Bioinformatics Assignment

Syed Hussain Ahmed, NED University 16/11/2024

Transitions and Transversions - Alignment

For DNA strings s_1 and s_2 having the same length, their *transition/transversion ratio* $R(s_1, s_2)$ is the ratio of the total number of transitions to the total number of transversions, where symbol substitutions are inferred from mismatched corresponding symbols as when calculating Hamming distance (see “Counting Point Mutations”).

Given

Two DNA strings s_1 and s_2 of equal length (at most 1 kbp).

Return

The transition/transversion ratio $R(s_1, s_2)$.

Sample Dataset

```
>Rosalind_0209
GCACGCCAGAGAAACCTTATGGGAAGTGGAATTATTTCTGGTATCGTTGTAGTTATTGGA
AGTAGGGCAGTACACCCAGTT
>Rosalind_2200
TTATCTGCAAAGAAAGGCCTGACCGGTGGATATTCCGCGATCGTCTCGGTGTTTACTGGC
GTTCACGAGTTCTCTTGGTGGGT
```

Sample Output

```
1.21428571429
```

```python
from readFASTA import readFASTA

def ti_tv(seqs: list[str]) -> float:
    """Returns the Transitio/Transversion ratio of the sequences"""

    purines = ["A", "G"]
    pyrimidines = ["C", "T"]

    ti_count = 0
    tv_count = 0

    for i in range(len(seqs[1])):
        if seqs[0][i] != seqs[1][i]:
            if (seqs[0][i] in purines and seqs[1][i] in purines) or (
                seqs[0][i] in pyrimidines and seqs[1][i] in pyrimidines
            ):
```

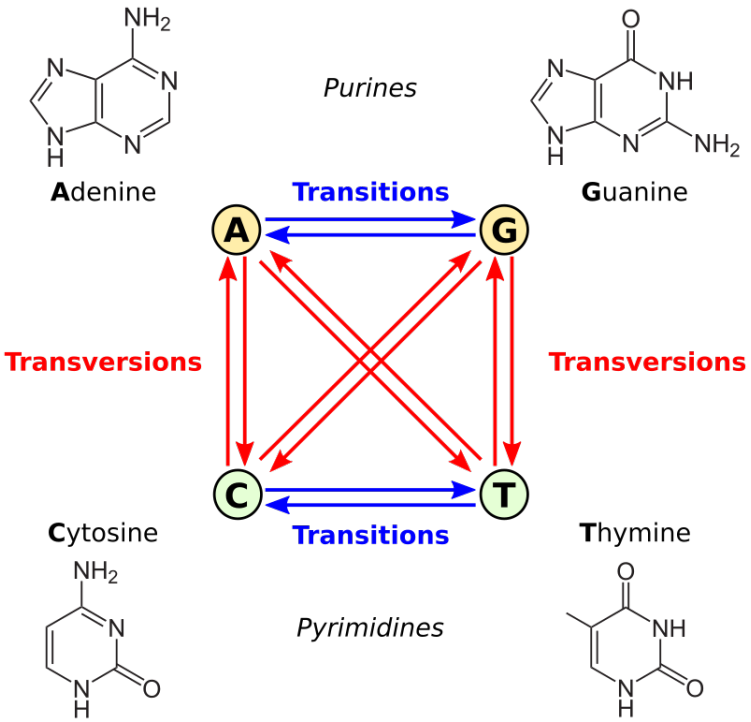
```
                ti_count += 1
            elif (seqs[0][i] in purines and seqs[1][i] in pyrimidines) or (
                seqs[0][i] in pyrimidines and seqs[1][i] in purines
            ):
                tv_count += 1
    return ti_count / tv_count

if __name__ == "__main__":
    filepath = <path_to_file>
    _, seqs = readFASTA(filepath)
    ratio = ti_tv(seqs)
    print(ratio)
```

```
1.9895833333333333
```

Explanation

The function ti_tv implements the logic to identify transitions and transversions by checking each nucleotide pair in the sequences. If both belong to the same category, it increments the transition count; otherwise, it increments the transversion count. Finally, it returns the ratio of transitions to transversions.

Biological Connection



Transistions mean when there is substitution between purine and purine or pyramidine and pyramidine. Transversions is substitution between purine and pyramidine. Normally the ti/tv ratio is approximately two but near coding regions it can get higher than 3. We can calculate the ti/tv ratio to identify the coding regions

Longest Increasing Subsequence - Dynamic Programming

A **subsequence** of a permutation is a collection of elements of the permutation in the order that they appear. For example, $(5, 3, 4)$ is a subsequence of $(5, 1, 3, 4, 2)$.

A subsequence is **increasing** if the elements of the subsequence increase, and **decreasing** if the elements decrease. For example, given the permutation $(8, 2, 1, 6, 5, 7, 4, 3, 9)$, an increasing subsequence is $(2, 6, 7, 9)$, and a decreasing subsequence is $(8, 6, 5, 4, 3)$. You may verify that these two subsequences are as long as possible.

Given: A positive integer $n \leq 10,000$ followed by a permutation π of length n.

Return: A longest increasing subsequence of π, followed by a longest decreasing subsequence of π.

Sample Dataset

```
5
5 1 4 2 3
```

Sample Output

```
1 2 3
5 4 2
```

```python
def read_file(filename):
    with open(filename) as f:
        n = int(f.readline().strip())
        perm = list(map(int, f.read().strip().split()))
        return n, perm


def print_seq(seq):
    for i in range(len(seq)):
        print(seq[i], end=" ")
    print()


def longest_increasing_subsequence(X, N):
    P = [-1] * N  # Predecessor array
    M = [-1] * (N + 1)  # Index array for LIS of each length
    L = 0  # Length of the LIS

    for i in range(N):
```

```
        # Binary search for the smallest positive l     L
        # such that X[M[l]] >= X[i]
        lo = 1
        hi = L + 1
        while lo < hi:
            mid = lo + (hi - lo) // 2
            if X[M[mid]] >= X[i]:
                hi = mid
            else:
                lo = mid + 1

        # After searching, lo == hi
        newL = lo

        # Update the predecessor of X[i]
        P[i] = M[newL - 1]
        M[newL] = i

        # If newL is longer than the current LIS length, update L
        if newL > L:
            L = newL

    # Reconstruct the LIS
    S = [0] * L
    k = M[L]
    for j in range(L - 1, -1, -1):
        S[j] = X[k]
        k = P[k]

    return S


def longest_decreasing_subsequence(X, N):
    P = [-1] * N
    M = [-1] * (N + 1)
    L = 0

    for i in range(N):
        # Binary search for the smallest positive l     L
        # such that X[M[l]] <= X[i]
        lo = 1
        hi = L + 1
        while lo < hi:
            mid = lo + (hi - lo) // 2
            if X[M[mid]] <= X[i]:
                hi = mid
            else:
```

```
                lo = mid + 1

        newL = lo
        P[i] = M[newL - 1]
        M[newL] = i
        if newL > L:
            L = newL

    S = [0] * L
    k = M[L]
    for j in range(L - 1, -1, -1):
        S[j] = X[k]
        k = P[k]

    return S


if __name__ == "__main__":
    filepath = <path_to_file>
    n, seq = read_file(filepath)

    lis = longest_increasing_subsequence(seq, n)
    print_seq(lis)

    lds = longest_decreasing_subsequence(seq, n)
    print_seq(lds)
```

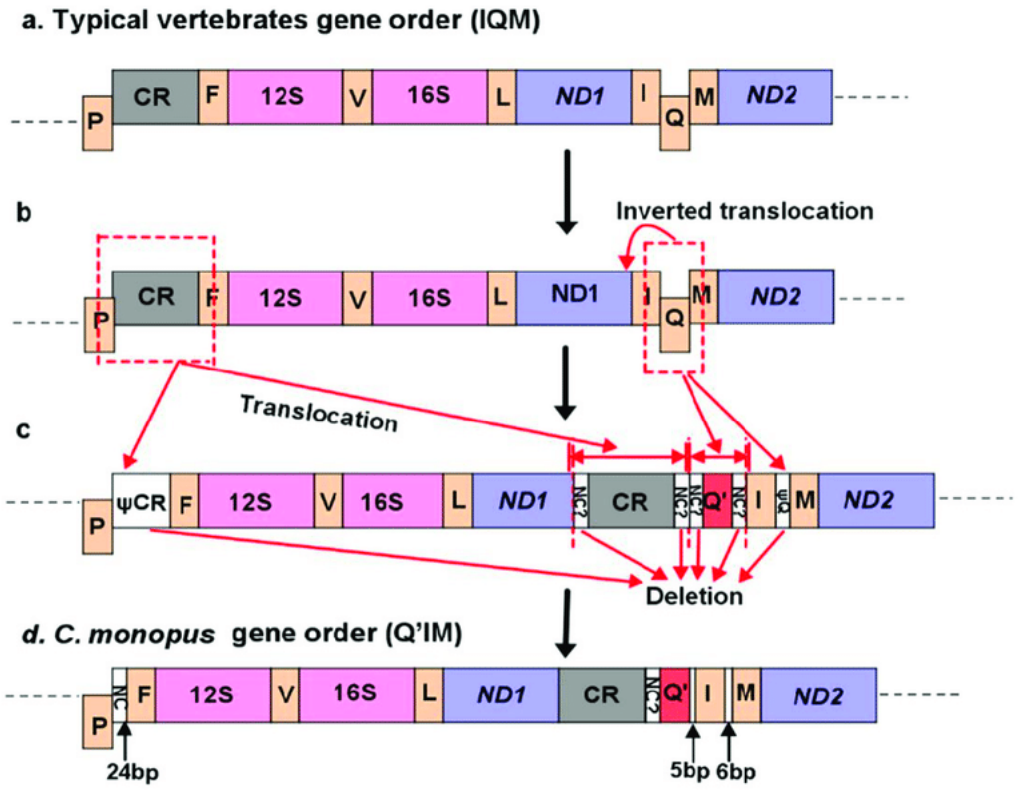
4 69 169 232 249 351 426 542 707 742 756 772 801 842 844 932 1044 1073 1087 1275 1288 1320
1343 1413 1416 1437 1473 1479 1481 1504 1608 1610 1612 1736 1769 1814 1845 1851 1887 2040
2042 2170 2199 2216 2254 2287 2380 2396 2403 2428 2470 2477 2528 2538 2546 2547 2628 2642
2689 2701 2732 2796 2829 2838 2921 2937 3166 3184 3197 3198 3221 3304 3318 3548 3616 3654
3660 3665 3745 3763 4005 4087 4119 4140 4141 4176 4254 4287 4344 4468 4485 4488 4489 4525
4562 4674 4710 4735 4801 4903 4970 5003 5008 5038 5105 5186 5228 5290 5301 5322 5355 5403
5629 5632 5642 5719 5839 5848 5873 6127 6187 6188 6238 6250 6289 6305 6323 6344 6411 6420
6436 6578 6631 6636 6805 6817 6836 6879 6929 6951 7003 7008 7012 7134 7161 7218 7224 7236
7255 7353 7358 7371 7573 7619 7755 7775 7780 7821 7854 7887 7968 8053 8176 8331 8445 8450
8465 8558 8569 8575 8639 8684 8759 8777 8813
8810 8789 8735 8732 8731 8686 8673 8648 8592 8590 8588 8486 8419 8417 8339 8307 8295 8292
8270 8236 8229 8189 8186 8183 8179 8169 8138 8130 8095 8059 7992 7980 7951 7910 7849 7791
7790 7746 7736 7685 7643 7627 7624 7604 7567 7508 7467 7419 7393 7373 7275 7190 7157 7081
6942 6924 6873 6778 6774 6669 6639 6632 6571 6556 6554 6537 6484 6475 6449 6448 6429 6409
6216 6196 6167 6153 6144 6109 6032 6015 5881 5800 5790 5778 5723 5657 5655 5645 5592 5583
5501 5488 5480 5445 5403 5276 5270 5163 5159 5118 5099 4990 4941 4938 4922 4858 4850 4803
4727 4664 4630 4627 4622 4544 4526 4517 4456 4414 4315 4274 4223 4218 4078 4048 4034 3974
3908 3805 3790 3787 3773 3760 3749 3743 3606 3600 3590 3512 3402 3315 3305 3226 3210 3070
2943 2804 2656 2560 2495 2355 2059 2037 2011 1922 1909 1877 1846 1611 1576 1569 1517 1498
1448 1377 1360 1199 1188 1137 1050 977 878 813 811 782 721 632 614 610 509 432 391 378 370
140 130

Explanation

The functions longest_increasing_subsequence and longest_decreasing_subsequence both compute subsequences efficiently using an O(n log n) algorithm described here .

The longest_increasing_subsequence function finds the Longest Increasing Subsequence (LIS) in a given sequence by maintaining a predecessor array P to track the indices of elements forming the LIS and an index array M to store the smallest end elements of LIS for various lengths. The function iterates through the sequence, using binary search to determine where the current element can extend or replace an existing LIS, updating P, M, and the LIS length (L) as needed. Finally, it reconstructs the LIS by backtracking through P. Similarly, the longest_decreasing_subsequence function finds the Longest Decreasing Subsequence (LDS) with the same structure, but the binary search logic is adjusted to ensure the subsequence elements decrease. Both functions return the reconstructed subsequences as lists, demonstrating their modularity and adaptability for different subsequence criteria.

Biological Connection



Biological sequences are prone to mutations. Despite these mutations, closely related organisms often have similar regions in their DNA. These conserved regions are called synteny blocks. Although these synteny blocks may be rearranged within the genome, they typically remain largely conserved. Due to genomic rearrangements, the order of genes within these blocks—the specific locations where a gene exists—may differ. One simple way to compare genes between two chromosomes is to search for the largest collection of genes that are found in the same order on both chromosomes. To identify the largest set of genes appearing in the same order, we need only to find the largest collection of increasing elements in the permutation.

Consensus and Profile - String Algorithms

A matrix is a rectangular table of values divided into rows and columns. An $m \times n$ matrix has m rows and n columns. Given a matrix A, we write A_{ij} to indicate the value found at the intersection of row i and column j.

Say that we have a collection of DNA strings, all having the same length n. Their profile matrix is a $4 \times n$ matrix P in which P_{ij} represents the number of times that 'A' occurs in the jth position of one of the strings, P_{ij} represents the number of times that 'C' occurs in the jth position, and so on (see below).

A *consensus string* c is a string of length n formed from our collection by taking the most common symbol at each position; the jth symbol of c therefore corresponds to the symbol having the maximum value in the jth column of the profile matrix. Of course, there may be more than one most common symbol, leading to multiple possible consensus strings.

DNA Strings

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

Profile

A	5	1	0	0	5	5	0	0
C	0	0	1	4	2	0	6	1
G	1	1	6	3	0	1	0	0
T	1	5	0	0	1	1	1	6

Consensus

A T G C A A C T

Given: A collection of at most 10 DNA strings of equal length (at most 1 kbp) in FASTA format.
Return: A consensus string and profile matrix for the collection. (If several possible consensus strings exist, then you may return any one of them.)

Sample Dataset

```
>Rosalind_1
ATCCAGCT
```

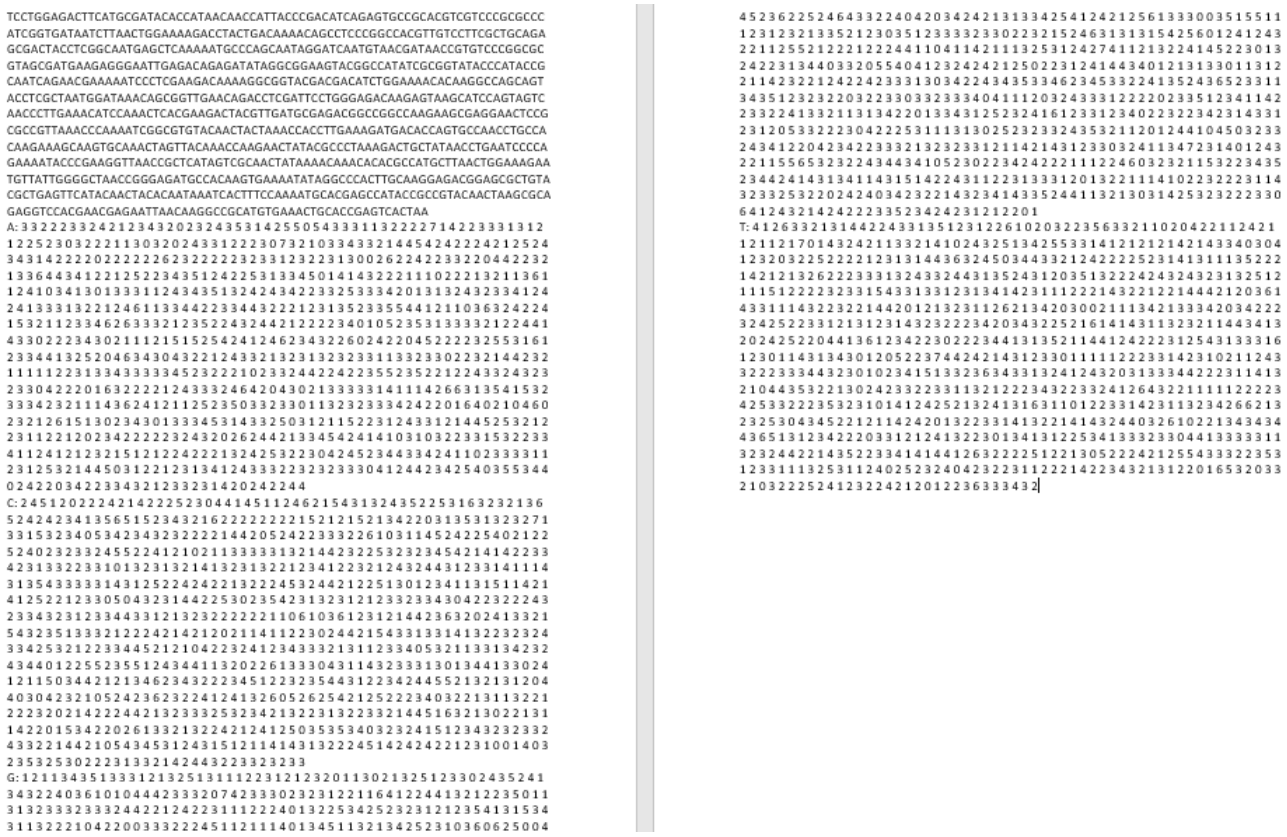
```
>Rosalind_2
GGGCAACT
>Rosalind_3
ATGGATCT
>Rosalind_4
AAGCAACC
>Rosalind_5
TTGGAACT
>Rosalind_6
ATGCCATT
>Rosalind_7
ATGGCACT
```

Sample Output

```
ATGCAACT
A: 5 1 0 0 5 5 0 0
C: 0 0 1 4 2 0 6 1
G: 1 1 6 3 0 1 0 0
T: 1 5 0 0 0 1 1 6
```

```python

from readFASTA import readFASTA

def print_matrix(matrix: list[list[int]]) -> None:
    residues = ["A", "C", "G", "T"]
    for row in range(len(matrix)):
        print(residues[row] + ": " + " ".join(map(str, matrix[row])))

def get_consensus_string(profile: list) -> str:
    con_str = ""
    residue_dict = {
        0: "A",
        1: "C",
        2: "G",
        3: "T",
    }
    for col in range(len(profile_mat[0])):
        max_value = -1
        for row in range(len(profile_mat)):
            if profile_mat[row][col] > max_value:
                max_value = profile_mat[row][col]
                residue_index = row  # 0=A; 1=C; 2=G; 3=T
```

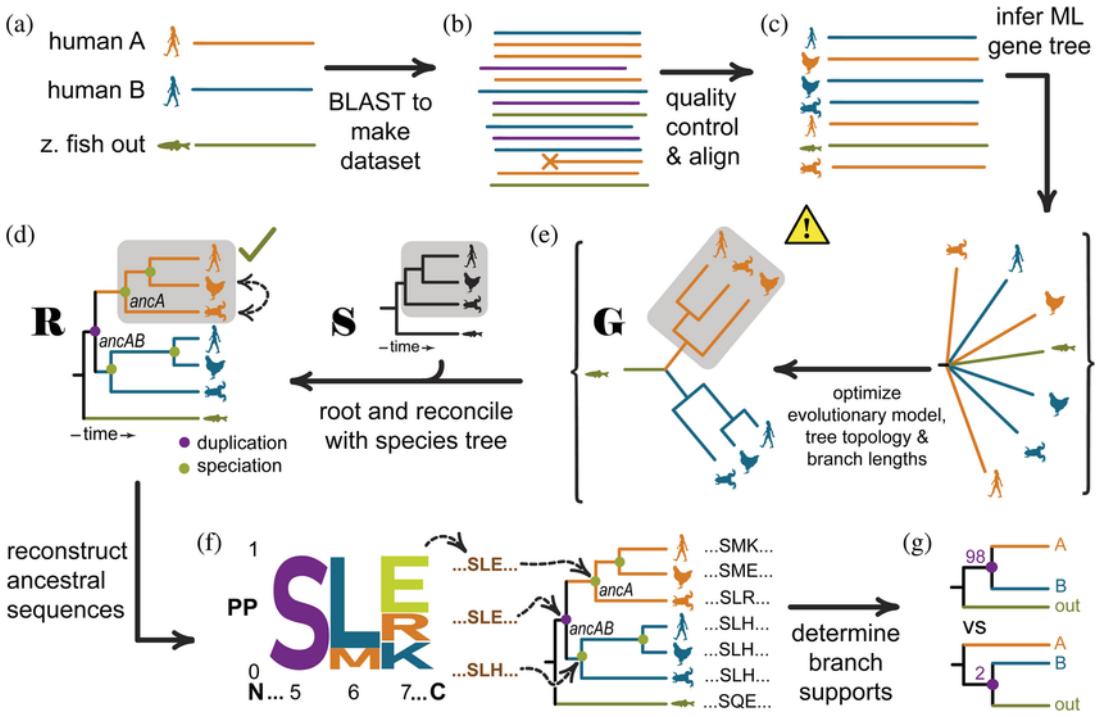
```
        con_str += residue_dict[residue_index]
    return con_str


def get_profile_matrix(seqs: list[str]) -> list:
    res_count = {
        "A": [0] * len(seqs[0]),
        "C": [0] * len(seqs[0]),
        "G": [0] * len(seqs[0]),
        "T": [0] * len(seqs[0]),
    }
    for col in range(len(seqs[0])):
        for row in range(len(seqs)):
            current_residue = seqs[row][col]
            res_count.get(current_residue)[col] += 1
    return list(res_count.values())


if __name__ == "__main__":
    filename = <path_to_file>
    _, seqs = readFASTA(filename)

    profile_mat = get_profile_matrix(seqs)
    print(get_consensus_string(profile_mat))
    print_matrix(profile_mat)
```



Explanation

The script features three key functions: get_profile_matrix, which generates a profile matrix by counting the occurrences of each nucleotide (A, C, G, T) at every position across multiple DNA sequences, storing the counts in lists grouped by nucleotide; get_consensus_string, which uses the profile matrix to construct the consensus string by identifying the most frequent nucleotide at each column and appending it to the string; and print_matrix, which formats and prints the profile matrix with row labels (A, C, G, T) for clarity, making it easy to interpret nucleotide counts at each position. These functions collectively help analyze DNA sequence alignments and identify conserved regions.

Biological Connection



With the passage of time, mutations are bound to occur, leading to the emergence of many new sequences. These sequences share a common ancestry and are said to be homologous. Given these homologous sequences, we can attempt to reconstruct the most likely ancestral sequence. However, it should be noted that the problem described above is an extreme over-simplification of the actual challenges involved.

Reference

Below is the implementation of readFASTA function. For reading the FASTA file I created a readFasta function as I did not know about the BioPython Module Bio.SeqIO.parse() method. I decided to not update it as it does the same thing.

```
def readFASTA(filepath) -> list[str]:
    header_indices = []
    seq_start_indices = []
    seqs = []
    headers = []

    with open(filepath) as rf:
        data = list(map(str.strip, rf.readlines()))

    for index in range(len(data)):
        if data[index].startswith(">"):
            header_indices.append(index)
            seq_start_indices.append(index + 1)

    for i in range(len(header_indices)):
        try:
            seq = data[seq_start_indices[i] : header_indices[i + 1]]
        except IndexError:
            seq = data[seq_start_indices[i] :]
        finally:
            seq = "".join(seq)
            seqs.append(seq)

    for header_index in header_indices:
        headers.append(data[header_index][1:])
    return headers, seqs
```