Assignment: asg6

Name: satish Dodda

ULID: sdodda1

Secret Directory: behappy

**Method description:**

rna\_folding method will accept the sequence as input it will find all possible valid pairs from the sequence and the number of valid pairs in the sequence, and it will return it back to the function call

get\_pairs method is a nested function that accepts the starting I and ending j positions of a subsequence It recursively backtracks through the track table to find and return all valid base pairs within the given subsequence

**Algorithm**:

Here I used a dynamic programming approach to solve the RNA secondary structure problem

there are four roles to solve this problem

1. A must pair with U and C must pair with G
2. each base must be paired with one, and the relation must be one-to-one
3. no sharp turns (i<j-4) , pair must separated by at least 4 positions
4. structure does contain crossovers with each other

above four rule’s we need to follow to find the optimal secondary structure the rna\_folding method will accept the sequence as input, initially, we will take the table of type 2d list of size n\*n and filled it with zeros which is used to store the valid base pairs, and another 2d list of size n \*n, initialized with None values this track list is used to keep track of the positions of valid base pairs, helping to backtrack and reconstruct the actual base pairs that form the optimal secondary structure, and a dictionary named pair which contains valid pairs, which is used to check a pair is valid or not , after initializing the table and track list, the algorithm beings by iterating over subsequence’s of increasing length, starting from 5 to ensure a minimum separation of 4 bases between valid pair for each subsequence, defined by the start index i and end index j, the algorithm first assumes that the base at position j is unpaired, copying the value from table[i][j-1] this means that if no valid pair is found for j, the number of valid base pairs in the subsequence from i to j is the same as in the subsequence from i to j-1 next, it checks if the base at position j can form a valid pair with any base t in the range i to j-4, ensuring that t and j are separated by at least 4 positions If a valid pair is found based on above four rule , the algorithm calculates the score for this pairing by adding 1 or the current pair to the optimal solutions of two smaller subsequences one from i to t-1 and another from t+1 to j-1 If this score is greater than the current value in table[i][j], the algorithm updates table[i][j] with the new score and records the pairing position t in the track table this process continues for all subsequences, filling the table with the maximum number of valid pairs for each subsequence ,finally the algorithm uses the track table to backtrack and reconstruct the optimal set of base pairs this is done through the get\_pairs(i, j) function, which recursively checks each subsequence If a valid pair exists stored in track[i][j], the function adds the pair (t, j) to the result and continues searching in the subsequences before and after t and j If no pair is found, the function moves to the subsequence i to j-1, this process builds the final list of valid base pairs, which is returned along with the total number of pairs stored in table[0][n-1]

**data structures used:**

**table** is 2d list of size n\*n and filled it with zeros which is used to store the valid base pairs,

**track** 2d list of size n \*n, initialized with None values this track list is used to keep track of the positions of valid base pairs, helping to backtrack and reconstruct the actual base pairs that form the optimal secondary structure,

**pair** is a dictionary named pair which contains valid pairs, and is used to check a pair is valid or not

**Time complexity :**

**for each sequence the algorithm fill the table[i][j] which will take O(n^2)**

**after that, for each sequence from i , j the algorithm checks all potential pairs, which will take O(n)**

**T(n)=n\*T(n-1)+O(n^2)**

**T(n)=O(n^3)**

**O(n^2)\*O(n) =O(n^3)**

**Space complexity:**

In total I have used two, 2d lists of size n\*n, those are table and track, here total space required is n\*n

Total space complexity is O(n^2)

Difficulties faced:

I faced a little difficulty while writing the recurrence relations and while backtracking the pair, after studying the textbook and spending some time on the problem, I finally figured out the solution, and I solved it