This document contains an explaination of algorithmic design and solutions to two problems. First will be the statement of the question followed by the mathematical formulation, the Brute-Force solution, our solution, proof of correctness for our solution and any slight improvements we could do.

1 Turtle Graphics Problem

Our program will allow the user to program a virtual turtle to move around the graphics window, drawing pictures as it goes. The virtual turtle in this application can move forward, turn left, and turn right. Each of these operations takes a parameter that indicates either how far the turtle should move or how many degrees it should turn. The turtle also comes equipped with a virtual pen in its belly that it can use to draw a line on the graphics window.

1.1 Mathematical Formulation

In the TurtleTokenizer algorithms we will be taking in a String of length N and removing all white-spaces so that the resulting string is of length N^* were $N^* \leq N$. We say that this string can be composed as $n_0 n_1 ... n_k = N^*$ where k is the number of tokens. Let M then represent the longest n_i. (*Note that each token does not need to be valid*)

1.2 Solution

We approach the problem by first determining how to split up tokens. We do this using our Tokenizer algorithm which will take a string and, upon request give the next token (action) that will be executed by TurtleGraphics.

Within the Tokenizer, we begin by initializing our tokens to be a string with no spaces and all letters being uppercase as well as keeping track of our current place within the string at all times. *(Note: The string is also checked for miss-matching backets, so that this is a valid input in that sense)*

Algorithm 1 Tokenizer I

procedure INITIALIZE(*command*)
command ← remove white-spaces and make all uppercase
command.ASSERTVALIDITY( ) // check to make sure that {}'s match up
(String) *tokens* ← *commands*
currentIndex ← 0

Whenever the user calls nextToken, a stringbuilder is initialized which will be returned at the end. We then have a defense for whether or not there is an invalid command entered *(i.e. 10 by itself)* by using the fact that a **valid** token must begin with one of the characters {F,R,L,X,D,U,'{'}. If it is not one of these, Tokenizer will skip over the character and continue. If one of the letters is found, it will append the letter to the stringbuilder and check for proceeding numbers. If a number is found it will be appended. The current token is complete, and the string is returned. If the next char is '{', Tokenizer keeps track of how many open and closed brackets are seen and builds the next token from the characters within the braces if there are an equal number of open and closed brackets. The reason for the counter is to ensure that all nested brackets are also accounted for. Once this (possibly mega) token has been built, it is returned.

Algorithm 2 Tokenizer II

```
procedure nextToken( )
    token ← String
    while currentIndex <tokens.length( ) do
        if tokens.charAt(currentIndex) ∈ {F, R, L, X, D, U} then
            token.append(letter)
            while nextLetterIsANumber do
                token.append(number)
            return(token)
        else if tokens.charAt(currentIndex) == "{" then
            numOpen = 1
            while thereAreMoreChars && numOpen > 0 do
                token.append(char)
                if nextchar is { then
                    numOpen++
                else if nextChar is } then
                    numOpen−−
                    if numOpen == 0 then
                        token.append(})
            return(token)
        else
            currentIndex++
    return(token)
```

TurtleGraphics takes the given token and uses the *execute* method to scan it for one of the command characters {F, R, L, X, D, U}. If the command is F, R, or L, TurtleGraphics moves the Turtle the specified amount or default distance if proceeding number is given. If the token contains U or D, the Turtle raises or lowers its pen to draw. If a token containing X is given, the Turtle will be executing a token, separated by {}, repeated a specified number of times. *execute* gets the number proceeding X and the token contained within the brackets. Then the *execute* is called the specified number of times on the bracketed token.

Algorithm 3 Turtle Graphics I

```
procedure EXECUTE(command)
    tokenizer ← TurtleTokenizer(string)
    while tokenizer.HASMORETOKENS( ) do
        if nextToken = 'F' then
            if nextToken only 'F' then
                turtle.FORWARD(50)
            else
                turtle.FORWARD(trailingNumbers)
        else if nextToken = 'L' then
            if nextToken only 'L' then
                turtle.LEFT(90)
            else
                turtle.LEFT(trailingNumbers)
        else if nextToken = 'R' then
            if nextToken only 'R' then
                turtle.RIGHT(90)
            else
                turtle.RIGHT(trailingNumbers)
        else if nextToken = 'U' then
            turtle.PENUP( )
        else if nextToken = 'D' then
            turtle.PENDOWN( )
        else if nextToken = 'X' then
            numRepeats ← trailingNumbers
            repeatable ← nextToken
            for numRepeats do
                EXECUTE(repeatable)
```

ReplaceAction begins by getting the replacement field string from the user interface. For the length of the string, our program looks through the first half up to the - symbol of the ->and appends each character to a StringBuilder original. Next the program builds the replacement commend from the latter half of the replacement field. ReplaceAction then calls the helper method replaceAll to replace all instances of the original command and return the new token to the user interface.

Algorithm 4 Turtle Graphics II

```
procedure replaceAction( )
    replacement ← ui.getReplacementField( )
    replacement ← remove white-spaces, make all uppercase and assert validity
    hasArrow = false
    for replacement.length( ) do
        if there is a "->" then
            hasArrow = true
    if !hasArrow then
        Throw Exception
    original ← StringBuilder
    for replacement.length( ) do
        if current_char = '−' then
            break( )
        else
            original.append(current_char)
    replaceWith ← StringBuilder
    for from (original.length( ) + 1) to replacement.length( ) do
        replaceWith.append(current_char)
    oldProgramText ← ui.getProgramText( )
    newProgramText ← replaceAll(replacement, original, replaceWith)
    ui.setProgramText(newProgramText)
```

ReplaceAll scans the original token given for the original command to be replaced with the replacement command. Scanning each character of the token, ReplaceAll checks for the first character of the original command. If the character is not the same as the original command, it is appended to the new token. If the same character is seen, ReplaceAll looks ahead to check if the commands are the same. If they are, the original command is replaced by the replacement comment. If not, ReplaceAll continues appending characters. Once the end of token is reached, the newly built token with all original commands replaced is returned.

Algorithm 5 Turtle Graphics III

```
procedure replaceAll(token, original, replacement)
    result ← StringBuilder
    currentIndex = 0
    while currentIndex < token.length() do
        if currentChar = original.firstChar then
            for original.length() do
                if chars are not equal then
                    BREAK
            if Found original then
                result.append(replacement)
            else
                result.append(stringFrom_currentIndex)
            update currentIndex
        else
            result.append(currentChar)
            currentIndex++
    return(result)
```

1.3 Correctness

Proposition 1. *The TurtleTokenizer algorithm either outputs the next valid token while ignoring invalid tokens.*

Proof. Our Tokenizer algorithm will "clean" the inputed string so that if it has invalid brackets, an exception is thrown and all of the spaces are removed to allow for accurate pointers to keep track of the current position. This cleaned version is stored as well as the pointer as instance variables allowing for the user to easily ask for the *nextToken*(), returning the next valid token. The validity of the token is assertained by the if-statements, if not a valid letter or not a '{', the program will skip this portion of the string. The program will also determine if a multiplier is followed by a '{', and if not will throw the appropriate exception. □

Proposition 2. *The execute() algorithm will interpret the user inputted string and carry out the corresponding commands*

Proof. This is more of a trivial proof as it is largely dependent on the success of the aforementioned tokenizer algorithm, however when it gets a valid token (determined to be so by the tokenizer), it will execute the command with the starting letter. If the token is a multiplyer, it will parse the number of repetitions from it, and use a for-loop to execute the subsequent command block, calling *execute* recursively. □

Proposition 3. *The replaceAction() algorithm will replace all occurances of a given command with a user specified command and throw an exception if the command called with and invalid replacement statement.*

Proof. The *replaceAction*() first checkes that the replacement statement is valid through linearly searching for the "->" symbol, if not found will end. Otherwise it will get the original command to be replaced and the command to replace it. It will then linearly search through the tokens again and look for every instance where the original is. If found, the program will replace it. Otherwise it will keep appending the original commands. □

1.4 Analysis

Proposition 4. *The nextToken() algorithm will have time and space complexity in respect to M, the length of the longest token.*

Proof. Since TurtleTokenizer is a linear process that moves through each index of the token, the average time complexity is $\mathbf{O(M)}$ where M is the length of the longest token.

Giving us an overarching $\mathbf{O(M)}$ complexity.

□

Proposition 5. *The execute algorithm will have time and space complexity in respect to $N^{\dagger}$, the longest expanded string of commands*

Proof. Given an input of length N, if it is expanded to its maximum length where all loops are expanded and written out, it can be represented as $N^{\dagger}$, where $N^{\dagger} \geq N^{*}$, the length of the cleaned-up strings. Since from this expansion, the *execute* algorithm simply linearly goes through the string and executes all valid tokens.

This gives us an overarching $\mathbf{O(N^{\dagger})}$ complexity.

□

Proposition 6. *The replaceAction() algorithm will have time and space complexity of $\boldsymbol{O(N + (N^{\ddagger}+k+m))}$ where:*

- $\boldsymbol{N}$*: The length of the original string of tokens*
- $\boldsymbol{N^{\star}}$*: The length of the original string of tokens without spaces, $N^{*} \leq N$*
- $\boldsymbol{n}$*: The length of the token to be replaced*
- $\boldsymbol{k}$*: The number of occurences of the above token*
- $\boldsymbol{m}$*: The length of the tokens that will be doing the replacing*
- $\boldsymbol{N^{\ddagger}}$*: $N^{\star} - k * n$*

Proof. This is the case since we will first go through the original to remove spaces, $\mathbf{O(N)}$, making $N^\star$ the linear search then we go throug and replace all of the occurances of the tokens to be replaced with the tokens to be replacing. Note that if the tokens to be replaced are the longer ones, will not exceed $\mathbf{O(N^\star)}$. Otherwise the we have a string of length$N^\ddagger + k + m$ to build which is also a linear process, $\mathbf{O(N^\ddagger{+}k{+}m)}$

Giving us an overarching $\mathbf{O(N + (N^\ddagger{+}k{+}m))}$ complexity.

□

1.5 An Example

We first note that only an example of the TurtleGraphics working is sufficient to show that TurtleTokenizer works. For this example we entered the string: X3 {F81 L120}, this produces a triangle of side length 81. We then used the replace functionality with the command F81 -> F27 R60 F27 L120 F27 R60 F27 to produce: X3 {F27 R60 F27 L120 F27 R60 F27 L120}. This produces a Koch snowflake of order 1. We then employed the following replace commands to create a 4th order Koch snowflake:

- **2nd Order:** F27 -> F9 R60 F9 L120 F9 R60 F9
- **3rd Order:** F9 -> F3 R60 F3 L120 F3 R60 F3
- **4th Order:** F3 -> F1 R60 F1 L120 F1 R60 F1

Resulting in:
X3 {F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1

L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1
L120 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1
R60 F1 R60 F1 L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120 F1 R60 F1
L120 F1 R60 F1 R60 F1 R60 F1 L120 F1 R60 F1 L120}

This is sufficient to show that the Turtle algorithms all work since when we press run on the ui, it will execute the given code correctly, meaning that the execute code is functioning correctly and therefore also the Tokenizer class.

2 Anagrams Problem

Two strings are anagrams of each other if they contain the same characters with the same frequencies: "stops" and "psost" are anagrams, while "stops" and "stoop" are not. (It's not necessary that the permuted strings be English words.)

Your program will read a text file, with its name specified as a command line argument, that contains one string of characters per line. For each line of the file, your program must print all anagrams of the string in alphabetical order to standard output, with any repetitions removed. The output from different lines should be separated by a blank line.

2.1 Mathematical Formulation

In this problem, we will be given a word of length N consisting of k unique letters. We will find M anagrams of this word, M can be represented in terms of N as: $\frac{N!}{n_0! \cdot n_1! \cdot \ldots \cdot n_k!}$ where n_i = number of repetitions for each letter s.t. $n_0 + n_1 + ... + n_k = N$ (*Note that* n_i *may* $= 1$)

2.2 Solution

Our algorithm for finding all anagrams in a list of words is accomplished through several steps. We look at one word at a time, checking its length, N. If $N > 1$, this is the trivial case, if not we begin to enumerate the Anagrams.

Algorithm 6 Initialization Step.

```
procedure ANAGRAMS(fileName)
    while fileName contains at least 1 word do
        word ← fileName
        if word.length <2 then
            print (word)
        else
            anagrams = Initialize HashSet
            findANAGRAMS(word)
```

Begin by sorting the word and enumerating the anagrams via a *swap* method. *Note* : *We do not enumerate these in lexiographical order, see below for details.* Once all of the anagrams have been found, we move onto the next phase which would be to take all of the anagrams and sort them so that they are in alphabetical order.

Algorithm 7 Finding Anagrams

```
procedure findANAGRAMS(word)
    sort and input (word) into hashset
    for c_0 ∈ word do
        for c_1 ∈ word \ {previouscharacters} do
            if c_0 equals c_1 then
                CONTINUE
            else
                SWAP(word, c_0, c_1)
    SORT(anagrams)
    PRINT(anagrams)
```

Swap makes recursive calls checking to see if the permutation of the word has already been found via a hashset, inserting them if they are not contained within the set.

Algorithm 8 Swaping Characters

```
procedure SWAP(word, c_0, c_1)
    newWord ← Switch characters c_0 and c_1 within word
    if newWord ∉ anagrams then
        anagrams.INSERT(newWord)
    INCREMENT(c_0 and c_1)
    for c_0 ∈ newWord do
        for c_1 ∈ newWord \ {previouscharacters} do
            if c_0 equals c_1 then
                CONTINUE
            else
                SWAP(newWord, c_0, c_1)
```

2.3 Correctness

Proposition 7. *This algorithm with effectively find and print out all of the M Anagrams of a word of length N consisting of k unique letters, the frequency of each can be represented as n_i s.t. $n_0 + n_1 + ... + n_k = N$.*

Proof. We do this proof inductively.

Base Cases:
For $N = 1$, (i.e. word is "a"), this is trivial, so look instead at $N = 2$. Here we have 2 cases:

1. $\mathbf{k = 2}$, word = "ab"
2. $\mathbf{k = 1}$, word = "aa"

In **case 1**, the algorithm would sort first to "**ab**" and store that, then swap the first letter, 'a', with the next distinct letter, 'b', producing "**ba**". This would end the program as there are no other swaps to be made. In **case 2**, the algorithm would sort first to "**aa**" and store that, then see that the two characters are the same so it would end the program as there are no other swaps to be made.

The next step would be $N = 3$ where we have the following cases:

1. $\mathbf{k = 3}$, word = "abc"
2. $\mathbf{k = 2}$, word = "aab"
3. $\mathbf{k = 1}$, word = "aaa"

In **case 1**, the algorithm would sort first to "**abc**" and store that, then swap the first letter with the next distinct letter, 'b', producing "**bac**". Now 'b' is an anchor and so we will look to enumerate all anagrams of the remainder, putting 'b' in front.

Note that the remainder is like that of **case 1** when $N = 2$ so the resulting product would be "**bca**" since "**bac**" was found already. Now we return to the original sorted string "**abc**" and swap 'a' with the next unique letter, 'c', producing "**cba**". Now 'c' is the anchor and for the same reasoning as previous, we see that the string "**cab**" is generated so we now return to the original sorted string and attempt to swap 'a' with the next unique letter. Since there are no more, we anchor 'a' and now attempt to swap 'b' with its next unique letter, 'c'. This produces "**acb**" and would end this portion of the algorithm as there are no other swaps to be made. Now that all of the possible combinations have been found, the program will sort all of the found strings and print them out.

In **case 2**, the algorithm would sort first to "**aab**" and store that, then swap the first letter with the next distinct letter, 'b', producing "**baa**". Now 'b' is an anchor and so we will look to enumerate all anagrams of the remainder, putting a 'b' in front. Note that the remainder is like that of **case 2** when $N = 2$ so there are no more strings to enumerate so we return to the original sorted string, "**aab**". Now the program will swap the next letter, 'a', with the next distinct letter, 'b'. This produces the string "**aba**". Since now 'b' is the anchor and the second 'a' has no proceeding letters, the process ends as all of the possible combinations have been found, the program will sort all of the found strings and print them out.

In **case 3**, the algorithm would sort first to "**aaa**" and recognize this to be a trivial case so it will print out the found string.

Assumption:
Let us have a word (when sorted) be $n_{1_1}n_{1_2}...n_{1_i}n_{2_1}n_{2_2}...n_{j_1}...n_{k_1}n_{k_2}..n_{k_p}$ s.t. $n_{l_1} + ... + n_{l_m} = n_1$ and $n_1 + ... + n_k = N$ for $i, j, k, l, m \in \mathbb{N}$. We assume that we can find every anagram for this form of word.

***k+1* Step:**
Now we use the same logic as we did in the cases of the $N = 3$ stage when we had "anchor" substrings so that way we enumerate first all of the strings that begin with n_{2_1}, then n_{2_2}, ..., then finally n_{k+1_1} and n_{1_1}. Therefore the algorithm will progress anchoring also the next term, up until there are only have 3 terms not anchored which we saw from the base cases will be fully enumerated. Therefore the algorithm continues swapping in letters until all cases have been found.

□

2.4 Analysis

Proposition 8. *Our algorithm will have time and space complexity in respect to the number of anagrams M that can be derived from a word of length N with k unique letters such that $n_0 + n_1 + ... + n_k = N$. (Recall: $M = \frac{N!}{n_0! \cdot n_1! \cdot ... \cdot n_k!}$). We say that this algorithm has:*

***O(M· log(M))** complexity.*

Proof. We begin this proof by stating that it is impossible to get better than O(M) since every possible enumeration must be explored and worst case is that there are no repeating letters so this would be $N!$ outcomes so an $N!$ algorithm must be used. This being said, our algorithm will only enumerate each possible outcome once and due to fact that the senario where two of the same letters are swapped which would lead to an anagram that has already been found has been eliminated, our algorithm performs at the afformentioned M complexity for finding the anagrams. After we have found all of the anagrams, we now must put them in alphabetic order, which we do through linearly copying them into an array, followed by sorting them, and finally printing them. Therefore we can list the complexities as such:

1. **Finding Anagrams:** O(M) Enumerating all possible M anagrams
2. **Copying:** O(M)
3. **Sorting:** O($M \cdot log(M)$) Sorting all M found anagrams
4. **Printing:** O(M)

Giving us an overarching **O(M· log(M))** complexity.

□

2.5 An Example

Say that we input the simple word **jeep**. Before beginning tracing, let us first observe that this word has $N = 4$ letters, which implies that it has $M = \frac{4!}{1!\cdot 2!\cdot 1!} = 12$ anagrams. to solve, first the algorithm would initialize our HashSet with the sorted form of the word **eejp**:

HashSet	
Stage-Found	**Anagram**
sorted	eejp

Now the **FindAnagrams** portion beings, pointers i_0 and $j_0 = 0$ and 1 respectively so that they are pointing to **e** and **e**. Since they are equal, j increments so that $i_0 = 0,\ j_0 = 2$

i.e. i_0 = first **e** and j_0 = **j**

These two letters are not the same so we call *swap*

Within $S_1 = swap(eejp,\ 0,\ 2)$; we swap **e** and **j** to get **jeep**, since this is not in the HashSet, we add it.

HashSet	
Stage-Found	**Anagram**
sorted	eejp
S_1	jeep

Now we will attempt to recursively call swap, setting $i_1 = 1$ and $j_1 = 2$. since our word now is **jeep**, we see that the letters are again both i_1 and j_1 points to **e** so increment $j_1 = 3$ now. Our next call of *swap* therefore will be $S_2 = swap(jeep, 1, 3)$; resulting in the word **jpee**, since not in the HashSet, we add it.

HashSet	
Stage-Found	**Anagram**
sorted	eejp
S_1	jeep
S_2	jpee

Now will attempt to recursively call swap again, setting i_2 and j_2 to 2, and 3 respectively, seeing again that both i_1 and j_1 points to **e** but since we have reached the end of the for loop, S_2 call will end.

Returning to S_1 (recall that the word is **jeep**, $i_1 = 1$ and $j_1 = 3$). We continue in the loop such that now $i_1 = 2$, $j_1 = 3$. Therefore i_2 points to the second **e** and j_0 to **p**. Swapping them through calling: $S_3 = swap(jeep, 2, 3)$; results in the word **jepe**, not in the HashMap so we add it.

Note: the for loop will not be able to execute here so S_3 terminates here.

HashSet	
Stage-Found	**Anagram**
sorted	eejp
S_1	jeep
S_2	jpee
S_3	jepe

Returning to S_1 (recall that the word is **jeep**, $i_1 = 2$ and $j_1 = 3$) we have ended the for loop now so terminate S_1 returning to the original *sorted* state.

*Note: we have successfully found all anagrams of **jeep** beginning with **j**!*
Recall the word now is **eejp** and $i_0 = 0,\ j_0 = 2$, continuing in the for loop we get, $i_0 = 0,\ j_0 = 3$, so we call $S_4 = swap(eejp, 0, 3)$; resulting in the word **peje**. From here much like before, we shall find all anagrams of **jeep** beginning with **p** this time, finally followed by all anagrams beginning with **e** so our HashSet will look like:

HashSet	
Stage-Found	**Anagram**
sorted	eejp
S_1	jeep
S_2	jpee
S_3	jepe
S_4	peje
S_5	pjee
S_6	peej
S_7	ejep
S_8	ejpe
S_9	epje
S_{10}	epej
S_{11}	eepj

Now that we have reached the end of the enumerating stage, we come to sorting the above HashSet and printing all of them out.