Question 2

In biology, circular sequences are common, especially in bacterial genomes. These sequences can be cut at any arbitrary point, transforming them into linear sequences. The resulting sequences differ based on where the cut is made. However, since they originate from the same circular sequence, they are just cyclic permutations.

For example, if we take a circular sequence like "ATCGGTCA" and cut it at different points, we generate the following linear sequences:

- Breaking at the first "A" gives: ATCGGTCA

- Breaking at the first "C" gives: CGGTCAAT

These different linear sequences represent the same underlying circular sequence. Given two linear sequences, "X" and "Y," derived from the same circular sequence, we can use the Z-algorithm—typically used for pattern matching—to identify if one sequence is a cyclic shift of the other. The approach is simple: we concatenate one sequence with itself (forming XX) and check if the other sequence, Y, is a substring of this new string. This works because concatenating a sequence with itself results in all possible cyclic shifts of that sequence.

Applying this to the example:

Let's consider X = "ATCGGTCA" and Y = "CGGTCAAT"

1. Concatenating X with itself:

   - X' = X + X = "ATCGGTCA" + "ATCGGTCA" = "ATCGGTCAATCGGTCA"

2. Creating the new string with Y as the pattern and X' as the text and add a delimiter `"$"` to separate the pattern from the text:

   - New string: Y + "$" + X'= "CGGTCAAT$ATCGGTCAATCGGTCA"

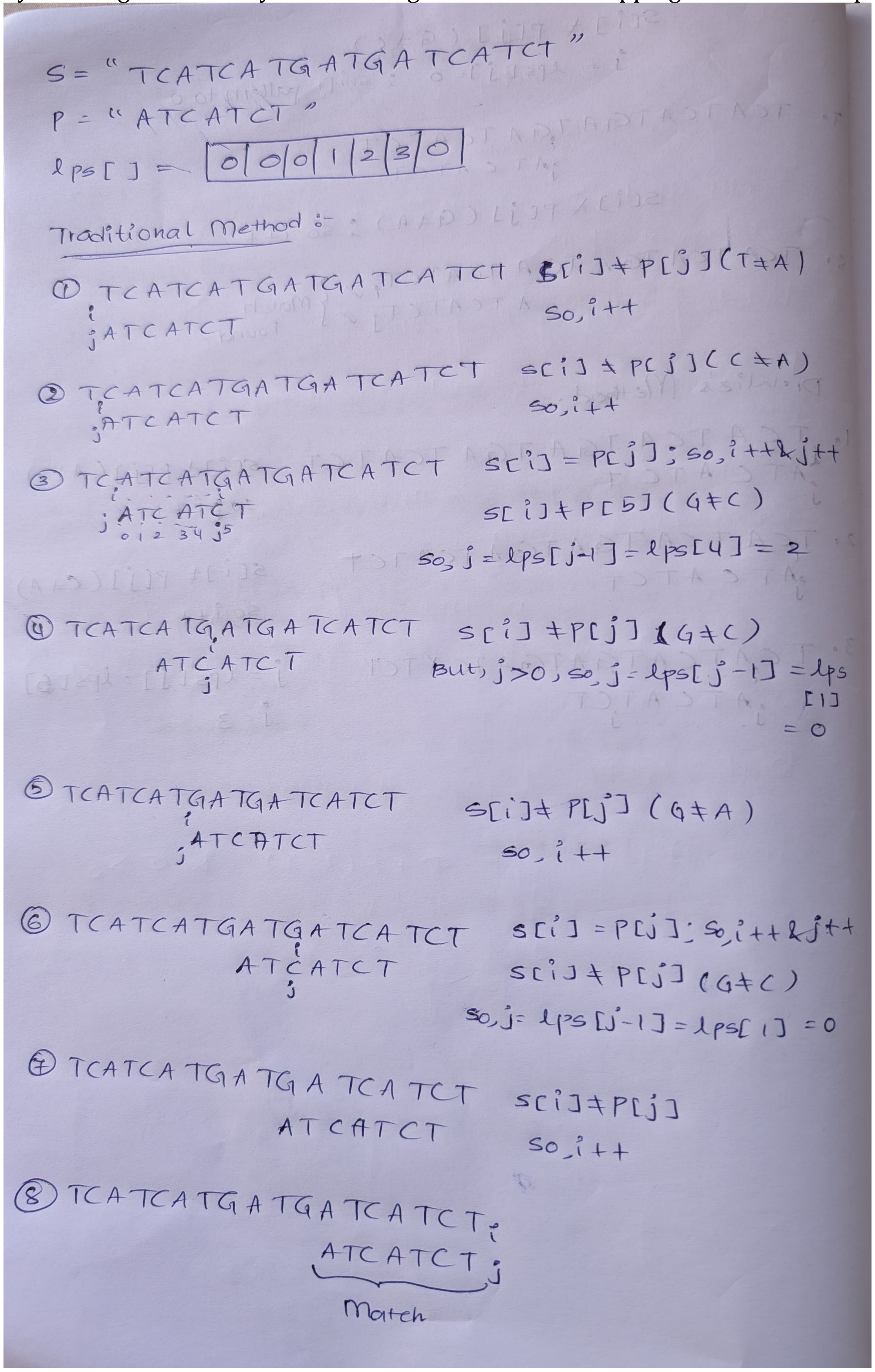
3. After applying the Z-algorithm to compute the Z-array for this new string: "CGGTCAAT$ATCGGTCAATCGGTCA". We get the following Z-array,

   Z-array - [0, 0, 0, 0, 0, 1, 2, 0, 0, 8, 0, 0, 0, 0, 0, 0, 1, 8]

From the Z-array, we find two substrings of length 8, which is the length of Y. This confirms that Y is a cyclic shift of X, meaning both sequences are derived from the same underlying circular sequence.

Question 3

The standard KMP algorithm uses the longest prefix-suffix (lps) array to optimize pattern matching by avoiding unnecessary backtracking in the text and skipping redundant comparisons.



S = "TCATCATGATGATCATCT"

P = "ATCATCT"

lps[ ] = | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

Traditional Method :-

① TCATCATGATGATCATCT $S[i] \neq P[j]$ (T≠A)
i
j ATCATCT So, i++

② TCATCATGATGATCATCT $s[i] \neq P[j]$ (C≠A)
i
j ATCATCT So, i++

③ TCATCATGATGATCATCT $s[i] = P[j]$; so, i++ & j++
i i
j ATCATCT $s[i] \neq P[5]$ (G≠C)
0 1 2 3 4 j5
So, $j = lps[j-1] = lps[4] = 2$

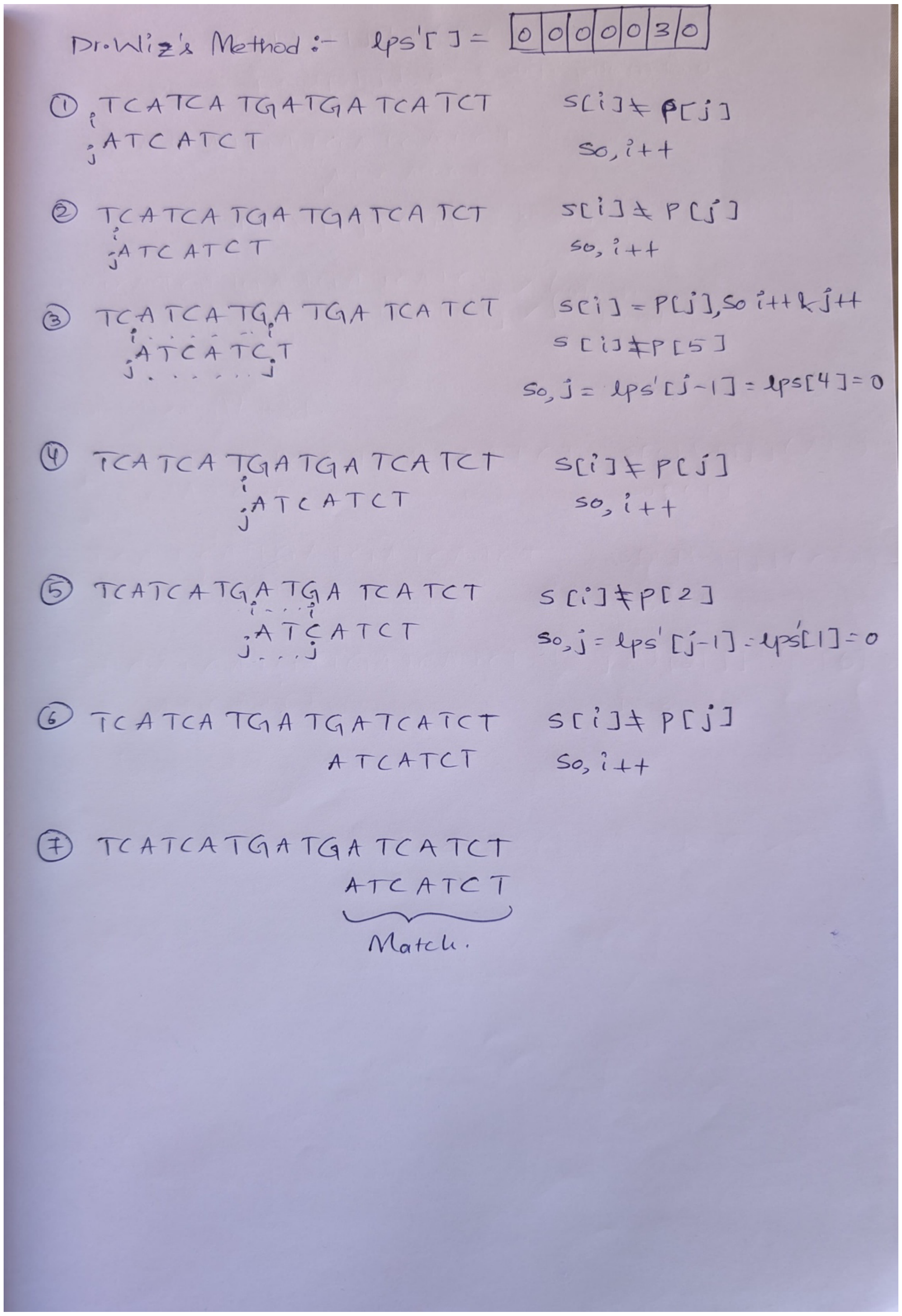
④ TCATCATGATGATCATCT $s[i] \neq P[j]$ (G≠C)
i
ATCATCT But, $j > 0$, so, $j = lps[j-1] = lps[1] = 0$
j

⑤ TCATCATGATGATCATCT $S[i] \neq P[j]$ (G≠A)
i
j ATCATCT So, i++

⑥ TCATCATGATGATCATCT $s[i] = P[j]$; So, i++ & j++
i
ATCATCT $s[i] \neq P[j]$ (G≠C)
j
So, $j = lps[j-1] = lps[1] = 0$

⑦ TCATCATGATGATCATCT $s[i] \neq P[j]$
ATCATCT So, i++

⑧ TCATCATGATGATCATCT i
$\underbrace{\text{ATCATCT}}_{\text{Match}}$ j



Dr. Wiz's Method :- lps'[ ] = $\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{3}\boxed{0}$

```
① TCATCA TGATGA TCATCT        s[i] ≠ p[j]
  i
  j ATCATCT                   so, i++

② TCATCA TGATGA TCATCT        s[i] ≠ P[j]
   i
   j ATCATCT                  so, i++

③ TCATCATGA TGA TCA TCT       s[i] = P[j], So i++ & j++
    i . . . . . . i
    ATCATCT                   s[i] ≠ P[5]
    j . . . . . j
                              So, j = lps'[j-1] = lps[4] = 0

④ TCATCA TGATGA TCATCT        s[i] ≠ P[j]
         i
         j ATCATCT            so, i++

⑤ TCATCATGA TGA TCATCT        s[i] ≠ P[2]
           i . . i
           ATCATCT            so, j = lps'[j-1] = lps'[1] = 0
           j . . j

⑥ TCATCA TGA TGATCATCT        s[i] ≠ P[j]
               ATCATCT        So, i++

⑦ TCATCATGATGA TCATCT
               ATCATCT
```

$\underbrace{\text{ATCATCT}}_{\text{Match.}}$

In this specific case, Dr. Wiz's modified algorithm appears to be more efficient than the standard KMP, as it eliminates one additional step. However, this efficiency is context dependent. It may perform well in strings where repeated characters cause frequent mismatches, but it could introduce inefficiencies in scenarios where the standard KMP would have used the lps array to skip over already matched characters. Overall, Dr. Wiz's modified algorithm may lead to more frequent restarts, making it less efficient than the standard KMP in general.

Question 4

For a given string “S” and a Z-array “Z”, we can construct an lps' array by iterating through the string in reverse. At each index, we check if the corresponding Z value is non-zero and update the lps' array by taking the maximum of the current lps' value and the Z value at that index. Essentially, if the pattern occurs in the text, the Z value at the start of the pattern will be equal to the length of the pattern.

Pseudo Code -

```
ModifiedLPSPrime(S, Z)

    length <- length of S

    lps_prime <- array of size n initialized to 0

    for i = length – 1 to 0

        if Z[i] != 0

            j <- i + Z[i] – 1

            lps_prime[j] <- max( lps_prime[j] , Z[i])

    return lps_prime
```