1. The original code uses the brute force method. It consists of two functions the overlap function and the shortestCommonSuperstring function . The time complexity of the first function (overlap function) is $O(m^2)$, where m is the maximum length of the string. The second function uses permutations functions so it has a time complexity of $O(n!)$, where n is the number of strings and then it compares the superstrings to find the shortest one , its time complexity is $O(n)$. So the time complexity will be $O(n! * n * m^2)$.

   For the optimized version the same two functions have been used by optimizing them. The overlap function iterates m times where m is the length of the string so the time complexity will be $O(m)$. The second function no longer uses permutations instead it loops over the no of strings so the time complexity can be considered as $O(n^2)$ (lower order terms are ignored). So the time complexity will be $O(n^2 * m)$. This method is basically the greedy approach.

   The main idea was to remove the permutations because for larger inputs the original method wouldn't be feasible. So using the greedy approach the number of comparisons have been reduced. Since all the permutations are not being checked the solution might not be the most efficient one but it effectively reduces the time complexity for large inputs.

2. 2.1 Using the given dataset generator for the UTA ID: 1002142811 a unique dataset of reads has been generated and stored in the text file "1002142811.txt".

   2.2 Considering k = 2, and reading in the "1002142811.txt" which contains the generated set of reads, a file named "kmers.txt" has been created which contains unique kmers of the reads.

   2.3 For plotting the De Bruijn Graph the "kmers.txt" and "1002142811.txt" files have been read into the code and then networkx library(used for directed graphs) and matplotlib has been used.

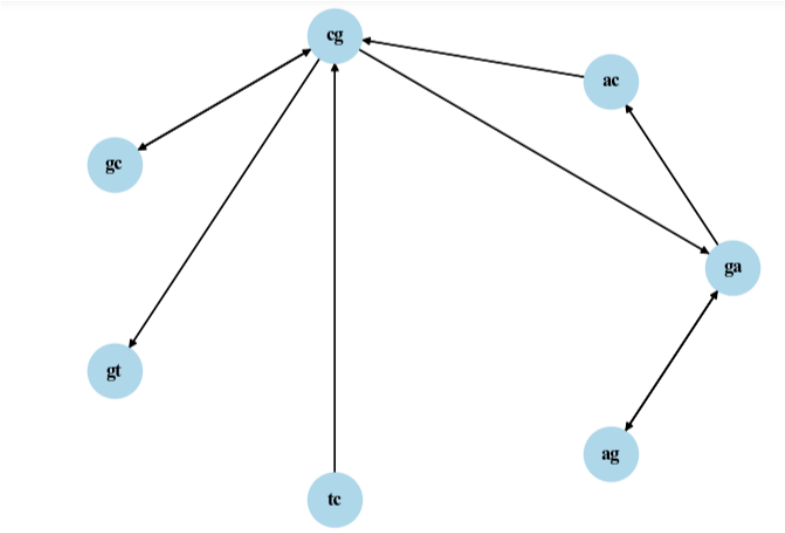

Figure 1: De Bruijn Graph

   2.4 There's a Eulerian walk in the generated debruijn graph.

For node cg: in-degree=3, out-degree=3 (balanced)

For node ac: in-degree=1, out-degree=1 (balanced)

For node ga: in-degree=2, out-degree=2 (balanced)

For node gc: in-degree=1, out-degree=1 (balanced)

For node ag: in-degree=1, out-degree=1 (balanced)

For node tc: in-degree=0, out-degree=1 (semi-balanced)

For node gt: in-degree=1, out-degree=0 (semi-balanced)

A debruijn graph is said have a Eulerian walk if it is directed, connected and if and only if it has at most 2 semi-balanced nodes and all other nodes balanced. Hence the generated debruijn graph can be called as Eulerian.

2.5 Since the generated Debruijn graph has a Eulerian walk , the walk gives a genome sequence.

```
tc->cg->ga->ag->ga->ac->cg->gc->cg->gt
```

The assembled genome sequence is

"tcgagacgcgt"

3. Difficulty adjustment:

- It took me almost 5 days to work on this assignment. (I worked for about 3-5 hours each day).
- The first question confused me a bit. I used greedy approach to optimize the code (explained in one of the lectures) and I tried running the code by giving some strings as input but the original version and my optimized version produced different outputs because of less number of comparisons.
- For the second question I had to refer to the lecture notes as well as the internet (very few articles were available) to learn about the concept of debruijn graph, eulerian path and eulerian circle .