Name - Aastha Rajani Entry No. - 2021CS10093

To prove - The long division method gives integer square root of any given number and the smallest number which should be subtracted from that given number to obtain a perfect square.

Proof - By logical deduction.

Let us say we need to find the integer square root of any number say a.

Let b be it's integer square root. Let h be the least significant digit of b and x be the number formed by removing h from b.

At every iteration of long division method, the value of "a" gets updated, initially being the ~~mos'~~ first one or two digits of given number.

At each iteration we are required to find x and h such that

$$(10x+h)^2 \leq a \quad \text{and} \quad (10x+h+1)^2 > a.$$

Both the conditions together guarantees the uniqueness of x and h.

Simplyifying the first inequality

$$(2 \cdot 10 \cdot x + h)h \leq a - (10x)^2$$

The subtraction in the right hand side of this inequality is done by placing digits from the leftmost side.

Analyzing the left hand side of the inequality $2 \cdot 10x$ involves doubling the number x and multiplying further by 10 to make space for next digit h. We have to choose next digit h so that the new number $20x+h$ when multiplied by h equals ~~the~~ ~~no~~ or is less than ~~the updated a (obtained after multiplying the difference~~ of the number obtained after subtraction on right hand side. The value of a then gets updated by adding the next two digits of given number with 100 times the difference obtained above. This is exactly what we do in the long division process. Therefore every iteration of long division outputs the next digit of integer square root.

The process terminates when all digits are exhausted or when we are unable to find any such x and b for which the inequality above is satisfied. Hence, it is justified that this method returns the integer square root, and the lastly updated a is the smallest number which on subtracting with the given number gives a perfect square.

classmate
Date
Page

Pseudo Code

fun mul (arr, i, carr, index, pr)
arr → string of integers
i → The digit which should be multiplied
carr → Current carry
Index → the current index.
pr → product calculated so far.
if Index = -1
return pr
else.
↳ ~~Add~~ Find the product of digit at current index with i to the product calcuted so far.
↳ Concatenate the units digit of this product to pr and rest of the digits are stored as carry.
↳ Iterativiley starting from the last digit, perform digit by digit multiplication and update carry & pr until all digits are exhausted.

fun single_add (arr, carr, index, sum)
arr → string of integers
carr → Current carry
index → current index
sum → current sum.
if index = -1
then sum
else.
↳ Find the sum of digit at current index with carr
↳ Concatenate the units digit of this sum to sum and rest of digits as carry.
↳ Iteratively starting from the last digit, perform digit by digit addition and update sum and carry, until all digits are exhausted.

fun multi_adder (. arr1, arr2, carr, index, sum)
arr1, arr2 = two strings of integers of same length
carr = current carry # index = current index
sum = current sum.
if index == -1
return sum

else.
↳ Find the sum of digits of arr1, arr2 at the current index with carr.
↳ Concatenates the units digit to current sum and stores the rest in carr.
↳ Iteratively starting from the last digit, perform digit by digit addition and update carr & sum until all digits are exhausted.

fun complement(arr, index, ans).
arr → string of integers # ans → complement of
index → current index of arr upto index.
↳ Recursively subtract every digit by 9 and concatenate it to ans.
↳ Once all the digits are subtracted once by 9, we add 1 to get 10's complement.

fun make_equal (arr2, i, p)
arr2 → string of integers # p → Original ~~current~~ length.
i → Required length
↳ If i = p
return arr2
else
make_equal ("0"+arr2, i-1, p)

fun sub (arr 1, arr 2)
arr 1 = Subtractor # arr 2 = Subtrahend
↳ Find 10's complement of arr 2 using complement function
↳ Add 10's complement of arr 2 to arr 1 using multi_adder

fun compare_ge (num1, num2, i) # num1, num2 → string of integers
If i = Size(num1)
return true
else
compare the ith digits of num 1, num2.
If (num1)i > (num2)i
return true
else if (num1)i < (num2)i
return false
else compare_ge (num1, num2, i+1)

fun compare_gt (num1, num2, i)
Works similar to compare_ge, except for i = size(num1) it returns false.

fun mul_10 (s).
Adds "0" to end of s.
fun mul_100 (s)
Adds "00" to end of s.
fun compare_ge_final (num1, num2)
By make_equal ~~(m~~, make the lengths of num1, num2 equal and call compare_ge (num1, num2, Size(num1)-1)

fun multi_adder_final (num1, num2)
By make_equal, make the lengths of num1, num2 equal and call multi_adder (num1, num2, 0, size(num1)-1, "").

fun binary_search_helper (a1, a2, low, high, arr)

Takes two strings of integers and finds $h \in [0, 9]$ such that $(10a_1 + h)\,h \leq a_2$.

If high − low = 1 or 2.

$p = (10a_1 + arr[mid]) \times arr[mid]$

$q = (10a_1 + arr[low]) \times arr[low]$

$r = (10a_1 + arr[high]) \times arr[high]$

if $q \leq a_2$ and $a_2 < p$

(arr[low], $10a_1$ + arr[low])

else if $p \leq a_2$ and $a_2 < r$

(arr[mid], $10a_1$ + arr[mid])

else

(arr[high], $10a_1$ + arr[high])

else.

if $(10a_1 + arr[mid]) \times arr[mid] < a2$

binary_search_helper (a1, a2, mid, high, arr)

else

binary_search_helper (a1, a2, low, mid, arr)

```
fun binary_search( a1, a2)
    binary_search_helper( a1, a2, 0, 9, [0,1,2,3,4,5,6,7,8,9])
```

Pseudo Code

fun int_sqrt (s) :

Takes a string of integers and returns the integer sq. root and difference of int(s) & nearest perfect square,

S2 = make a list storing every character of string as integer in each index, and making the length of the list even if it is not by adding 0 at the start of the list.

call int_sqrt_helper .

fun int_sqrt_helper (r, q, num, length, l, str_list):

\# r → current value of a (as described in proof above)

\# q → the value of sqrt found so far.

\# num → the current value of divisor.

\# length → the number of digits we have operated

\# l → length of the input string

\# str_list → S2 made above.

if all the digits are operated

return (r, q)

else .

Taking num as x and r as a (in the previous proof) find h and the updated value of num. say (h, updated_num). using binary_search (num, r)

if you are not dealing with last two digits

call int_sqrt_helper (updated r, 10q+h, updated_num, length+2, l, str_list).

where updated r is updated a described in proof.

else .

call int_sqrt_helper (updated r, 10q+h, updated_num length+2, l, str_list) where updated r is just the difference of current r and $(10q+h)\cdot h$.

Ø My algorithm takes integer in the form of string as input and firstly checks if the length of this string is even or odd. In case it is odd, it adds a 0 in the beginning of the list formed by every character of input converted to Integer. It then starts with first two elements of the list (as integer) and finds the nearest square less than or equal to that number, and this digit is the most significant digit of our integer square root. The difference of first two digits and this square is added alongwith next two digits of input forms a in next recursion and twice of current integer square root multiplied to 10 forms next divisor. In this way recursion goes on until we have completed considering all the digits of input. This is what is done in the long division method which we have proved gives the integer square root. Hence, my algorithm also returns the integer square root and the ~~nearest~~ smallest number which when subtracted with the given number gives a perfect square.

Since, there is a maximum limit upto which integer can take values in sml, I have defined functions performing arithmetic operations with the help of strings in order to deal with larger inputs.