Transitions and Transversions Documentation

Introduction, and Background

When analysing a DNA sequence, it is important to locate coding regions of DNA, known as *exons*. Exons, once transcribed into an mRNA strand, are translated by ribosomes and tRNA molecules into functional proteins. *Introns* are non-coding regions of DNA, and are sandwiched between exons; meaning the DNA sequence consists of exons followed by introns, and the cycle continues. If there is already known information about a DNA sequence, such as the proteins it encodes, it is easier to identify the exons. However, given a novel DNA sequence, it is not possible to study the proteins it forms (at least not easily).

Due to the nature of how large the human genome is, it is difficult to identify where potential exons are located, and a computational method would be incredibly useful.

DNA consists of adenine, thymine, guanine, and cytosine bases. Adenine and guanine are grouped as 'purines', whereas thymine and cytosine are classified as 'pyrimidines'. Sometimes the DNA undergoes a mutation, where one nucleotide base is substituted for another; if a purine is changed for another purine, this is known as a ***transition*** mutation. Whereas if a purine is changed for a pyrimidine (or vice versa), is known as a ***transversion*** mutation.

To our luck, it was discovered that within exons, the **ratio** between **transitions** and **transversions** is much **greater** than the mean across the DNA. Transition mutations are more common within exons, because if there is a change in the nucleotide, it is less likely to change the amino acid the codon/ triplet encodes (known as a *silent mutation*). This discovery can be used to identify exons because these are regions with a greater than 'normal' ratio.

The purpose of this program, is to take two DNA sequences of equal length, and calculate the transition/transversion ratio.

Solution, and Implementation

To solve this problem once given two sequences, I could use a similar solution that I had used for my [*point mutation calculator*](https://github.com/Ilja-Lom/Point-Mutation-Calculator) whereby I manually list all possible outcomes within *if* statements to identify the nucleotide combination present. However, this solution, although simple and effective, wasn't much fun to implement and did not expand on my programming abilities. Instead, I wanted to make an algorithm which detects different nucleotide combinations within having to input each configuration.

This new solution is more complicated than using *if* statements, and is hence more computationally taxing; meaning it takes more processing power and time. The idea behind this solution was that an initial check would assess whether the nucleotide extracted from ***dna\_seq1*** (the first DNA sequence string) is a purine, or a pyrimidine; once identified, it would find the **index** of the nucleotide from ***dna\_seq1*** within the corresponding purine/ pyrimidine group. This process is repeated for ***dna\_seq2*** (the second DNA sequence), to find the **index** of the adjacent nucleotide; if the ***dna\_seq1*** nucleotide is identified as a purine, the nucleotide from ***dna\_seq2*** is searched **within the purine** variable as well. This is an important step because if ***dna\_seq2*** contains a **pyrimidine**, the nucleotide from ***dna\_seq2*** will **not** be found within the **purine** variable – returning a traceback error. To overcome this, and use this to my advantage, a set of ***try* and *except*** commands are used. In a scenario where ***dna\_seq1*** and ***dna\_seq2*** are both **purines** (because a traceback error was **not** returned), the index of the nucleotide in the **purine** group for the nucleotide from ***dna\_seq2****,* is stored. If the indexes of the **adjacent** nucleotides of ***dna\_seq1***, and ***dna\_seq2*** are the same, it means that the same nucleotides are in adjacent positions – this is **neither** a transition, nor a transversion; the loop is told to repeat the cycle for the next nucleotide within ***dna\_seq1***. However, if the indexes of the nucleotides from ***dna\_seq1***, and ***dna\_seq2*** are different (from the purine, or pyrimidine variables), it indicates that **different** nucleotides are present – this means a **transition** mutation has occurred. To explain the indexing better, the index does not come from the DNA sequence. The index comes from the position of the nucleotide of interest in either the purine variable, or the pyrimidine variable. For example, if the purine group was:

purines = "A", "G"

The index of adenine would be 0. This is the principle behind checking whether the nucleotides are the same. If both DNA sequences have a returned index of 0, it means both adjacent nucleotides are adenine; in which case you cannot have a transition, or transversion mutation.

To find **transversion** mutations between the two sequences builds up on the aforementioned traceback error. If nucleotide within ***dna\_seq2*** is **not** found within the same variable/ group (purines, pyrimidines) as ***dna\_seq1***, it **cannot** be a transition error – it must be a transversion error. In this case, when the *try* command fails due to the traceback error, the ***exception*** is executed; this section of code is simple – it checks whether the nucleotide groups from ***dna\_seq1***, and ***dna\_seq2*** are **different**. If they are, it indicates a purine has been **substituted** for a pyrimidine (or vice versa), which is a transversion.

For each cycle of the ***for*** loop along the length of ***dna\_seq1*** (doesn't matter which sequence is used, as the lengths are equal), the count for the number of transitions and transversions is stored. Finally, once the sequence has finished being processed, the ratio (R) of transitions/transversions is returned.

Limitations, and Improvements

A limitation of the approach I have used is that it requires more processing power than using simple *if* statements for all the combinations possible; the latter method is still a viable option, because there are only a few different combinations possible. The reason I did not select this approach, is because it's too easy (low hanging fruit), and I wanted to challenge myself to make an algorithm that is more universal; this is an important skill for the future when I want to work on projects with a larger set of combinations.

Due to the lower efficiency of this approach, there might be issues when analysing sequences with a million or more nucleotides. Due to the fast processing speed of modern computers, I doubt that will have a dramatic effect on the overall computation time.

Conclusion

In conclusion, the more basic program I have for solving this problem is included in this repository as '*main\_basic.py*'. Whereas the more interesting, and universal approach is titled as '*main\_advanced.py*'.