Note: Any solution to an algorithm design question MUST contain the following

four sections:

(1) Problem statement. A clear unambiguous statement of the problem to

be solved, which includes the input, the output, and the object function

with the constraints.

(2) Algorithm description. A clear, unambiguous description of the algo-

rithm.

(3) Correctness proof. A convincing mathematical argument that the algo-

rithm described solves the computational problem described.

(4) Time analysis. A time analysis of the algorithm, up to order, in terms of

all relevant parameters.

You may use any algorithms and data structures from class.

1. RNA Base Pair Maximization Problem (60 pts)

Formalize the pseudocode to predict an RNA secondary structure based on

the base pair maximization model including the trace back procedure for the pre-

dicted secondary structure. Please refer to slides 18-21 in 3.2.RNAfolding.pdf".

The secondary structure can be represented by the matching parentheses and dots

to denote paired and free bases, respectively. For example:

GAGCCAUUAGCUCAGUUGGUAGAGCAUCUGACUUUUAAUCAGAGGGUCGAAGGUUCGAGUCCUUCAUGGCUCA

(((((((..<<<<........>>>>..<<<<<.......>>>>>.....<<<<<.......>>>>>)))))).

Given an RNA sequence, s[1...n], we can use Nussinov’s base pair maximization algorithm to determine a set of base pairs in an RNA sequence such that the number of base pairs is maximal and no base pairs cross each other.

The pseudo code for maxtrix approach is below. I created it after getting intuition from the professor’s slides and also a youtube video which I will cite.

Let *input* be the input RNA string

Assume a function called isValidRNA\_Pair(), which takes two characters and checks whether they are a valid base pair (i.e. A-U, G-U, or G-C)

N ← len (*input*)

dp ← N x N matrix filled completely filled with 0s

let variable *tc* be the traceback tracker

for i from N-2 to 0:

for j from i+1 to N-1:

left ← dp[i][j-1]

down ← dp[i+1][j]

diagonalDown ← dp[i+1][j-1]

if isValidRNA\_Pair(*input*[i], *input*[j]):

dp[i][j] ← diagonalDown + 1

add coordinate (i,j) to the tracker *tc*

and set its parent to the coordinates of diagonalDown, which is (i+1,j-1)

else:

dp[i][j] ← max(left, down, diagonalDown)

add coordinate (i,j) to the tracker *tc* and set its parent to the coordinates of ether left, down, or diagonalDown depending on which one gets assign to the current i,j

Now the traceback data structure should be ready. We can verify that traceback by starting from the last index of the first row in our dp table, and then recursively move to its parent, which can only be in one of three possible positions (i.e. down, left, or diagonal down), we terminate the recursive algorithm when we reach the diagonal separator of th dp table, which is whenever i == j.

Finally for our dynamic programming method, we return the last value of the first row of dp which represents the maximum number of possible base pairs that can be formed given the input RNA string.

The time complexity of this approach is O(N2) since for every i from i to N, we check every jth value from i+1 to N-1 in the matrix. This is more optimized than a recursive approach because we are caching our work each step rather than performing repeated computation.

2. K-exons Spliced Alignment (40 pts)

One disadvantage of the exon chaining formulation of the spliced alignment algo-

rithm is that it may prefer to concatenate many short putative exons to maximize

the alignment score. Modify the spliced alignment algorithm to consider the opti-

mal alignment between a genomic sequence and an mRNA sequence with at most

k exons.

We can add a parameter called k, then decrease k for each recursive call then terminate the algorithm when k is 0. Since we are always grabbing the biggest blocks with our max function, we would obtain our alignment score from those, and we would have fewer opportunites to grab the smaller blocks in the exon since we terminated early as a result of k being 0.

Modified spliced alignment recurrence

If i is not the starting vertex of block B:

S(i, j, B, k) =

max

{

S(i – 1, j, B, k-1) – indel penalty

S(i, j – 1, B, k-1) – indel penalty

S(i – 1, j – 1, B, k-1) + δ(gi, tj)

}

If i is the starting vertex of block B:

S(i, j, B, k-1) =

max

{

S(i, j – 1, B, k-1) – indel penalty

maxall blocks B’ preceding block B S(end(B’), j, B’, k-1) – indel penalty

maxall blocks B’ preceding block B S(end(B’), j – 1, B’, k-1) + δ(gi, tj)

}

The time complexity is still O(mn2 |B|) even though we are bounded by k because we can still capture every block and add it to our alignment score if k was big enough.