Trabalho Prático 1 - Programação Funcional e em Lógica

Test Cases

For each function developed, the following test cases were important to determine the function's efficiency and accuracy.

Original Fib.hs Functions

fibRec

Test	Expected Output	Output	Time Spent (in seconds)	Importance
0	0	0	0.01	One of the base cases for the Fibonacci sequence.
1	1	1	0.01	The other base case for the Fibonacci sequence.
10	55	55	0.01	A starter value to check the accuracy.
20	6765	6765	0.08	An even higher but important result to check if the speed holds.
30	832040	832040	3.68	An even higher where it is possible to see the processing demanded.
40	102334155	102334155	474.91	A very high number that the function takes a long time to complete.

fibLista

Test	Expected Output	Output	Time Spent (in seconds)	Importance
0	0	0	0.01	The first base case for the Fibonacci sequence.
1	1	1	0.01	The other base case of the Fibonacci sequence.
10	55	55	0.01	A starter value to check for accuracy.
40	102334155	102334155	0.01	The last value tested for the recursive version, to check the difference in speed.
100000	2.59 * 10^20898	2.6 * 10^20898	143.37	A very high number that helps checking the efficiency of the method and how high it could calculate within reasonable time.

fibListaInfinita

Test	Expected Output	Output	Time Spent (in seconds)	Importance
0	0	0	0.01	One of the basic results of the fibonacci sequence.
1	1	1	0.01	One of the basic results of the fibonacci sequence.
10	55	55	0.01	A higher but important result to check the accuracy.
100000	2.59 * 10^20898	2.6 * 10^20898	0.69	The last value tested for the normal list version, to check the difference in speed.
500000			13.21	A superbly high number to test only if the program still ran. As no values were found to compare it to, it is assumed to be correct.

BigNumber.hs Functions

scanner

Test	Expected Output	Output	Importance
"0"	(True,[0])	(True,[0])	Simple case to test if 0 is not eliminated.
"-1"	(False,[1])	(False,[1])	Important to test if the sign is being read correctly.
"200000"	(True,[2,0,0,0,0,0])	(True,[2,0,0,0,0,0])	Higher number to check if the list of digits is interpreted correctly.
"0050"	(True,[5,0])	(True,[5,0])	Zeroes on the beginning of the number are ignored.

output

Test	Expected Output	Output	Importance
(True,[0])	"0"	"0"	Base case that causes bugs sometimes.
(False,[0])	"0"	"0"	Although it is possible to declares a negative 0 in scanner, the result will always default to the "positive" version.
(True,[0,0,5])	"5"	"5"	Zeroes on the beginning of the number are ignored.
(False,[8,0])	"-80"	"-80"	Important to test if the bool is being interpreted as the negative sign.

somaBN

Test	Expected Output	Output	Importance
(True,[1]) (True,[0])	(True,[1])	(True,[l])	Case to check if the identity element is correct (0).
(True,[5]) (False,[7,0])	(False,[6,5])	(False,[6,5])	Check if the sign of the output is correct, contemplating the magnitudes of the numbers.
(False,[5]) (True,[7,0])	(True,[6,5])	(True,[6,5])	Check if the sign of the output is correct, contemplating the magnitudes of the numbers.
(False,[8,0]) (False,[2,0])	(False,[1,0,0])	(False,[1,0,0])	Test if the sum of two negative numbers returns an even smaller number.
(True,[9,9,9]) (True,[9,9,9])	(True,[1,9,9,8])	(True,[1,9,9,8])	Test if the sum of big digits like 9 and 9 would be a problem for the carry process.

subBN

Test	Expected Output	Output	Importance
(True, [0]) (True, [0])	(True,[0])	(True, [0])	Base case that checks for the identity element (0).
(True, [5]) (False, [7,0])	(True, [7,5])	(True, [7,5])	Check if the sign of the output is correct, contemplating the subtraction of a negative number (x - (-y) = x + y).
(False, [5]) (True, [7,0])	(False, [7,5])	(False, [7,5])	Check if the sign of the output is correct, contemplating the sign of the first number and the subtraction of a positive number.
(True, [1,8,4]) (True, [8,6])	(True, [9,8])	(True, [9,8])	Tricky example because it shows that the borrow list should be created along with the operation. At first, the subtraction of 8 by 8 wouldn't need to borrow 1 from the decimal house on the left, but since 4<6 and the operation starts on the right, we end up having to borrow 1 from 8 in 184 and our subtraction of 8 by 8 becomes 7 by 8. In this example, it becomes clear that mid-operation there might be a new need of borrowing 1 from the left.

mulBN

Test	Expected Output	Output	Importance
(True,[1]) (True,[0])	(True,[0])	(True,[0])	Base case that checks for the zero property of multiplication.
(True,[5]) (False,[7,0])	(False,[3,5,0])	(False,[3,5,0])	Check if the sign of the output is correct.
(False,[5]) (False,[7,0])	(True,[3,5,0])	(True,[3,5,0])	Check if the sign of the output is correct.
(True,[1,8,4]) (True,[1])	(True,[1,8,4])	(True,[1,8,4])	Check if identity property is maintained

divBN

Test	Expected Output	Output	Importance
(True,[0]) (True,[1, 0])	((True, [0]), (True, [0]))	((True, [0]), (True, [0]))	Case to check if the division of any number by 0 is still 0, with no remainder.
(True,[5]) (True,[7,0])	((True, [0]), (True, [5]))	((True, [0]), (True, [5]))	Check if the quotient and remainder are correct, when the divisor is larger than the dividend.
(True,[7, 0]) (True, [7,0])	((True, [1]), (True, [0]))	((True, [1]), (True, [0]))	Check if the division of a number for itself is correct.
(True,[8,2]) (True,[5])	((True, [1,6]), (True, [2]))	((True, [1,6]), (True, [2]))	Test if the quotient and remainder are correct in a regular division.

Functions

On Fib.hs

- fibRec: Determines the nth Fibonacci number through the traditional recursive approach.
- fibLista: Determines the nth Fibonacci number using a finite list to hold partial results of calculations and with a dynamic programming approach.
- fibListaInfinita: Determines the nth Fibonacci number using an infinite list and the same dynamic approach, but taking advantage of Haskell's lazy evaluation features.
- fibRecBN: Implements the fibRec function, with it's recursive approach, using exclusively BigNumbers, along with operations exclusive to BigNumbers.
- fibListaBN: Implements the fibLista function and its underlying approach using exclusively BigNumbers, along with operations unique to BigNumbers.
- fibListaInfinitaBN: Implements the fibListaInfinita function, along with it's lazy evaluation advantage, using exclusively BigNumbers and their operations.

On BigNumber.hs

- scanner: Converts a valid string into BigNumber format, taking into account the sign. Also, it eliminates some unnecessary left zeroes, in the case of not fully correct input.
- output: Converts a BigNumber into a string, taking into account it's sign as well as removing any zeroes on the left that may result from bad input.
- somaBN: Determines the sum of two BigNumbers with the help of somaBNAux. This function really only prepares the input for proper handling, reversing the list into a much easier to handle format, with units in the first place.
- somaBNAux: Helps with the calculation of the BigNumbers. Determines if it is easier to actually sum the numbers, or if it is more efficient to subtract them (like it happens in the case of opposite signs on the numbers).
- regularSum: Determines the sign of the final result, attending to the arithmetic properties of addition. Also, it determines the larger number so that the smaller one is padded with zeroes until the lists are of the same length, as a way to reduce the need to increase list lengths in the function where the calculations are done, and the larger one gets a left zero added, in case the result needs a new digit.
- regularSumAux: Computes the actual sum of the two numbers, in absolute values. As the sign is already handled by the regularSum function, this one only needs to worry about the digits of the numbers. It works by recursively adding the digits and the carry, storing the result in an accumulator, until the lists (which are of the same size) become empty.
- negativeSum: Passes control onto the subtraction functions, as according to arithmetic properties, the sum of two numbers with opposite signs is the same as subtracting the positive one for the absolute value of the negative one.
- subBN: Determines the sum of two BigNumbers with the help of subBNAux. Like somaBN, this only prepares the input for proper handling, reversing its digits so that units are at the start of the list.
- subBNAux: This function figures out if it is easier to perform an actual subtraction of the numbers or an addition, according to the arithmetic properties of subtraction.
- regularSub: This function is only called when the signs are the same, and so it determines the sign of the result based on which number is the largest. Also, it changes the order of the results so that the larger one is passed to the next function first, as a way to keep the calculations simpler. Finally, it pads the smaller number with zeroes so that it matches the length of the bigger one, much like what is done in regularSum.
- regularSubAux: Calculates the actual subtraction of the numbers, in absolute value. It assumes the first value passed is larger, so the functions calling it must handle it that happen. If all is well, computes the value much like in the regularSumAux function, the only difference being that instead of carrying the 1, we borrow from the next digit, with the result being held in an accumulator until the lists are empty.
- mulBN: Multiplies two BigNumbers. Much like the somaBN and subBN functions already seen, all it does is prepare the input for proper handling, reversing the digits lists.
- mulBNAux: Determines the sign of the resulting BigNumber, based on the sign of both operands, then passes them on into the actual multiplication part, in absolute value (forcing them to be true). This is one of two auxiliary functions that passes all the BigNumber instead of only its digits, as it uses the adding and subtracting functions described above as helpers.
- multiply: Adds together all the partial sums of the multiplication, giving the final result as an already properly formatted BigNumber.
- breakIntoParts: This function disassembles the multiplication into a series of sums. Each partial sum is the result of recursively adding the first number to itself x times, with x being the smallest digit at that time (since it works recursively, it will start with the units, then tens, then hundreds, etc...). Besides this, an offset of zeroes is added, corresponding to the place that the digit x occupies on the original number (for example, multiplying x by 342 is the same as x * 2 * 10^0 + x * 4 * 10^1 + x * 3 * 10^2).
- addPart: Adds a number to itself, n times. This is a helper function to the breakIntoParts, making the code more modular and allowing it to be used somewhere else, in case it is necessary.
- divBN: Divides two numbers, determining both the quotient and the remainder of the division. It, once again, uses an auxiliary function to determine this, only preparing the input for proper use.
- divBNAux: Computes the division, by subtracting the dividend to itself until the result is negative, whilst at the same time adding one to the quotient, and subtracting the divisor to the remainder, so that it keeps updating as the dividend diminishes.
- safeDivBN: This function operates much in the same way as the divBN function, but has an additional safety check for division with 0. To do this, it uses the Maybe Monad as a way to divert possible errors at runtime. When a division by 0 is attempted, the Nothing result is given, otherwise it can safely compute it, passing the control onto the divBN function.

On Utils.hs

- xor: Calculates the Exclusive or operation of mathematical logic.
- gt: Determines if the first BigNumber is bigger than the second, taking into account the sign.
- eq: Determines if two BigNumbers are equal, taking into account the sign.

- stuffZeroes: Creates a list of len zeroes.
- first: Returns the first element of a quadruple.
- second: Returns the second element of a quadruple.
- third: Returns the third element of a quadruple.
- fourth: Returns the fourth element of a quadruple.

Strategies

scanner

It was decided to separate negative and positive numbers, in order to parse properly the string when converting (through the function "map") each character into an integer. Also, by converting each character individually, the number can be arbitrarily large, as it will never cause an overflow. Finally, cases for both positive and negative numbers where added, so that the "-" sign could be parsed as well.

output

By checking if the first element of the tuple that constitutes our Big Number is true or false, we know its sign. After that, it is easier to convert it into a string, by placing '-' on the output before turning each element of the digits list into a string and concatenating them.

somaBN

This function uses four other auxiliary functions, so that all the operations that a normal addition incorporates are covered and the steps for the resolution are clear and separated.

The first important detail is reversing the list of digits, which is the job of the actual function called, somaBN.

Afterwards, we needed to distinguish the operations through the signs and later through the magnitudes of the operands, in order to determine beforehand the sign of the result, as well as the type of operation required. In short, the functions treats a sum between two positive or two negative BigNumbers as a regular positive sum, changing the sign accordingly and lastly, the ones with opposite signs as simple subtractions, with the arguments in the correct order, as for example (-x) + (+y) = (+y) - (+x).

The final preparation for the addition itself is to have in mind the occasions in which adding two digits would have a result greater than 9 and the possibility of one of the numbers being smaller in length, hence the padding added in the regularSum function.

Finally, the result is calculated recursively, first summing the units (and the carry, which at the start is always 0) and checking if the result is greater than 10. If not, then we add them to the final result. If it is, then a carry of 1 is added to the next sum, and the result stored is the (sum - 10). Then, we do the same for the next digits, recursively, until none are left, point at which the function returns.

subBN

This function uses four other auxiliary functions as well, and has a very similar structure to somaBN, since both operations are the inverse of each other.

The first important detail is, like in the sum one, reversing the list of digits, which is the job of the actual function called, subBN.

After that, we once again determine the operations through the signs and the magnitudes of the operands, in order to determine beforehand the sign of the result, as well as the type of operation required, according to arithmetic rules and much like in somaBN.

After that, and in the case of a regular subtraction, the smaller number is also padded, but this time the larger is not, since the result will never be bigger than the larger number.

Finally, the result is calculated recursively, first subtracting the units from each other (and subtracting the carry from the first unit) and checking if the result is negative. If not, then we add the subtraction to the final result. If it is, then we "borrow" 1 from the tens places, which means that it will have one subtracted from it, and the result stored in the final answer is (sub + 10). Then, we do the same for the next digits, recursively, until none are left, point at which the function returns.

mulBN

To multiply two BigNumbers, a simple heuristic was used.

The normal way to see a multiplication of x * y is that all that is done is add x onto itself y times. But, representing y as its digits (y1y2y3). multiplication is also doing x * y3 + x * y2 * 10 + x * y1 * 100, or, more generally, for the case of y = y_n y_n-1 y_n-2 ... y1 y0, x * y = sum(x * y_i * 10^i), i <- [0..n].

As such, it is possible to represent multiplication as the sum of partial, easier multiplications that can be done in the "traditional" way of adding to itself, with a lot less overhead. In fact, it does in the worst case 10 * log_10(n) + 1 sums, much better than the n sums required with the traditional method.

To do this, first the lists are reversed so that the units come first, like in the previous functions. Then, the sign is predetermined based on the sign of the operands, which, in the case of multiplication, is just checking if they are the same or not.

Then, to multiply, one of the numbers (here, it is the first one), is "fixed". This means that it will not change throughout the operation, and is the one that will be added to itself. The other number, is then, in the aptly named breakIntoParts function, broken into each digit, and another function (addPart) is called which takes the first digit n (in the beginning, the units one) and sums the fixed number n times to itself using the somaBN function (as a way to force the correct orientation of the lists of digits).

After that, the partial result is added to the list. But, before it can be added, it needs to have an offset, as we only did the x * y_i part. To do this, we check the size of the results list, which gives us all we need. If the list is empty, we are doing the partial sum of the units place and the offset of zeroes is 0. If it is the tens, the list will have size 1 and we will have an offset of 1 0, and so on. With this, there is no need to store the position of the digit relative to the original number y, as we can figure it out from the amount of results already calculated.

Finally, when the non fixed BigNumber is depleted, all that is necessary to do is sum all of the partial results, which is taken care of in the multiply function, using the somaBN one as a helper. This final result is then the multiplication of both numbers, with the list in the proper format.

divBN

The function used to divide two numbers uses a very simple strategy to achieve its purpose.

All it does is start with the values unaltered (as we need them in the original order to use the somaBN and subBN functions), as well as 0 (representing the initial quotient) and a copy of the dividend, representing the initial remainder, all stored in a quadruple.

Then, the function successively subtracts the divisor to the remainder , until the result turns negative. At each iteration of this subtraction, the value of the quotient is incremented by 1, which means an increase in the quotient. When the result effectively turns negative. The function halts, the value held by the quotient is the actual quotient, and the final result before the last subtraction is the remainder of the division.

Question 4:

- **Function fibRec :: Int -> Int** accepts up to value 40 with output 102334155, 207.78 seconds elapsed and usage of 97,131,015,400 bytes. Due to a huge time overhead, no larger values where tested.
- **Function fibLista :: Int -> Int** accepts up to value 92 with output 754011804746346429, 0.01 seconds elapsed, usage of 132,968 bytes and no overflow. All inputs greater than 92 cause overflow.
- **Function fibListaInfinita :: Int -> Int** accepts up to value 92 with output 7540113804746346429, 0.00 seconds elapsed, usage of 83,416 bytes and no overflow. All inputs greater than 92 cause overflow.
- **Function fibRec :: Integer -> Integer** accepts up to value 40 with output 102334155, 367.86 seconds elapsed and usage of 96,119,078,984 bytes.
- **Function fibLista :: Integer -> Integer** accepts up to value 100000 with output 2.6*10^20898, 257.73 seconds elapsed and usage of 523,370,656 bytes. No overflow with the values tested, but the greater they are, the longer it takes for the function to calculate the result.
- **Function fibListaInfinita :: Integer -> Integer** accepts up to value 100000 with output 2.6*10^20898, 0.53 seconds elapsed and usage of 474,183,824 bytes.
- **Function fibRecBN :: BigNumber -> BigNumber** accepts up to value 35 with output (True,[9,2,2,7,4,6,5]), 877.50 seconds elapsed and usage of 189,399,738,336 bytes.
- **Function fibListaBN :: BigNumber -> BigNumber** accepts up to value 10000 with output

(True,
[3,3,6,4,4,7,6,4,8,7,6,4,3,1,7,8,3,2,6,6,6,2,1,6,1,2,0,0,5,1,0,7,5,4,3,3,1,0,3,0,2,1,4,8,4,6,0,6,8,0,0,6,3,9,0,6,5,6,4,7,6,9,9,7,4,6,8,0,0,8,1,4,4,2,1,6,6,6,6,2,3,6,8,1,5,5,5,9,5,5,1,3,6,3,3,7,3,4,0,2,5,5,8,2,0,6,5,3,3,2,6,8,0,8,3,6,1,5,9,3,7,3,7,3,4,7,9,0,4,8,3,8,6,5,2,6,8,2,6,3,0,4,0,8,9,2,4,6,3,0,5,6,4,3,1,8,8

258.22 seconds elapsed and usage of 624,753,842,200 bytes.

- **Function fibListaInfinitaBN :: BigNumber -> BigNumber** accepts up to value 10000 with output

(True,
[3,3,6,4,4,7,6,4,8,7,6,4,3,1,7,8,3,2,6,6,6,2,1,6,1,2,0,0,5,1,0,7,5,4,3,3,1,0,3,0,2,1,4,8,4,6,0,6,8,0,0,6,3,9,0,6,5,6,4,7,6,9,9,7,4,6,8,0,0,8,1,4,4,2,1,6,6,6,6,2,3,6,8,1,5,5,5,9,5,5,1,3,6,3,3,7,3,4,0,2,5,5,8,2,0,6,5,3,3,2,6,8,0,8,3,6,1,5,9,3,7,3,7,3,4,7,9,0,4,8,3,8,6,5,2,6,8,2,6,3,0,4,0,8,9,2,4,6,3,0,5,6,4,3,1,8,8

268.44 seconds elapsed and usage of 621,970,070,488 bytes.

Conclusion: For fibRec and fibRecBN, it is always hard to calculate bigger numbers since it needs a lot of resources (in terms of byte usage) and time to get to the result.
For Int values on fibLista and fibListaInfinita, it is possible to observe overflow, so it can't calculate numbers bigger than 92. Although the algorithm is very efficient, when the cast of the parameters of the function is Int, the utility is limited. To calculate bigger numbers, it is surprising to see that fibLista and fibListaInfinita (Integer -> Integer) are more efficient than fibListaBN and fibListaInfinitaBN, since the functions envolved in operations between BigNumbers are more complex. That way it takes more resource and time for fibListaBN and fibListaInfinitaBN to get to a result that fibLista and fibListaInfinita can achieve without overflowing.

Project Authors

This project was developed by:

- Carolina Figueira up201906845
- Tiago Rodrigues up201907021