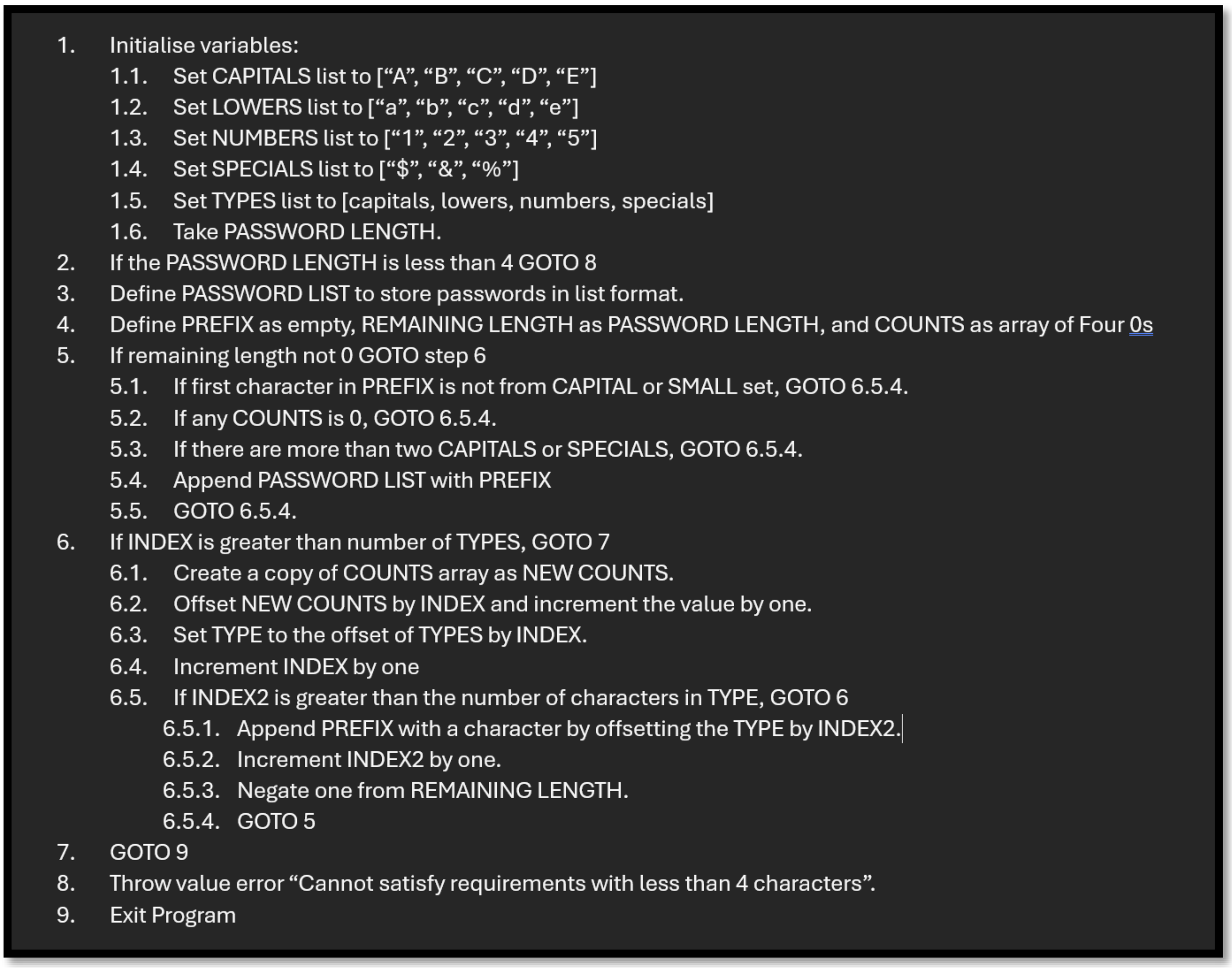
1.1 Password Generation

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.



```
1.    Initialise variables:
      1.1.   Set CAPITALS list to ["A", "B", "C", "D", "E"]
      1.2.   Set LOWERS list to ["a", "b", "c", "d", "e"]
      1.3.   Set NUMBERS list to ["1", "2", "3", "4", "5"]
      1.4.   Set SPECIALS list to ["$", "&", "%"]
      1.5.   Set TYPES list to [capitals, lowers, numbers, specials]
      1.6.   Take PASSWORD LENGTH.
2.    If the PASSWORD LENGTH is less than 4 GOTO 8
3.    Define PASSWORD LIST to store passwords in list format.
4.    Define PREFIX as empty, REMAINING LENGTH as PASSWORD LENGTH, and COUNTS as array of Four 0s
5.    If remaining length not 0 GOTO step 6
      5.1.   If first character in PREFIX is not from CAPITAL or SMALL set, GOTO 6.5.4.
      5.2.   If any COUNTS is 0, GOTO 6.5.4.
      5.3.   If there are more than two CAPITALS or SPECIALS, GOTO 6.5.4.
      5.4.   Append PASSWORD LIST with PREFIX
      5.5.   GOTO 6.5.4.
6.    If INDEX is greater than number of TYPES, GOTO 7
      6.1.   Create a copy of COUNTS array as NEW COUNTS.
      6.2.   Offset NEW COUNTS by INDEX and increment the value by one.
      6.3.   Set TYPE to the offset of TYPES by INDEX.
      6.4.   Increment INDEX by one
      6.5.   If INDEX2 is greater than the number of characters in TYPE, GOTO 6
             6.5.1.   Append PREFIX with a character by offsetting the TYPE by INDEX2.
             6.5.2.   Increment INDEX2 by one.
             6.5.3.   Negate one from REMAINING LENGTH.
             6.5.4.   GOTO 5
7.    GOTO 9
8.    Throw value error "Cannot satisfy requirements with less than 4 characters".
9.    Exit Program
```

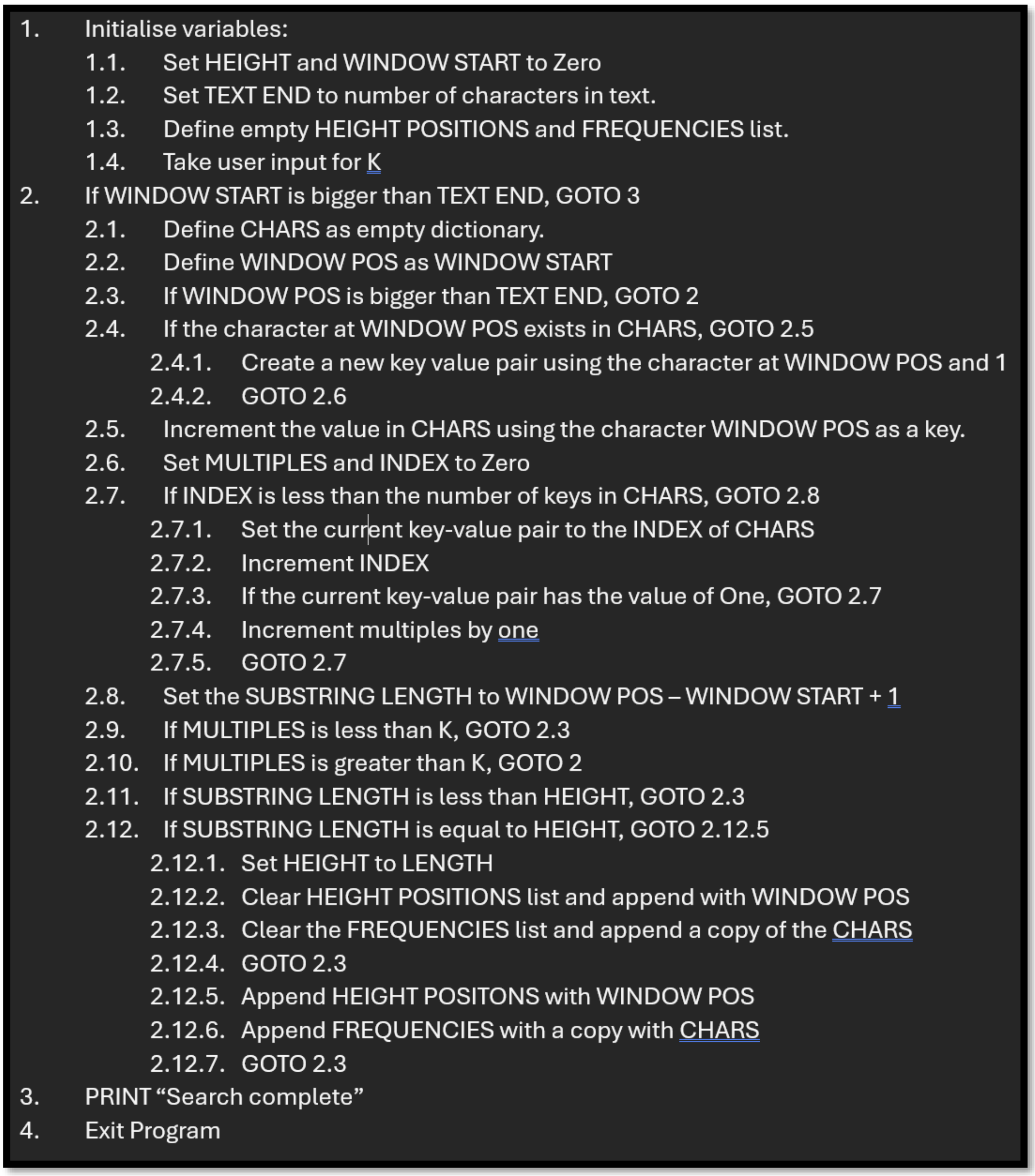
(Figure 1 – Pseudocode for Password Generation)

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 $O(18^n)$. The algorithm could theoretically be improved to achieve the best time complexity of $O(x<18^n)$, where x is the final time complexity after formatting. In the Algorithm's current state, it has a fixed notation of

O(18^n) 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.

1.2 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.



```
1.      Initialise variables:
        1.1.    Set HEIGHT and WINDOW START to Zero
        1.2.    Set TEXT END to number of characters in text.
        1.3.    Define empty HEIGHT POSITIONS and FREQUENCIES list.
        1.4.    Take user input for K
2.      If WINDOW START is bigger than TEXT END, GOTO 3
        2.1.    Define CHARS as empty dictionary.
        2.2.    Define WINDOW POS as WINDOW START
        2.3.    If WINDOW POS is bigger than TEXT END, GOTO 2
        2.4.    If the character at WINDOW POS exists in CHARS, GOTO 2.5
                2.4.1.  Create a new key value pair using the character at WINDOW POS and 1
                2.4.2.  GOTO 2.6
        2.5.    Increment the value in CHARS using the character WINDOW POS as a key.
        2.6.    Set MULTIPLES and INDEX to Zero
        2.7.    If INDEX is less than the number of keys in CHARS, GOTO 2.8
                2.7.1.  Set the current key-value pair to the INDEX of CHARS
                2.7.2.  Increment INDEX
                2.7.3.  If the current key-value pair has the value of One, GOTO 2.7
                2.7.4.  Increment multiples by one
                2.7.5.  GOTO 2.7
        2.8.    Set the SUBSTRING LENGTH to WINDOW POS – WINDOW START + 1
        2.9.    If MULTIPLES is less than K, GOTO 2.3
        2.10.   If MULTIPLES is greater than K, GOTO 2
        2.11.   If SUBSTRING LENGTH is less than HEIGHT, GOTO 2.3
        2.12.   If SUBSTRING LENGTH is equal to HEIGHT, GOTO 2.12.5
                2.12.1. Set HEIGHT to LENGTH
                2.12.2. Clear HEIGHT POSITIONS list and append with WINDOW POS
                2.12.3. Clear the FREQUENCIES list and append a copy of the CHARS
                2.12.4. GOTO 2.3
                2.12.5. Append HEIGHT POSITONS with WINDOW POS
                2.12.6. Append FREQUENCIES with a copy with CHARS
                2.12.7. GOTO 2.3
3.      PRINT "Search complete"
4.      Exit Program
```

(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.

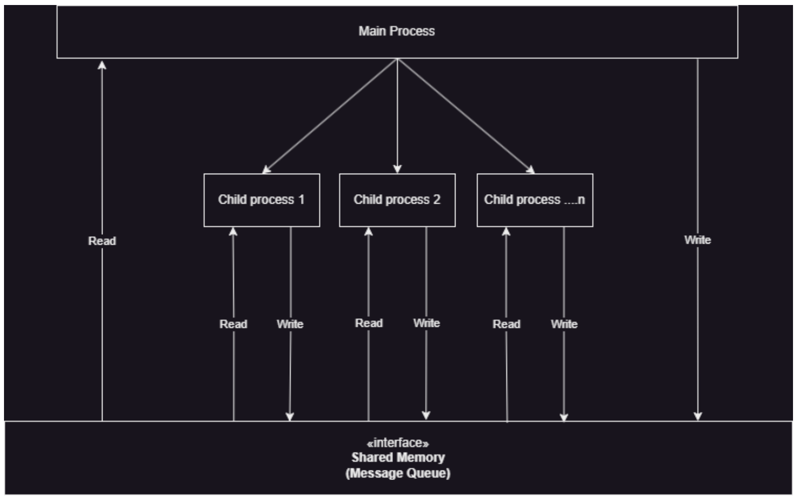
The time complexity for this algorithm is O((m*n)*x) where m is the length of the text, n is the remaining length of the text and x is the number of different characters in the current substring

1.3 Pattern Finding

Unlike task 1.2, it is typical to know the pattern before searching for it. This means more complex algorithms can be used to dramatically increase the efficiency. Developed by Knuth Morris and Pratt, KMP employs a failure function to skip over repeated evaluation. Comparing the approach seen in 1.2, the time complexity seen here is instead O(m+n): where n is the length of the text and m is the length of the pattern.

This comparison makes a stark difference when considering the completion of parallel processes as the limiting factor in locating the next patterns.

As shown in Appendix 1.1, depending on the specifications of the machine, different paths are taken with dramatically different processes. This is designed to ensure accessibility across multiple hardware levels whilst maintaining maximal efficiency. Parallelism in this task allows for multiple sections of the text to be scanned at once, or for multiple texts to be scanned at once, and combine values through shared memory, demonstrated by Figure 3.



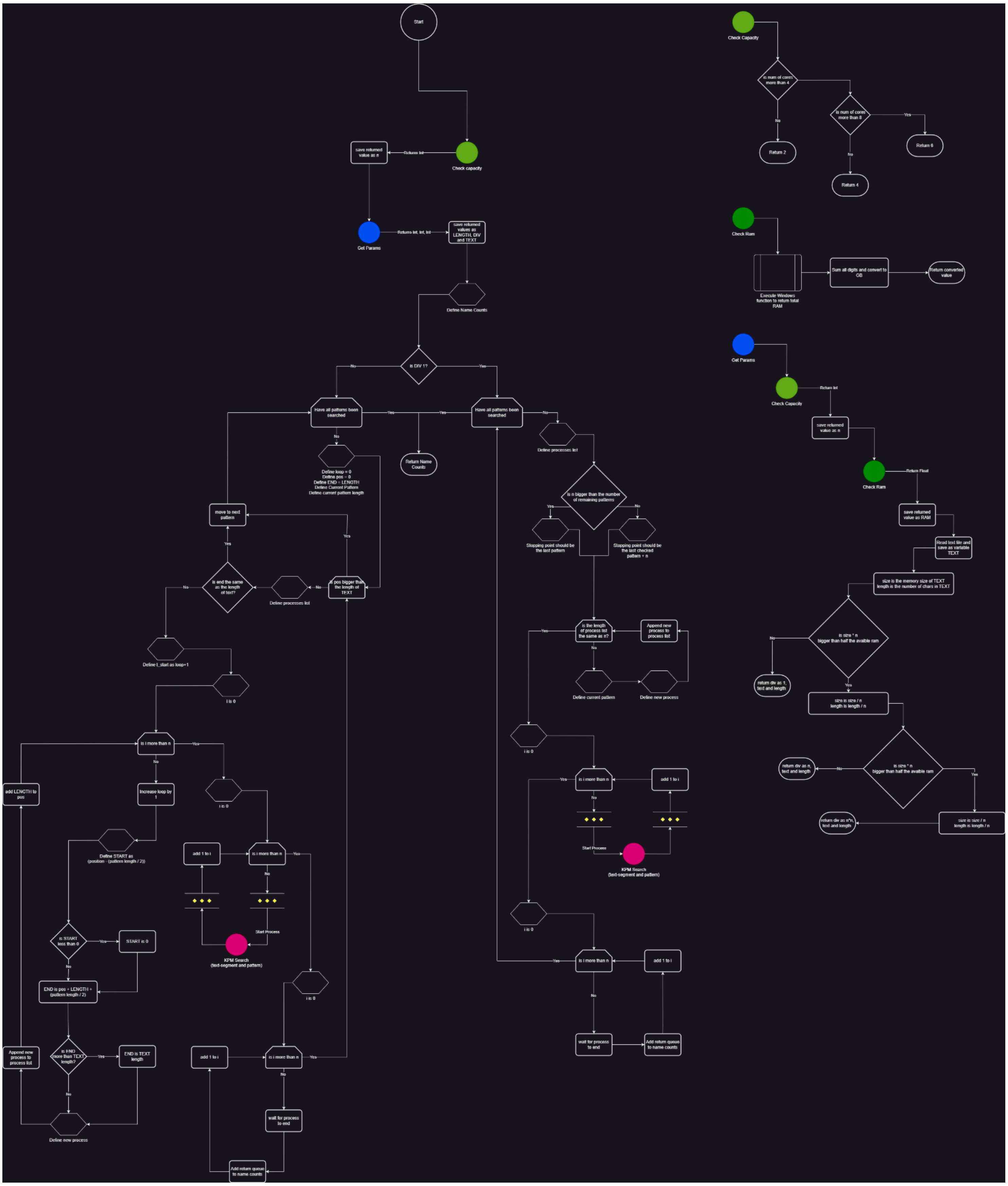
(Figure 3)

1.4 Cheapest Tickets

The algorithm in this task was developed by a Dutch computer scientist, Edsger Dijkstra. This well-known algorithm hails as one of the most efficient ways to find the best path between two vertices depending on a weight. This weight could represent any metric, so long as it remains positive. Common examples of this weighted value could include distance or prices. Choices were made here to allow operators to manually add new stations and paths. Additionally, sub algorithms were introduced to bring quality of life when searching for stations. If a station is mis-spelt, or not written completely: the application will suggest stations which have a matching substring. These stations are therefore always validated before entering the system.

Appendix:

1.1 Flowchart for parallel search



Start
Check capacity
Returns Int
save returned value as n
Get Params
Returns Int, Int, Int
save returned values as LENGTH, DIV and TEXT
Define Name Counts
is DIV 1?
No
Yes
Have all patterns been searched
Have all patterns been searched
Return Name Counts
Define loop = 0
Define pos = 0
Define END = LENGTH
Define Current Pattern
Define current pattern length
Define processes list
is n bigger than the number of remaining patterns
Stopping point should be the last pattern
Stopping point should be the last checked pattern + n
move to next pattern
is end the same as the length of text?
Define processes list
is pos bigger than the length of TEXT
Define I_start as loop+1
i is 0
is i more than n
add LENGTH to pos
Increase loop by 1
Define START as (position - (pattern length / 2))
is START less than 0
START is 0
END is pos + LENGTH + (pattern length / 2)
is END more than TEXT length?
END is TEXT length
Append new process to process list
Define new process
add 1 to i
Start Process
KPM Search (text-segment and pattern)
wait for process to end
Add return queue to name counts
is the length of process list the same as n?
Append new process to process list
Define current pattern
Define new process
Check Capacity
is num of cores more than 4
is num of cores more than 8
Return 2
Return 4
Return 8
Check Ram
Execute Windows function to return total RAM
Sum all digits and convert to GB
Return converted value
Return Int
Check Float
Return Float
save returned value as RAM
Read text file and save as variable TEXT
size is the memory size of TEXT length is the number of chars in TEXT
is size * n bigger than half the avalible ram
return div as 1, text and length
size is size / n length is length / n
return div as n, text and length
return div as n*n, text and length