct an alignment graph and the optimal path for computing the edit distance between these two strings. This can be done using dynamic programming. Here, cost of replacement/insertion/deletion is 1 and match is 0.

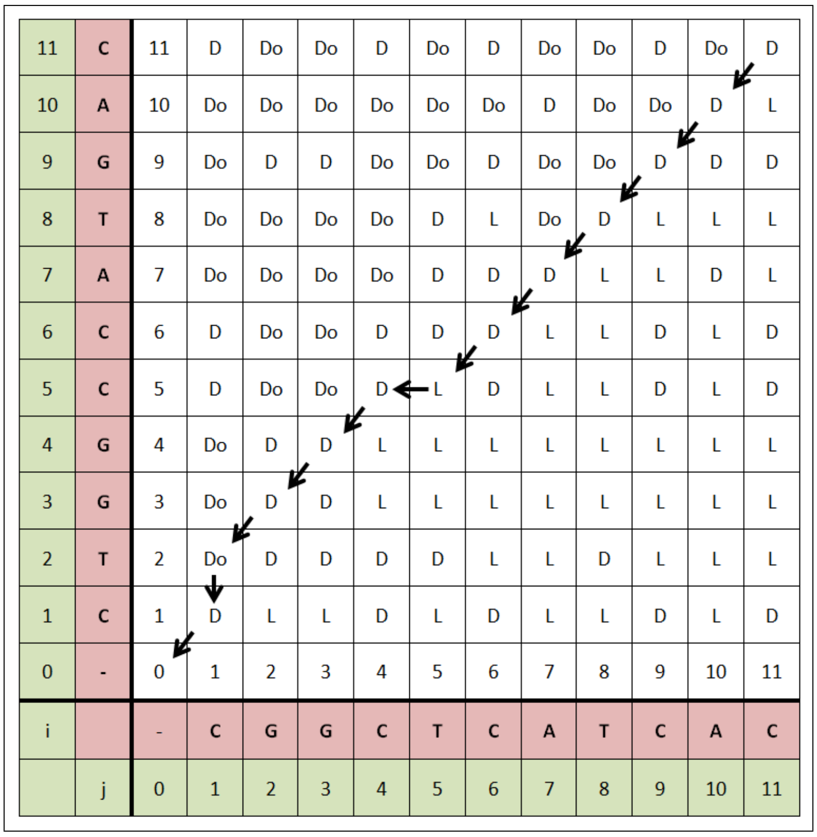
The alignment graph is shown in Figure 4. If D is the matrix, i represents no. of row and j represents no. of column then initialization condition is D[0,j]=j and D[i,0]=i . D[11,11] holds the minimum edit distance for aligning the two strings. Here, the minimum edit distance is 3.

Another matrix is shown in Figure 5. For each value, it keeps track where does the value come from (Down/Diagonal/Left). Here, Do $\rightarrow$ Down, D $\rightarrow$ Diagonal, L $\rightarrow$ Left. Multiple optimal paths can be formed from the alignment graph. Here, in Figure 5, one optimal path has been shown using the arrows. Final alignment formed following this path has been shown in Figure 6.



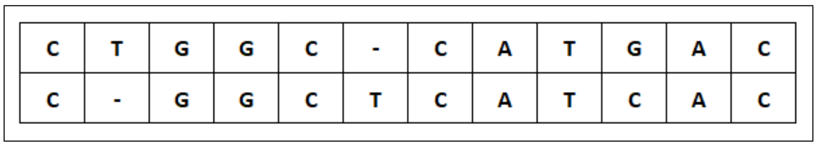
11	C	11	10	9	8	7	6	5	5	5	4	4	3
10	A	10	9	8	7	6	5	5	4	4	4	3	4
9	G	9	8	7	6	5	4	4	4	3	3	4	5
8	T	8	7	6	5	4	3	4	3	2	3	4	5
7	A	7	6	5	4	3	3	3	2	3	4	5	6
6	C	6	5	4	3	2	2	2	3	4	5	6	7
5	C	5	4	3	2	1	2	3	4	5	6	7	8
4	G	4	3	2	1	2	3	4	5	6	7	8	9
3	G	3	2	1	1	2	3	4	5	6	7	8	9
2	T	2	1	1	2	3	3	4	5	6	7	8	9
1	C	1	0	1	2	3	4	5	6	7	8	9	10
0	-	0	1	2	3	4	5	6	7	8	9	10	11
i		-	C	G	G	C	T	C	A	T	C	A	C
	j	0	1	2	3	4	5	6	7	8	9	10	11

Figure 4: Alignment graph



11	C	11	D	Do	Do	D	Do	D	Do	Do	D	Do	D
10	A	10	Do	Do	Do	Do	Do	Do	D	Do	Do	D	L
9	G	9	Do	D	D	Do	Do	D	Do	Do	D	D	D
8	T	8	Do	Do	Do	Do	D	L	Do	D	L	L	L
7	A	7	Do	Do	Do	Do	D	D	D	L	L	D	L
6	C	6	D	Do	Do	D	D	D	L	L	D	L	D
5	C	5	D	Do	Do	D	L	D	L	L	D	L	D
4	G	4	Do	D	D	L	L	L	L	L	L	L	L
3	G	3	Do	D	D	L	L	L	L	L	L	L	L
2	T	2	Do	D	D	D	D	L	L	D	L	L	L
1	C	1	D	L	L	D	L	D	L	L	D	L	D
0	-	0	1	2	3	4	5	6	7	8	9	10	11
i		-	C	G	G	C	T	C	A	T	C	A	C
	j	0	1	2	3	4	5	6	7	8	9	10	11

Figure 5: Optimal Path



C	T	G	G	C	-	C	A	T	G	A	C
C	-	G	G	C	T	C	A	T	C	A	C

Figure 6: Final Alginment

2.2 Question II

Suppose two DNA sequences S_1 and S_2 are given. They can be aligned using 'Minimum Edit Distance' algorithm. But the backtracking approach gives only one optimal path while this can actually have multiple optimal solutions. The algorithm [Algorithm 1] given below can compute the number of distinct optimal alignments using dynamic programming. Here, cost for replacement/insertion/deletion is 1 and match is 0.

Here, in Algorithm 1, at first, the alignment graph is formed using 'Minimum Edit Distance' algorithm. M and N are the lengths of S_1 and S_2 respectively. δ is used to see if it's a match or replacement. If two characters match, $\delta = 0$ (no cost), if they don't match then $\delta = 1$ (replacement). Matrix B is created to calculate the multiple optimal paths at each position of D. If i = no. of row and j = no. of column, then B[i,j] will hold the no. of possible optimal paths for the position D[i,j]. While calculating the value of D[i,j], if the minimum value comes from one direction (D[x,y]), then value of B[i,j] comes from that direction of B (B[i,j]=B[x,y]), means B[i,j] has same number of paths as B[x,y]. If the minimum value comes from multiple directions (tie), then value of B[i,j] is the summation of paths of those multiple directions. The whole procedure has been depicted clearly in Algorithm 1 and Recursive Equation Section.

Algorithm 1 Algorithm for Calculating the Number of Distinct Optimal Alignments

```
 1: M ← length($S_1$)
 2: N ← length($S_2$)
 3: $\delta$ ← 0
 4: D[i,0] ← i                                                    ▷ initialization
 5: D[0,j] ← j                                                    ▷ initialization
 6: B[0,0] ← 1                                                    ▷ initialization
 7: B[i,0] ← 1                                     ▷ initialization, 1<= $i$ <=M
 8: B[0,j] ← 1                                     ▷ initialization, 1<= $j$ <=N
 9:
10: for i ← 1 to M do
11:     for j ← 1 to N do
12:         if $S_1[i] = S_2[j]$ then
13:             $\delta$ = 0
14:         else
15:             $\delta$ = 1
16:         D[i,j] ← min ( D[i-1, j]+1 , D[i, j-1]+1, D[i-1,j-1]+$\delta$)
17:
18:         ▷ Calculating 2D array B
19:         if D[i-1, j]+1 = D[i, j-1]+1 = (D[i-1,j-1]+$\delta$) = min then
20:             B[i,j] = B[i-1, j]+ B[i, j-1]+ B[i-1,j-1]            ▷ 3 paths from D[i,j]
21:         else if (D[i-1, j]+1 = D[i, j-1]+1 = min) and (min < D[i-1,j-1]+$\delta$) then
22:             B[i,j] = B[i-1, j]+ B[i, j-1]                        ▷ 2 paths from D[i,j]
23:         else if (D[i-1, j]+1 = D[i-1,j-1]+$\delta$ = min) and (min < D[i, j-1]+1) then
24:             B[i,j] = B[i-1, j]+ B[i-1,j-1]                       ▷ 2 paths from D[i,j]
25:         else if (D[i, j-1]+1 = D[i-1,j-1]+$\delta$ = min) and (min < D[i-1, j]+1) then
26:             B[i,j] = B[i, j-1]+ B[i-1,j-1]                       ▷ 2 paths from D[i,j]
27:         else if (D[i-1, j]+1 = min) and (min < D[i, j-1]+1) and (min < D[i-1,j-1]+$\delta$) then
28:             B[i,j] = B[i-1,j-1]                                   ▷ 1 path from D[i,j]
29:         else if (D[i, j-1]+1 = min) and (min < D[i-1, j]+1) and (min < D[i-1,j-1]+$\delta$) then
30:             B[i, j] = B[i, j-1]                                   ▷ 1 path from D[i,j]
31:         else if (D[i-1,j-1]+$\delta$ = min) and (min < D[i-1, j]+1) and (min < D[i,j-1]+1) then
32:             B[i, j] = B[i-1, j-1]                                 ▷ 1 path from D[i,j]
33: return B[M,N]
```

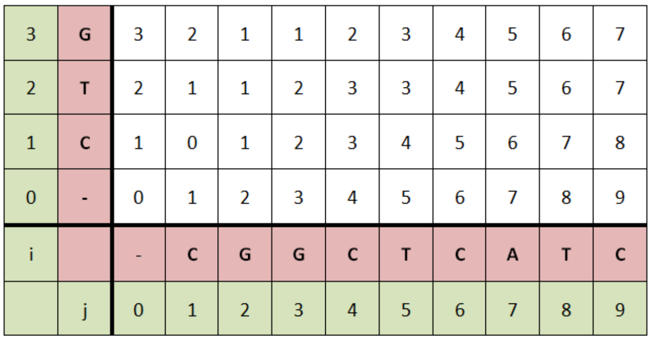
2.2.1 Recursive Equation

Suppose, the minimum value for D[i,j] coming from Diagonal or Left or Down is represented by **min**. If $S_1[i] = S_2[j]$ then $\delta = 0$, else $\delta = 1$. There is at least one possible way (1 optimal path) for aligning 2 sequences, even if their length is 0. That's why, B[0,0]=B[0,j]=B[i,0]=1.

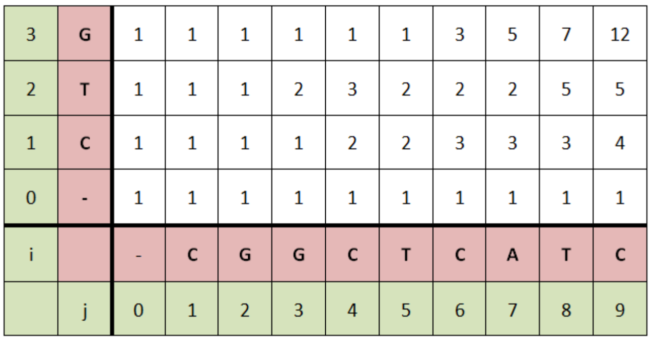
$$B[i,j] = \begin{cases} 1\,;\, if\ i = 0\ or\ j = 0 \\ B[i-1,j] + B[i,j-1] + B[i-1,j-1];\ \text{if min} = D[i-1,j] + 1 = D[i,j-1] + 1 = D[i-1,j-1] + \delta \\ B[i-1,j] + B[i,j-1];\ \text{if min} = D[i-1,j] + 1 = D[i,j-1] + 1 \\ B[i-1,j] + B[i-1,j-1];\ \text{if min} = D[i-1,j] + 1 = D[i-1,j-1] + \delta \\ B[i,j-1] + B[i-1,j-1];\ \text{if min} = D[i,j-1] + 1 = D[i-1,j-1] + \delta \\ B[i-1,j]\,;\ \text{if min} = D[i-1,j] + 1 \\ B[i,j-1]\,;\ \text{if min} = D[i,j-1] + 1 \\ B[i-1,j-1];\ \text{if min} = D[i-1,j-1] + \delta \end{cases}$$

2.2.2 Example

Suppose we have 2 string, CTG and CGGCTCATC. From 1st matrix (D) in Figure 7, we can see minimum cost is 7 and from 2nd matrix (B) in Figure 7, we can see that there are 12 possible optimal paths for aligning these two sequences.



3	G	3	2	1	1	2	3	4	5	6	7
2	T	2	1	1	2	3	3	4	5	6	7
1	C	1	0	1	2	3	4	5	6	7	8
0	-	0	1	2	3	4	5	6	7	8	9
i		-	C	G	G	C	T	C	A	T	C
	j	0	1	2	3	4	5	6	7	8	9



3	G	1	1	1	1	1	1	3	5	7	12
2	T	1	1	1	2	3	2	2	2	5	5
1	C	1	1	1	1	2	2	3	3	3	4
0	-	1	1	1	1	1	1	1	1	1	1
i		-	C	G	G	C	T	C	A	T	C
	j	0	1	2	3	4	5	6	7	8	9

Figure 7: 1st table represents D matrix and 2nd table represents B matrix

2.3 Question III

Suppose two RNA sequence s($s_1, s_2, ..., s_m$) and t($t_1, t_2, ..., t_n$) are given. We have to write an algorithm to find the maximum possible number of base-pairs between s and t such that there is no crossing. Two inter-RNA base-pairs (s_i,t_j) and (t_k,t_l)are called crossings if (1) i < k and j > l; or (2) i > k and j < l. This can be done using Dynamic Programming (DP).

To solve this problem using DP, we consider a 2 dimensional matrix D. We place one RNA sequence(s) at Row and another one(t) in column. Here, i represents row number and j represents column number. The pseudocode is given below as Algorithm 2.

Algorithm 2 Algorithm to compute the maximal number of non-crossing base pairs between two RNA sequences

```
1:  m ← length(s)
2:  n ← length(t)
3:  D[i,0] ← 0                                                        ▷ Set first row to 0
4:  D[0,j] ← 0                                                        ▷ Set first column to 0
5:
6:  for i ← 1 to m do
7:      for j ← 1 to n do
8:          if (s[i] = ‘G’ and t[j] = ‘C’) or (s[i] = ‘C’ and t[j] = ‘G’) or (s[i] = ‘A’ and t[j] = ‘U’) or (s[i] = ‘U’ and t[j] = ‘A’)
            or (s[i] = ‘G’ and t[j] = ‘U’) or (s[i] = ‘U’ and t[j] = ‘G’)  then
9:              D[i,j] ← D[i-1,j-1]+ 1)
10:         else
11:             D[i,j] ← max ( D[i-1, j] , D[i, j-1])
12: return D[m,n]
```

2.3.1 Initialization

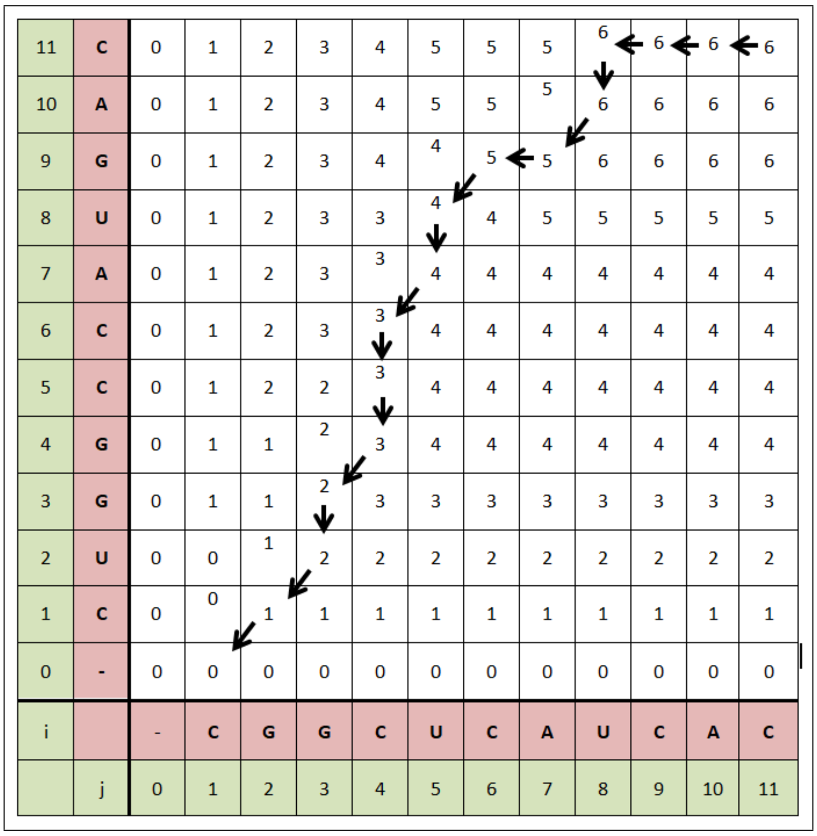
When i=0 and j=0, that means no character of s or t is considered, so number of base pairs will be 0. We set D[0,0]=0. When, i = 0 then no character of s is considered, so no base-pair is possible. We set D[0,j]=0 for any value of j. Similarly, we set D[i, 0] = 0 for any value of i.

2.3.2 Calculation

In this problem, we are considering both Watson-Crick pair (A-U, G-C) and Wobble pair (G-U). For each character of s, DP is considering the possibility of its forming a pair with all other characters of t. If s[i] and t[j] are complementary then it will take the number of maximum base-pairs formed up to s[i-1] and t[j-1] and add 1 to it. So, it will set D[i,j] = D[i-1,j-1]+1. This condition ensures that no crossing pair will be formed. If s[i] and t[j] are not complementary, then it takes the maximum value from D[i-1,j](Down) and D[i,j-1] (Left). If takes the maximum value from down then it means the path coming through down has the maximum number of base-pairs. Same goes for choosing the left option. D[m,n] will hold the maximum number of possible base-pairs between s and t where there are no crossing pairs.

2.3.3 Example

Suppose, s=CUGGCCAUGAC and t=CGGCUCAUCAC
Figure 8 shows that the maximum possible base-pairs between s and t is 6. It also shows one optimal path. But there can be multiple possible paths (each forming 6 base-pairs). According to Figure 8, the 6-pairs formed between s and t are are shown in Figure 9.



11	C	0	1	2	3	4	5	5	5	6	6	6	6
10	A	0	1	2	3	4	5	5	5	6	6	6	6
9	G	0	1	2	3	4	4	5	5	6	6	6	6
8	U	0	1	2	3	3	4	4	5	5	5	5	5
7	A	0	1	2	3	3	4	4	4	4	4	4	4
6	C	0	1	2	3	3	4	4	4	4	4	4	4
5	C	0	1	2	2	3	4	4	4	4	4	4	4
4	G	0	1	1	2	3	4	4	4	4	4	4	4
3	G	0	1	1	2	3	3	3	3	3	3	3	3
2	U	0	0	1	2	2	2	2	2	2	2	2	2
1	C	0	0	1	1	1	1	1	1	1	1	1	1
0	-	0	0	0	0	0	0	0	0	0	0	0	0
i		-	C	G	G	C	U	C	A	U	C	A	C
	j	0	1	2	3	4	5	6	7	8	9	10	11

Figure 8: Calculating maximum possible base-pairs

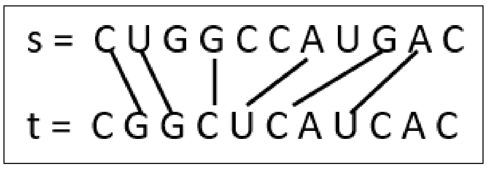


Figure 9: Six possbile base-pairs between s and t according to the calculation

2.3.4 Question IV

In 1953, Richard Bellman (applied mathematician) invented the technique of Dynamic Programming (DP).
He explained the reason behind using the term ‘Dynamic Programming’ in his autobiography, ‘Eye of the Hurricane: An Autobiograph’. During 1950’s, Charles Erwin Wilson, the Secretary of Defense, wasn’t positive about the words ‘Research’ and ‘Mathematical’. So, Richard wanted to use a term that wasn’t related to these two terms but also could express his concept of successive decision making based on subproblems. He used the word ‘Programming’. As this concept is multi-stage and time-varying, he decided to use the wrod ‘Dynamic’ along with ‘Programming’. He also really liked the word ‘Dynamic’ cause it can never be used in a negative sense. So, he decided to use the name ‘Dynamic Programming’.

3 Acknowledgment

To understand suffix array, I read an article on Suffix Array from GeeksforGeeks. To save time, I used the images of Suffix array [Figure 1] and Suffix tree [Figure 2] from internet instead of drawing it again. For Question no 2(iv), I read article from Wikipedia.