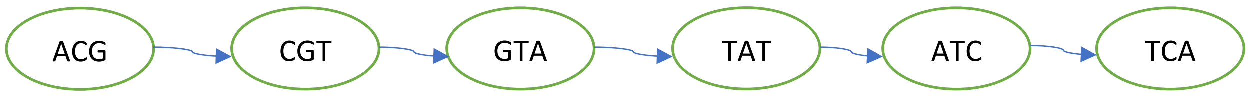
# ECE368 Programming Assignment #3

*First Submission: Monday, July 27, 2020, 11:59pm*
*Peer Review Due: Wednesday, July 29, 2020, 11:59pm*
*BRAC and Final: Friday, July 31, 2020, 11:59pm*

In bioinformatics, genome assembly refers to aligning and merging DNA fragments to recover the original DNA sequence. A DNA fragment can be represented by the bases that make it up. There are 4 different bases (characters) in DNA: A,C,G,T. A DNA fragment is sometimes referred to as a DNA string. The human genome is around 3 billion bases long (3 billion characters), but the technology that can convert your DNA to strings for the computer to work on usually only gives strings with length in the hundreds. Genome assembly is a very challenging problem with various solutions on how to align and merge DNA fragments, but for this project, you will be required to build a graph (a de Bruijn Graph) and traverse the graph to recover longer DNA sequences, which is one of the key parts in genome assembly.

You will be given an integer length *k,* along with DNA strings, also referred to as reads or sequences. From these reads you will extract substrings of length *k,* that will be saved into a vertex or node in your graph. Given an input string *ACGTATCA* and *k*=3, the nodes that you will create in your graph will be ACG, CGT, GTA, TAT, ATC, TCA. For edges in your graph, you will add an edge between consecutive vertices from an input string. In the current example, there would be edges (ACG, CGT) (CGT, GTA) (GTA, TAT) (TAT, ATC) (ATC, TCA). The corresponding graph would be as follows:



We can then traverse the graph to get our input read back, starting at the ACG node, traversing forward. Given multiple input reads, our graph may have multiple different connected components, with our graph not necessarily being completely connected.

<u>**Inputs**</u>:

argv[1] should be an integer of what our *k* length should be.

argv[2] will be the input filename that will contain the different reads. Each line in the input file corresponds to one read.

argv[3] will be the filename of where you should output the strings from your traversal

**OPTIONAL:** argv[4] - If you allow for this, it is the name of the output filename that will hold the unique nodes. If you do not implement this that is fine, it is only used for partial credit.

<u>**Outputs**</u>:

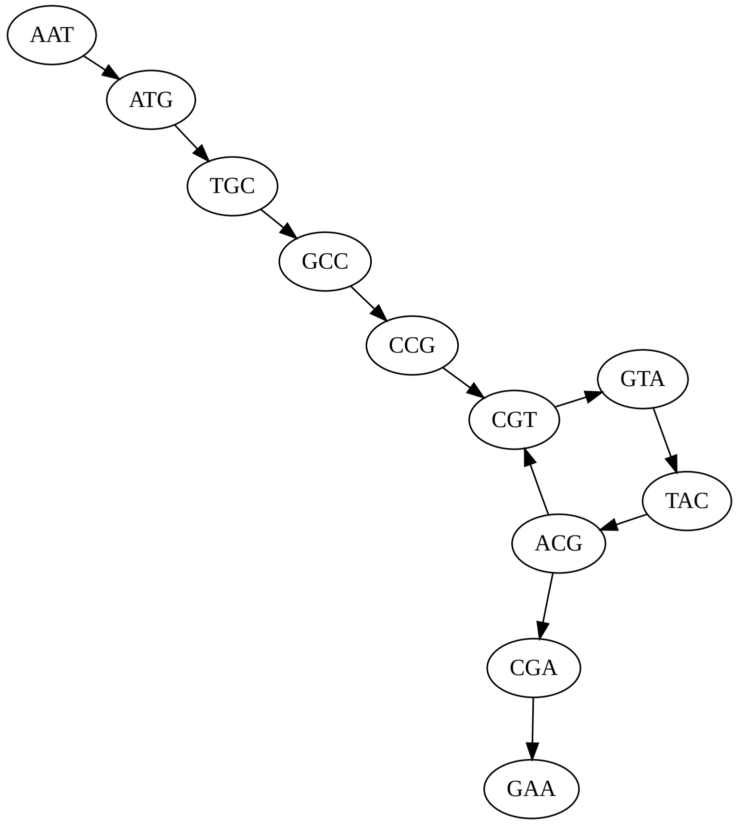
Your output file will contain strings that should be sorted by length, with the output strings 1 per line. When 2 strings have the same length, you should output the strings in alphabetical order. Your output strings should be unique. To obtain the output strings you will first build the graph, then traverse the graph when there is only 1 unique path to follow in the graph (example and further explanation below).

**OPTIONAL:** For partial credit, you may print out the unique nodes of k length in an output file given by argv[4]. You should print one node key per line, with a newline character after each key. If there are 100 nodes, you should print 100 lines.

**<u>Building the Graph:</u>**

Given input *k,* you will look at each *k* length substring in each read in the input and create vertices with these keys. Consecutive k length substrings *in the same input* read will produce an edge between the vertices with the given *k* substring keys. The edges are unique, meaning a directed edge only occurs once in the graph, there are no parallel edges. In your graph, each vertex is unique, so if the same *k* length substring occurs multiple times, there is only one vertex in the graph that corresponds to the *k* length substring. As parts of strings are repeated in different input strings, multiple edges may occur from a single vertex in your graph. Design your graph structures correspondingly. You may create your own structures or use the structures given as part of the starter code.

Example: Given the Read Set: AATGC, ATGCC, GCCGT, CGTAC, GTACG, TACGT, ACGTA, CGTAC, GTACG, TACGA, ACGAA. If you are given *k*=3, You will create the nodes: AAT, ATG, TGC, GCC, CCG, CGT, GTA, TAC, ACG, CGA, GAA in your graph. The corresponding graph would be:



__Traversing the graph__:
You will perform multiple traversals, each one starting at a hub vertex. A hub vertex is a vertex that doesn't have both InDegree and OutDegree each exactly equal to 1. At each hub vertex you will start a traversal and use each of the out edges from the hub vertex as the start of a *different* traversal. In the graph above, the hub vertices are AAT, CGT, ACG, GAA. To traverse, you will follow the edges from vertex to vertex as long as the in and out degree of the node being visited is equal to 1. If it isn't we stop the traversal. For example, the traversal starting at AAT will be: AATGCCGT. The traversal starting at CGT will be: CGTACG. The traversals starting at ACG will be: ACGT and ACGAA. There will be no traversals from GAA, as there are no out edges. The final output file would contain:

```
AATGCCGT
CGTACG
ACGAA
ACGT
```

__**Compilation:**__

```
gcc -O3 -std=c99 -Wall -Wshadow -Wvla -pedantic *.c -o pa3
```

__**Running:**__

```
./pa3 3 input.txt output.txt
```

This will read the input file, create a graph with vertices of length 3, then output the sorted traversals to output.txt.

**Optional:** If you are implementing the partial credit, you will print to argv[4] as follows:

```
./pa3 3 input.txt output.txt nodeoutput.txt
```

__**Submission and Grading:**__

This assignment requires the submission (electronically) of a zip file called pa3.zip through Brightspace. The zip file should contain any source code, including header files (if you use the provided .h files, include that as well as you are welcome to modify this file).

I do not expect you to turn in a Makefile, as I will compile using the command above. Each test case will be split evenly for the 5 points available for this project. There will be no "sub"-tests to test individual components for partial credit, other than the option above for partial credit.