ASSIGNMENT 1

PIYUSH CHAUHAN
2021CS11010

## OVERVIEW:

This algorithm uses the long division method to calculate the square root of an integer in the form of a string. It first separates the input string into pairs of digits and stores them in a list, which is then reversed. Next, the algorithm uses a helper function called "nearest Sqrt" to find the nearest square value which is less than or equal to the first digit of the input string. This nearest square value is then used as the first digit of the square root.
The algorithm then uses another helper function "step1" to update the input list by subtracting the square of the nearest square value from the first digit of the input string. This updated list is then passed to the next helper function "helper1" which uses a recursive approach to find the next digits of the square root.

The function "helper1" takes the updated list and the nearest square value as inputs and starts a loop. In each iteration of the loop, it concatenates the first two digits of the input list, and uses a function "recurse" to find the next digit of the square root by dividing the concatenated digits by a value calculated from the nearest square value and the current digit of the square root. It then updates the input list by subtracting the product of the current digit of the square root and a value calculated from the nearest square value and the current digit of the square root. The current digit of the square root is then added to the final square root value.

The loop continues until the input list has only one digit left. At that point, the final square root value and the remaining digit in the input list are returned as the integral part and the remainder of the square root respectively.

This algorithm can be useful when the input integer is very large and cannot be stored or processed directly as a whole number. The long division method allows the calculation of the square root to be done incrementally and in a way that is memory efficient. However, it may have a relatively large time complexity due to the many recursive calls and looping through the input list.

## PSEUDO CODE:

```
1 Define a function "isqrtld" that takes in a value (string)
2 Divide the input value into pairs of digits and store them in a
      list "ls"
3 Find the nearest square root of the first element in "ls" using a
      helper function "nearestSqrt"
4 Subtract the square of the square root found in step 3 from
      the first element in "ls" and store the result in a new list
      "ls_in”.
5 Concatenate the first and second element of the list
6 Use another helper function "gusser" to find the next digit of
      the square root using the updated value in "ls_in" and the
      previous digit of the square root.
7 subtract (20*the previous digit + next_dig)*next_dig from the
      first element of the ls_in
8 Repeat step 5-7 for each digit of the square root.
9 if ls_i contains only one element then return.
10    Concatenate all the digits of the square root and return the
      result.
11    End.
```

- **Dividing into pairs;**

```
1 define a function divideHelper(s: string, acc: list)
2 if the size of string s is 0, return acc
3 else, if size is odd then add 1st element of string to list and
      call divideHelper(substring of s starting from index 2 with
      length size of s - 2, append substring of s starting from
      index 0 with length 2 to acc) and assign the result to acc
4 else do the above step without adding the first element.
5 return acc.
```

- **nearestSqrt:**

```
1 Create a variable "guess" and initialize it to 1
2 Create a function "tryGuess" which takes in two variables "m"
      and "n" of type int
3 Inside "tryGuess":
4 a. Create a variable "temp" and set it to the value of "m"
      multiplied by itself
5 b. Create a variable "temp2" and set it to the value of "(m+1)"
      multiplied by "(m+1)"
6 c. Check if "n" is greater than or equal to "temp" AND "temp2"
      is greater than "n"
7 d. If the above condition is true, return "m"
8 e. If the above condition is false, call "tryGuess" again with
      updated values for "m" and "n"
9 Inside the "nearestSqrt" function, call "tryGuess" function with
      variables "guess" and "n"
10    Return the value returned by "tryGuess" function
11    Function ends
```

- **Gusser:**

```
1 Define a function called “gusser” that takes in 3 inputs:
      "guesso", "one", and "x"
2 Initialize a variable "guess" as 0
3 Check if the following condition is true: (((20*x)+guesso)
      guesso <= one ) and ((((20x)+(guesso+1)) *(guesso+1 ))>
      one )
4 If the condition is true, return "guesso"
5 If the condition is false, increment "guesso" by 1 and call the
```

"recurse" function again with the updated "guesso", "one", and "x" as inputs

6 Repeat step 3-5 until the condition in step 3 is true

7 Return the value of "guesso" when the condition is true.

## PROOF OF CORRECTNESS:

This algorithm uses structural induction to prove that it correctly calculates the integral part and the remainder of the square root of an integer in the form of a string. The base case for this proof is when the input string has only one digit.

In this case, the "nearestSqrt" function will always return the square root of this digit, and the "step1" function will always return an empty list as the remainder. The "helper1" function will not be called in this case, and the algorithm will return the square root and an empty list as the remainder.

For the induction step, we assume that the algorithm correctly calculates the integral part and the remainder of the square root of a string with n digits. We need to show that it also correctly calculates the integral part and the remainder of the square root of a string with n+2 digits.

The "nearestSqrt" function, "step1" function and the "helper1" function are all working correctly as described above, and are being used to calculate the integral part and the remainder of the square root of the input string. In each iteration of the loop in the "helper1" function, the next digit of the square root is found by dividing the concatenated digits of the input list by a value calculated from the nearest square value and the current digit of the square root. This value is then subtracted from the input list, and the current digit of the square root is added to the final square root value.

The algorithm will continue this process until the input list has only one digit left, at which point the final square root value and the remaining digit in the input list are returned as the integral part and the remainder of the square root respectively.

Since we have assumed that the algorithm correctly calculates the integral part and the remainder of the square root of a string with n digits, and we have shown that it correctly calculates the integral part and the remainder of the square root of a string with n+2 digits using the same process, by mathematical induction we can conclude that the algorithm correctly calculates the integral part and the remainder of the square root of any string with any number of digits

**<u>Why n+2 and not n+1 in induction hypothesis?</u>**
The reason for using n+2 digits in the induction step rather than n+1 digits is because the algorithm uses pairs of digits to calculate the square root. In other words, it processes the input string two digits at a time, so in order to prove that the algorithm is correct for an input string with n+1 digits, we would need to prove it for an input string with n+2 digits.

For example, if the input string is "123", the algorithm would process it as "12" and "3" separately. Therefore, in order to prove the algorithm is correct for an input string of n+1 digits, we need to prove it for an input string of n+2 digits.

This is because the algorithm is designed to work in pairs of digits, and so in order to prove that it is correct for any input string, we need to take into account that it processes the input string two digits at a time, which is why we use n+2 digits in the induction step.

## HANDLING ARBITRARY LARGE INTEGER VALUES:

To handle the large integer case, I have defined the multiplier, addition , subtraction(using 10’s complement) and a comparator on strings.

Addition: It takes two arbitrary long strings and split them into lists. Then it starts addition from left to right and takes care of the carry as well.

Multiplier: It takes one arbitrary long string and one single length string and multiplies the two by splitting the larger string into list of string and carrying multiplication on each element of string one by one recursively.

Subtraction: it takes   two arbitrary long strings , converts one string into its 10’s complement and add it to the other string .

Comparator: It compares two strings from left to right and output 1 if first string is strictly greater than other else 0.