# CS 482: Computational Techniques in Biological Sequence Analysis Homework #2

Eric Haoran Huang[1]

[1]e48huang@uwaterloo.ca, 20880126, e48huang

**Abstract**

This assignment was an exercise in phylogeny, alignment-free methods and genome assembly.

## Setup

I used Python 3.10.12 on the student server machines in this run with external dependencies described in 'requirements.txt'. Run 'pip install requirements.txt' for proper setup.

## Part 1: K-mer Composition

All code and examples are found under 'q1/'. This problem was tackled in a $O(nk)$ runtime where we had the following strategy:

1. Grab the sequence using the BioPython.
2. Calculate the k-mer frequency array by iterating over all k-substrings (running $n - k + 1$ times with the length of the sequence being $n$).
   - ◇ For each k-mer, we generate all possible k-mers, e.g.: 'NA' will produce 'AA', 'CA', 'GA', 'TA'.
   - ◇ We do this by iterating through each k-mer string one nucleotide at a time and keeping all possible k-mer prefixes. We extend each prefix by the possible next nucleotide base according to IUPAC notation. This process takes $O(nk)$ times because we have $O(n)$ possible k-mers which we spend $O(k)$ time reconstructing all possible k-mers.
   - ◇ Taking the total k-mers possible, weigh each possible string equally across each k-mer. Add this weight to a k-mer frequency array whose index is defined as mapping of the possible kmer string to its lexicographic index in the $4^k$ frequency matrix. This can be easily found by setting the weights of A to 0, C to 1, G to 2, T to 3 and then turning the string (alias of base 4) to base 10.
3. Return and print out to a file as needed.

Run the code with the following line: 'python kmer_comp.py -i < input_file > -k < kmer-length >' with optional flags of '-o <output_dir>' to output to the file under '<output_dir>/<input_file_name>_len_<k-mer_length>_k-mers.txt' and '$--debug$' for more logging information.

I decided to use this method of averaging over the possible k-mers for ambiguous bases, i.e. if we have 'NA' we give 0.25 frequency weights to 'AA', 'CA', 'GA', 'TA'. This assumes that given an ambiguous base, that there is equal chance of any base that represents it to be the true base. This might not necessarily be true, but is the best way to utilize the heuristic of the ambiguous base given.

## Part 2: deBruijn Graph Construction

All code and examples are found under 'q2/'. This problem was tackled in a $O(nk \log n)$ runtime with the following strategy, where $n$ is the number of strings, $k$ is the length of the strings:

1. Calculate the reverse complements of all the strings, taking $O(nk)$ time.
2. Calculate the edge list, taking $O(nk)$ time of the joint set of the given set and the reverse complements.

3. Return the sorted list, taking $O(k)$ time in comparison and needing $O(n \log n)$ comparisons, or done in $O(nk \log n)$ time.

Run the code with the following line: 'python build_deBruijn.py -i < input_file >' with optional flags of '-o <output_dir>' to output to the file under '<output_dir>/<input_file_name>_deBruijn.txt' and '$--debug$' for more logging information.