Course Name: COMP20131 Data Analytics Tech & Prog

Assignment Title: Data Analytics Techniques and Programming

Student Name: Beemnet Amdissa Teshome

Student ID: T0338757

Date of submission: 28/03/2025

This coursework is designed to cover Module Learning Outcomes 2, 3, 4, 5 and 6 of the module specification. Max words: 5,000 with template included.

Problem 1: Applications of iterative and bisection search

There is no written report element for this problem except for attribution of sources you used. Correctness for this problem is demonstrated by code only.

Attribution: None
(if you didn't use any sources, write *none* next to Attribution.)

Problem 2: Modelling knowledge with Python classes – designing a Fraction class

Your code will be checked for correctness and good practice regarding design and comments, for example, if it shows adherence to PEP 8.

Present in this report:

Objective: *Describe the class design and how it fits into the Python data model.*

The Fraction class will initialise an object with a numerator and a denominator. It is designed to model rational numbers. The __init__ function will check if the denominator is zero to raise a value error. The class integrates with python data model by implementing several dunder methods. These methods allow objects to interact seamlessly with Python's built-in functions and operators, letting users write expressions easily using math signs (-, +, *, /).

Methodology: *Explain key OOP principles used (abstraction, decomposition)*

The Fraction class follows key Object-Oriented Principles. The class encapsulates objects by hiding the numerator and denominator. The user will interact with the objects as a string because the dunder method __str__ is used to display the fraction. Abstraction is implemented by generalising details and focusing on functionality rather

than objects. More complex functions are broken down within the methods, thereby applying the principle of decomposition.

Implementation: *Discuss coding structure*

What data attributes did you create and why? Numerator and denominator attributes are created to store values to make the fraction object. The used in the dunder and other methods to do calculations.

What methods did you create and why? I have created the following dunder methods to add, subtract, multiply, divide, convert to float, to inverse and fraction and reduce to lowest form. I have created the dunder methods to simplify common mathematical operations. I have created the reduce method to put a fraction in its lowest form.

What were the design principles behind the following methods?

Part A:__init__, __str__, __add__, __sub__, __float___

__init__: This method is used to create an object of Fraction class. Within the method, it checks if the denominator is equal to zero and raises a value error. It displays a message "Denominator cannot be zero" to inform user about the error.

__str__: This method is used to display the fraction object to the user as a string. The numerator and denominator of the fraction object are assigned to a variable and the method returns a formatted string with a "/" symbol to separate the numerator and denominator.

__add__: This dunder method describes the action of the addition operator (+). It handles the addition of two fraction objects. The sum is returned as a new Fraction object after the addition is completed. The user can simply use the (+) sign to add two fraction objects because a dunder method is implemented.

__sub__: This dunder method describes the action of the subtraction operator (-). It allows the subtraction of two fraction objects. The result of the subtraction is returned as a new Fraction object. Just like addition, the (-) sign is used to perform the operation.

__float__: This method returns the fraction object as a float value. When the method is called, the numerator is divided by the denominator and the value is displayed.

Part B: What were the design principles behind any further methods you chose to design?

__mul__: this dunder method allows user to multiply two fraction objects using the asterisk (*) sign. Method returns a new fraction object of the result.

__truediv__: this dunder method allows user to divide two fraction objects using the forward slash (/) sign. The method will check if the numerator of the second fraction

object is zero and raise a zero division error if that is the case. The result is returned as a new fraction object.

__eq__: this method is used to check the equality of two fraction objects. It cross multiplies the numerator and denominator and returns a Boolean value of the result.

Inverse: this method is used to inverse a fraction object. It will return the fraction object as a string and simply swaps the position of the numerator and denominator.

Part C: What were the design principles behind your reduce() method?

The reduce method will output the lowest form of the fraction object. It used a gcd() method imported from math library to calculate a common divisor between the numerator and the denominator. Finally, the method returns a fraction object by floor division of the numerator and denominator by the common divisor.

Results: *briefly show evidence of functionality and correctness.*

Attribution: none
(if you didn't use any sources, write *none* next to Attribution.)

Problem 3: Understanding program design and efficiency - runtime comparison of sorting algorithms

Objective: *Compare the different sorting algorithms based on runtime with respect to the number of operations.*

Methodology: *explain the sorting algorithms being compared – Bubble Sort, Selection Sort, Merge Sort – and the method of comparison:*

Bubble sort: this sorting algorithms will step through the whole list and compare adjacent elements. Then swaps them if they are in the wrong order. For instance, it will compare the first two elements in the list. If the first is greater than the second, swap the elements. Then move to the next pair and do the same thing. It repeats the process until all the elements are in the proper position.

Selection sort: this sorting algorithm will repeatedly select either the smallest or largest element and move it from the unsorted portion to the sorted portion.

Merge sort : this sorting algorithm will break down the list into smaller lists and merge them back after sorting the smaller lists.

The method used to compare the three sorting algorithm is by counting the number of operation performed while sorting the list. By adding up the number of operations in the sorting algorithms, the results can describe the efficiency of the algorithms.

Implementation: *describe how operations are counted and how lists to sort are generated.*

The lists-to-sort is created by taking an argument for the list size and then generating random numbers between 1 and 100 for the list size. If the list size is 6, a list with 6 randomly generated numbers will be created. Then the list will be passed as an argument to the sorting algorithms.

The operations are counted using global variables for each sorting algorithms. The variables are initialised to zero. Then every time the algorithm compares elements of the list, the variable is increased by 1. After the list is sorted, the variables are printed out to compare the number of operations for each algorithm.

Results: *Present insights gained through comparing the result of Part A with Part B.*

Part A: Describe outcome of experiments on random lists

For bubble sort, elements are repeatedly swapped until they are in a sorted position. Therefore, bubble sort is inefficient if the size of the list if large. It has a $O(n^2)$ complexity so the number of comparisons grow exponentially as the list gets larger.

For selection sort, similar to bubble sort, involves moving elements from unsorted portion to a sorted position. Therefore, it has a $O(n^2)$ complexity and is inefficient for larger lists.

For merge sort, has an efficient complexity of O(n log n) because of the splitting into smaller lists and merging the sorted lists.

Part B: Describe outcome of experiments on random lists after they have already been sorted.

In bubble sort algorithm, it goes through the list only once if the list is sorted. Therefore, the complexity is O(n) – linear. This happens because the algorithm recognizes that elements are in the correct position and does not swap elements.

In selection sort, the algorithm does not recognize that the list is sorted therefore proceeds as if the list is unsorted. The complexity stays as normal $O(n^2)$.

In merge sort, the complexity stays the same and the algorithm does not skip the splitting and merging process.

Is there a change in performance between sorted and unsorted lists?

The biggest change in performance happens with bubble sort. The complexity shrinks from exponential to just linear. There is no significant change in performance of selection and merge sort.

Part C: What are the best- and worst-case runtimes of the three algorithms? Which of the three sorting algorithms are stable sorts and why?

The best and worst case runtimes for selection and merge sort does not change. Selection sort is $O(n^2)$. Merge sort is O(n log n). For bubble sort the best case runtime complexity is O(n), when the list is already sorted. The worst case is $O(n^2)$.

The stable sort of the three sorting algorithms is bubble sort and merge sort. Both algorithms do not disrupt the order of equal elements. Selection sort could disrupt the order of equal elements when selection smallest element therefore is not stable.

Attribution:
(if you didn't use any sources, write *none* next to Attribution.)

GeeksforGeeks (2016). *Time Complexities of all Sorting Algorithms*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/ [Accessed 25 Mar. 2025].

Problem 4: Optimization Problems - Dynamic Programming

Objective: *Implement dynamic programming solutions to solve the given problems efficiently and correctly and justify the correctness of the solution.*

The objective of the program is to find the lowest number of coins needed to give the correct change. The program uses dynamic programming with tabular method to solve the problem. The function takes a list for value of coins and the change.

Methodology: *Describe the ideas upon which dynamic programming is based (optimal substructure, overlapping subproblems).*

Dynamic programming is an algorithm design that is used to efficiently solve problems by dividing them into smaller subproblems. It capitalizes on optimal substructure and overlapping subproblems.

If an optimal solution for the problem can be constructed from an optimal solution from a subproblem, then that problem is said to have an optimal substructure. It can typically be broken down recursively.

If the same subproblems are solved multiple times while solving the main problem, then the problem exhibits overlapping subproblems. Dynamic programming shines in scenarios where subproblems recur throughout the process of solving the global problem.

Implementation: *Explain how the ideas of dynamic programming apply to each of the two given problems and how you implement them.*

Coin change problem

When solving the coin change problem, first a table is initialised and will be filled with the least number of coins needed for that index. The base case will be that first element with value of 0. Table is used to store the solutions for the subproblems to avoid redundancy. Therefore the optimal substructures stored in the table will be used to solve for the change in the function argument.

Word break problem

When solving the word break problem, first a table is initialised with the length of the string segment. At first all values except the index zero are set to false. The first true after index zero will mean the string slice starting from 0 to the true position is a word from the dictionary list. The previously computed values are stored in the initialised list/ table. The solved subproblems are used to build up and solve the main problem.

Results: *discuss the correctness and computational efficiency of your solutions (each separately).*

The coin change problem has been tasted and provides a correct solution. The for loop iterates through 1 up to value of change and the inner for loop iterates through the list of coin values. Therefore the runtime complexity of the coin change program is O(n * m) where n is the change and m is the coin values.

The word break problem will output true or false depending on whether the string can be segmented or not. In addition to returning a Boolean value, it also prints out the segmented string by saving the index in a separate list and backtracking through the dynamic programming table to display the segmented strings. The big O of the word break program is $O(n^2)$ due to nested for loops iterating twice through the length of the segment.

Problem 5: Problem modelling and analysis through simulation with Numpy – The Knight's Tour Problem

Objective: *use randomness to automatically generate candidate knight's tours to study the structure of the Knight's Tour problem. Investigate a heuristic approach to the problem. Visualize results effectively.*

The basic knight tour program starts at row 4 column 3 and randomly chooses from a given set of legal moves. It stops when no unvisited legal moves are available. The board is initialised as a NumPy array with rows and columns with values of zero. The brute force approach will repeatedly perform the basic knight tour, and the results are displayed for analysis in the form of a histogram. Heuristic approach will make the knight visit as many unvisited positions as possible. This approach will give the knight best chance to visit all positions in the board.

Methodology: *explain the approach you used for Parts A-C.*

The algorithm for a basic knight tour will choose a random move from a list of possible moves. The list of moves is created by choosing rows and columns from a list of legal moves for a knight. Conditional statement is used to check for unvisited and inbound positions.

The brute force approach is similar to basic knight tour except it will run repeatedly. The number of moves per tour are saved in a list. I have used a histogram to visualize the data and analysis the result from the brute force approach and compare it to heuristic approach.

The heuristic approach starts by creating the reachability table, so the knight starts by choosing the least reachable position first. It moves through the board to visit the most moves possible. The moves are stores in a list and visualised using a histogram.

Implementation: *describe the algorithms used in Parts A-C.*

Part A: basic Knight's tour

At first, I initialised a chess board using 8 * 8Numpy array. The legal moves are divided into horizontal and vertical moves. In a while loop, possible moves are added into a list by checking the position is visited and within the chess board boundaries. From the list of possible moves, the program chooses a random move. It stops when there's no more valid moves left. The output is displayed in the console window.

Part B: simulating random tours 1,000,000 times.

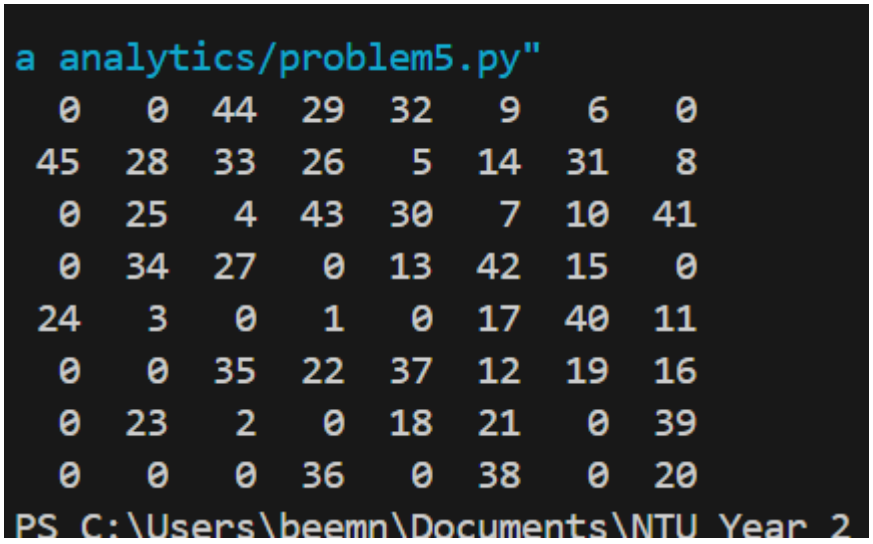
The algorithm for simulating knight tour 1,000,000 times is similar to the random knight tour. The algorithm will iterate 1,000,000 times in a for loop, creating the chess board using Numpy array, listing possible moves by checking if the move is unvisited and within the chess board boundary. The result is stored in a list and visualised using a histogram.

Part C: using the heuristic to search for a solution.

The heuristic approach will start by initializing a reachability table for the knight to start by choosing the least reachability first. The valid moves are stored in a list. A function is used for encapsulation purposes. The reachability is updated using a function. The reachability is reduced by one every time the knight makes a move. Within the for loop that moves the knight, the minimum reachability is selected from the valid moves. The move with the least reachability is selected. If there is more than one, program randomly selects from the list of moves with least reachability. This is repeated until no more moves available. The process repeated multiple times like the brute force approach and program creates a histogram to compare results to brute force method.

Results: *Give an informative visualization of a basic Knight's Tour. Describe the results of the random simulations in Part B. Did you successfully find a tour? How many? Give an informative visualization of this experiment. Describe the heuristic approach in Part C. Did you successfully find a tour? How many? Giva an informative visualization of this experiment. Finally, compare and interpret the outcome of your experiments.*

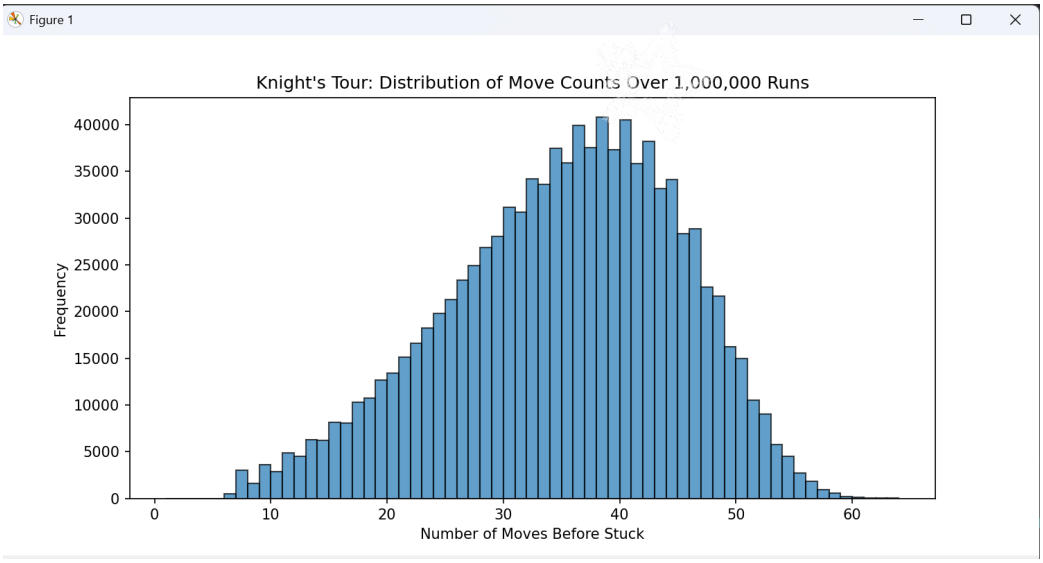
The basic knight tour runs once. The output is designed to display the chess board in the console.



the numbers represent the order of movement of knight. The zeroes represent the unvisited positions. The 1 is the starting position.

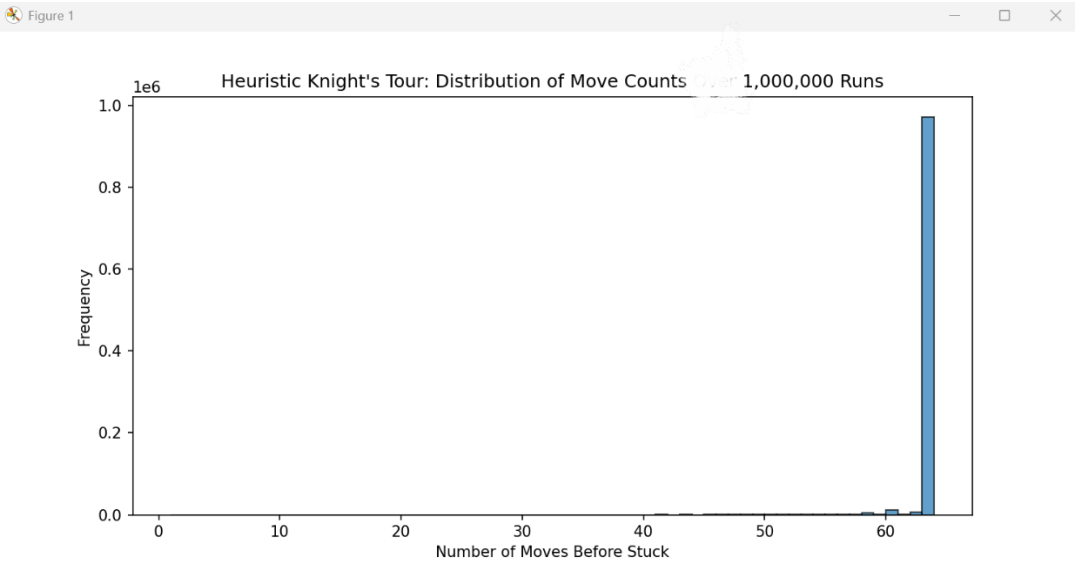
The brute force approach will run the basic knight tour repeatedly for 1,000,000 times. The above chess board is not displayed because it is not relevant. Instead the results

are stored in a list and visualised using a histogram.



The histogram follows a roughly normal distribution. This suggests that most tours end within a predictable range. Highest frequency of tours occur between 35-45 moves before getting stuck. Frequency drops sharply after 50 moves, so very few tours reach 60 moves. This indicates a fully complete tour is very unlikely. This histogram shape indicates that random movement can result in a predictable pattern of tour lengths.

The heuristic approach also runs 1,000,000 times like the brute force approach. The histogram will look significantly different from the brute force due to the knight choosing the least reachable moves first.



This histogram indicates that nearly all tours making 64 moves. This is very different to the random brute force approach. In rare cases knight gets stuck before completing the 64 moves. But the number is almost negligible compared toe the number of completed tours. This suggests the heuristic approach is very effective compared to brute force to get a completed knight tour.

Attribution:
(if you didn't use any sources, write *none* next to Attribution.)

Problem 6: Developing a tool for text analysis using Python, Graph Algorithms, Pandas and Project Gutenberg.

Objective: *Use various techniques to explore texts form the Project Gutenberg e-text corpus. Show evidence of data analytical skills of preparing, modelling, analysing and displaying results to the user.*

The objective of the text analyser program is to present several analysis tools for the user in the form of a menu. User can choose from eight options which perform different analysis techniques.

Methodology: *describe the techniques used for functions implemented.*

There are 8 different options to choose from.

1) Number of distinct words: this option counts the number of distinct words in the text. The document is first prepared by discarding punctuation and making every word lower case. That so words like "The" and "the" are not counted as different words. The cleaned text is split into words and passed into a pandas series. Value_counts() method is used to count the distinct values within a pandas series.
2) Most frequent word in the text: the most frequent word in the text is found my applying the idxmax() method to the pandas series and printing the result with the frequency.
3) Word with the largest number of unique neighbours: this is implemented by first setting up a dataframe with frequency and unique neighbours as columns. Frequency column is filled with the panda series. A dictionary is used to put the words side by side. The unique neighbour column is filled with values from the dictionary. Largest number of unique neighbours is found by applying idxmax() method to find word with largest unique neighbour and loc[] methods to locate the word from the dataframe.
4) Word with the least number of unique neighbour: this is implemented the same way as number 3. Used idxmin() to find the smallest number of unique neighbours and loc[] to locate the word from he datafarme.
5) Other descriptive statistics: pandas has a built-in function to display basic statistics of a dataframe. The function is df.describe().
6) Find shortest path between two words: the user will enter 2 words to find shortest path between. The function to find shortest path will check if the two

words exist in the text. Then a deque is initialised to save the path between the two words. If no path exists, function return a message to inform user.

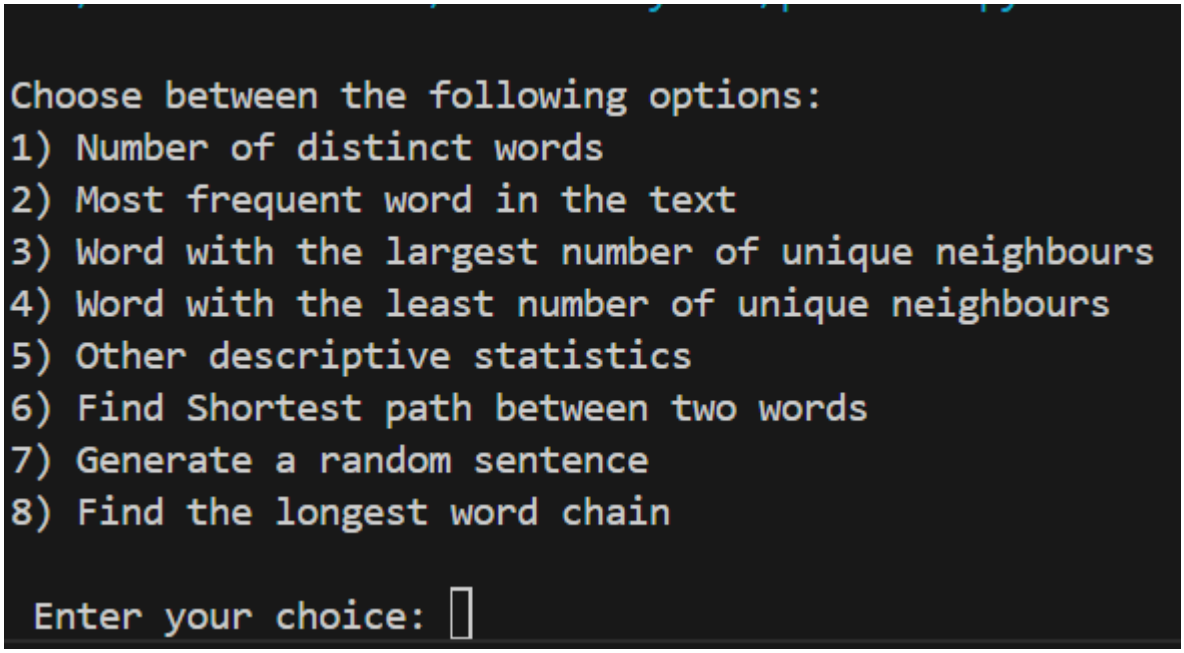
7) Generate a random sentence: the user is asked to input a word, else function will choose a random word. Maximum length is set to 20 words. A for loop will iterate until 20 and add words starting from the input word. The words come from the dictionary from earlier and added into a list containing the input word at the beginning. At the end words are joined together using join() function to make the sentence.
8) Find the longest word chain: this option will find the longest word chain where no word is repeated. First the function will find the longest path using a recursive function. The longest path is used to find the words from the dictionary and finally returned.

Implementation: *describe coding approach – models, data structures and additional libraries used.*

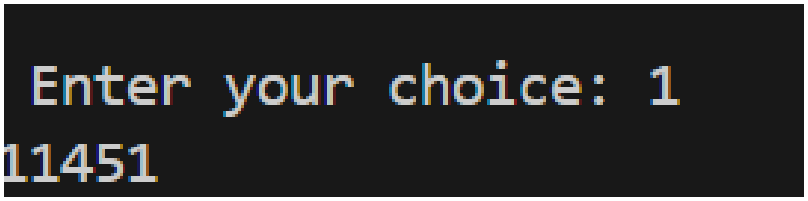
Functions are created for the specific options in the menu to work on each option in isolation. The menu displays the options numbered from 1 to 8. There is no loop to iterate through menu choices. When one option is run, the user has to run the program again to view another option. Data structures used are dictionaries, pandas dataframes, pandas series, and lists. Dictionary is used to store the words next to each other. Pandas series is used to find the frequencies of the words. Pandas data frame is used to store word frequencies and unique neighbours.

Results: *show evidence of functionality, as well as examples of visualizations used and analyse and reflect on the resultant tool you have created.*

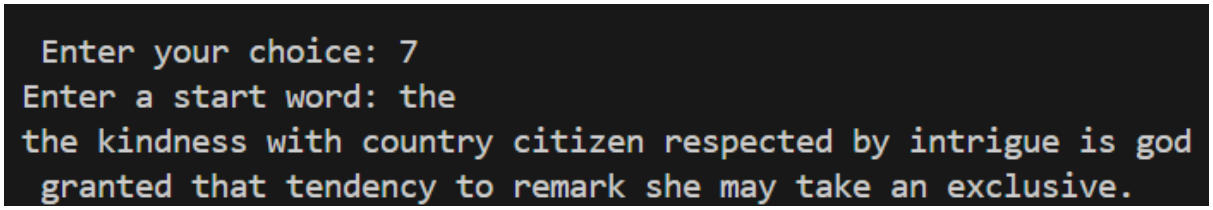
The first thing the user sees is the menu to select an option to analyse the text. The menu is presented in the console. It looks like this:



The user will enter a number in accordance with the displayed menu. The analysis is performed by different supporting functions. The data structures are populated when the program runs.



When the user enters an option, the program will output the desired analysis.



The data should be in the same directory as the program, otherwise the program will not open the file. The file name should be passed as an argument to the function.