* 1. Password Generation

The password generation algorithm designed uses Regression to achieve dynamic password creation with elegance and efficiency. Other options could include chained For loops; however, this logic would require much more thorough analysis and could be confusing to explain. Making use of recursion allows for an easy explanation of the process and requires minimal code for execution. In languages that are interpreted, such as python, this is highly relevant. Within the python compiler, each line is compiled individually and thus, the code must be as minimal as possible.

The rough approach for this solution was to consecutively add a new character to the current set of characters. Once the length has been reached, the defined rules should be checked to ensure that the password does meet the requirements. Only then should the password be added to the list. Otherwise, the algorithm should continue to the next potential password. A more detailed description can be found with the pseudocode provided in Figure 1.

A screenshot of a computer program

Description automatically generated

(Figure 1 – Pseudocode for Password Generation)

The documented flow of the program shows how the exponential growth of increasing password size. Given the length of a password (inputs), *n*, the number of categories (branching factor), 4, and the average number of chacaters per category 4.5 (5,5,5,3), the base worst case time complexity takes form of O((4\*4.5)n), or O(18n). The algorithm could theoretically be improved to achieve the best time complexity of O(x < 18n), where x is the final time complexity after formatting. In the Algorithm’s current state, it has a fixed notation of O(18n) because the formatting occurs as the last step. If step 5.1 were to be brought before step 5, the algorithm would instantly become more efficent as the invalid passwords, which start without a capital or lowercase letter, wouldn’t be created past the initial character.

* 1. Longest Substring

The algorithm chosen for finding the longest substring is an adaptation of Sliding Window, giving it an adaptive size. Given the unknown length of the longest string, the base window must remain a single character. From this base window, a secondary view is made. This secondary view should iteratively grow until the conditions for K are met. If a longer string is found, the string height is updated, and the base window position is saved in new list. If a string of the same length is found, the position is appended to the list. Otherwise, the algorithm continues to the next position. This is described in greater detail by Figure 2. A black and white list of text

Description automatically generated

(Figure 2 – Pseudocode for Adaptive Sliding Window)

As mentioned, the strings found are stored as locations in the Integer format. Given python’s integer size, 28 bytes, and a character size of 50 bytes: the resulting memory usage is

O(((n \* 50) + (m \* 50) + 50) + ((x \*28) + 28))

Where n is the length of the text to be analysed, m is the length of the adaptive window and x is the number of longest string positions. This is much better than if found strings were stored as a string as only one value is needed. Additionally, the stored values take far less memory to store, using 22 bytes less for each instance.

The time complexity for this algorithm is O((m \* n )\* x) where m is the length of the text, n is the remaining length of the text and x is the number of different characters in the current substring

* 1. Pattern Finding

Similar to 1.2, this function uses a typical Sliding window approach

* 1. Cheapest Tickets