**Plan:**

**To implement our simulator**, we mainly have 3 steps. Firstly, we read in the argument and parse the DNA string. Secondly, we calculate the number of reads we need to simulate and obtain those substrings from the DNA string. Here, in order to handle edge case, we will just keep the short string when the read starts towards the end of the DNA string. Finally, we introduce the errors to the reads. We use the built in random function to simulate the probability. We produce a random number, if it is bigger than the given error rate, then we do nothing; otherwise we introduce error. To introduce error, we change the particular nucleotide to the a random one of the other three nucleotides.

**To implement our assembler**, we need 4 steps. Firstly, we parse the reads file and obtain the k-mers as a list. Secondly, we form the De Bruijin graph. To form the graph, we need to first break the k-mers into left and right (k-1)-mers, and then collapse the same (k-1)-mers. To store each kmer, we introduce a node class. Each node contains the (k-1)-mer string as its inner data. Then we store the nodes in a vertex hashmap whose key is a string of (k-1)-mer and its value is a node. Before we build the graph, we first check redundancy when we want to store a (k-1)-mer. If we already have this node in our vertex hashmap, we take that node out for further use. If not, we store it in the vertex hashmap and use that specific node. We are planning to implement the edges in the graph using hashmap. If each (k-1)-mer will be a node, the key of the hashmap will be a node, while its value would be a list of nodes the key node is pointing to. The implementation is similar to implementing a graph with adjacency list. Thirdly, we will prune the graph with some simplifying techniques. For example, to collapse the paths, we could check through each key in the hashmap, and if its value only has one element, then it is collapsible. Finally, we plan to find the longest superpath. We are looking for the longest because there might be error in the de Bruijn graph, and errors are probably displayed as short branches out of the main graph.

**Implementation status of simulator: finished.**

**Implementation status of assembler:**

Our assembler could parse the reads, form the k-mers and store them in an array list. It also forms the De Bruijn graph. In our implementation, nodes of the De Bruijn graph are (k-1)-mers and the edges join a left (k-1)-mer to a right (k-1)-mer. To form the graph, it breaks the k-mers into left and right (k-1)-mers. Then, we “collapsed” the same left (k-1)-mers into one (k-1)-mer and their respective right (k-1)-mers into a list. A hash map stores the left (k-1)-mers as keys and their lists of right (k-1)-mers as values. Therefore, we could check the neighbors of every left (k-1)-mer by querying the hash map.

To help us visualize, our assembler now prints all the edges in the form of “left -> right”.

**Current problems:**

- Looking for biological evidence to explain our decisions (eg. how to deal with edge reads).

- We are working on pruning the de Bruijn graph.

- We are also thinking whether we should deal with the error when we find the superpath or when we are simplifying the graph. This is because if we could get rid of some error when we are simplifying, we might be able to simplify the graph further or better.

- How to validate our result