Implementation of arbitrary precision arithmetic -Big Integer

Submitted by **Group 12**

- Anirudh K V
- Malini K Bhaskaran
- Neha Nirmala Srinivas
- Saumya Ann George

Executive summary

This project implements arithmetic operations for Integers which cannot be stored in normal integer data types. Our motivation is to do this implementation effectively which can help to do big Integer arithmetic and learn about how JAVA Big Integer is handling these operations effectively.

Problem statement

Develop a program that implements arithmetic with large integers, of arbitrary size.

Pseudocode

Base is hardcoded as 100.

Add: Addition is performed using the function "add"

Parameters: BigNum num1 -First number in the addition pair

BigNum num2 -Second number in the addition pair

BigNum res -Result list after performing addition

Algorithm: Uses the traditional algorithm for calculating addition using node wise addition and passing carry.

If both numbers are having different sign, then performs subtraction and assigns the sign of greater number.

Subtraction: Performed using the function "subtract"

Parameters: BigNum num1 - Bigger number in the subtraction pair

BigNum num2 - Smaller number in the subtraction pair

BigNum res - Result of subtraction

Algorithm: Uses the traditional algorithm for calculating subtraction using node wise subtraction and passing borrow.

SubtractUtil: Compares whether the first number is greater than the second number and if not then the "subtract" function is called with greater number as the first parameter and negative symbol added to the first number in the list. Also performs checks for negative numbers and calls corresponding functions.

Input: BigNum number1

BigNum number2

Output: BigNum result after subtraction

Calls remove Leading Zeros.

Algorithm: Decides which functions to call based on the sign of the numbers.

RemoveLeadingZero:

Function to remove leading zeros from the subtraction result.

Removes all zeros except the one at most significant digit if the result is zero

Input: The result value BigNum passed from the Subtract function

Output: List after removing the leading zeros

Product

Input: 2 numbers in BigNum format for calculating product

Output: Result of product in BigNum representation

Implemented traditional Multiplication algorithm.

First performs a check with 0 to check whether one number is 0, if it is then the result is stored as 0.

If both elements are non-zero then performs the multiplication using traditional multiplication algorithm.

toString()

This function converts the List representation of the number to the string representation.

Works on BigNum object.

Returns String representation of BigNumber.

printList()

This function works on BigNumber object.

Prints the internal list representation with the base information.

Factorial

Finds the factorial for the BigNum passed as input to the function.

Calculates the product of numbers up to the input iteratively.

Power(BigNum x, Long n)

Input: Bignum x

Long n

Result: Passes the result of the power operation.

Algorithm: Uses the recursive method for calculating power. Calls product method inside the power function.

<u>Power(BigNum x, BigNum y)</u>

Input: BigNum x – value for which power has to be performed

BigNum y – Power value

Output: Returns result in BigNumber format.

Algorithm: Uses recursive call to calculate the power. Uses ShiftLeft operation and calculates power. :

<u>Division:</u>

Input: two long numbers

Output: quotient of numerator divided by denominator.

We compute the quotient by using the following technique. If the numerator is equal to the denominator we return the quotient as 1. If the denominator is greater than the numerator return 0. If the denominator is zero, prompt that division by zero cannot be performed and return. Otherwise continue to find the quotient using Binary Search technique. Compute the midpoint between 1 and the numerator initially. If the product of denominator and midpoint is equal to the numerator or the difference between the numerator and product is less than the denominator the result will be the quotient. Else if the product is greater than the numerator we search in the lower half, else the upper half. We continue till the condition is satisfied.

<u>Mod</u>

Input: Two Big numbers

Output: modulus of numerator and denominator.

We compute the modulus in the following manner: Compute quotient of numerator when divided by denominator. Compute the product of quotient and denominator. Modulo will be the numerator minus the product.

<u>SquareRoot</u>

Input: BigNumber x – For which square root has to be performed. The midpoint value is calculated by calling "divideBy2" function.

Checks for perfect square and non-perfect square is implemented.

Using Binary search method, square root is calculated.

<u>Compare</u>

Compares two BigNum values and returns 0, 1 and 2 accordingly.

- If the first number is less than the second number return 0.

- If the first number is greater than the second number the function returns 1.
- If both numbers are equal, return 2.

<u>CheckGreaterNum</u>

Calls compare function if both values are of same sign.

Otherwise, perform checks for signs and determine which is greater and return.

<u>BigNumDriver</u>

This program handles the input parsing and calling all the corresponding BigNum class methods and printing output accordingly.

Test results

Sample Outputs

1.

Output for the sample driver program given: (Level 1)

```
a = 1234567890123456789012345678901234567890
b = 999
c = a+b = 1234567890123456789012345678901234568889
a+b-a = 999
a*c =
152415787532388367504953515625666819573386677799586953101083523842220835237421
0
a*0 = 0
2^1025 =
359538626972463181545861038157804946723595395788461314546860162315465351611001
926265416954644815072042240227759742786715317579537628833244985694861278948248
755535786849730970552604439202492188238906165904170011537676301364684925762947
826221081654474326701021369172596479894491876959432609670712659248448274432
Internal representation:
base:100 32 44 27 48 84 24 59 26 71 70 96 60 32 94 95 76 18 49 94 98 47 96 25
17 69 13 2 1 67 32 74 44 65 81 10 22 26 78 94 62 57 92 84 46 36 1 63 67 37 15
1 70 41 90 65 61 90 38 82 18 92 24 20 39 44 60 52 5 97 30 97 84 86 57 53 55
87 24 48 89 27 61 48 69 85 49 24 33 88 62 37 95 57 17 53 71 86 27 74 59 77 22
40 22 4 72 50 81 44 46 95 16 54 26 26 19 0 11 16 35 65 54 31 62 1 86 46 45 31
61 84 78 95 53 59 23 67 94 4 78 15 38 10 86 45 15 18 63 24 97 26 86 53 59 3
```

2.

Testing Square Root Performance

Number1:

8888881245783930493473948394374958492390859435743850758493483765432123456789876543212345678909876543212345678987654321234567887654321234567890 0

Result of Square root is:
2981422688211775598695165542755181228911285818860653308820301675095726 59

Time for square root = **273ms**

3. Testing Mod function performance

Number1:

98765432123456789098765444444444444444444444444444444444444444444445678900000000000000000000000000000000000000098765432123456789876543223456789098765434 5678

Result of mod is:
6191865599209876533846915478313104107775152563284999469015964303800583808148289735 60823877640

Time for Mod = **319ms**

4.

Performance evaluation of division

Number1:

09876543212345678900000000000000000000000000000000000000000000000000000000000000000000000000000000000000987654321111111111111111111111111111111111111111111111111234567

Number2:

1234567786756453423456374895607968756453423000000000000000000000000000000000000000 00000

Result of div is:
8000007438182506972404823997866520304845292254502133694242341138588445860829 6

Time for div = **328ms**

5. Performance evaluation of Product

Number1:

9876543212345678909876543234567890987654321234567890123456789098765432123456789987654322345678987654321234567890876543212345678909876543234567

Number2:

999999999999999999999000000000000000000000000000000000222222222222222222222222222255 5555555555555555555

Result of product is:
9876543212345678909866666913555453087444446913333222544170085827434840330589848342

20829068013693244437807962964944949304938161749389074304526054732557901920962919142
963415629621399024773661848397187929492318210019471887515474622827846364869685

Time for product = **6ms**

6. Performance evaluation of Addition

Number1:

99999999999999999999999999999555555555123456789098765432123456789087654321234567 8
9087654323456788765432123456786543456789987654322345678987654323456789

Number2:

87654322345678987654321234567890000000000000000000000000000000000000000000000000
00000000000000000000000000000000000987654321234567887654323

Result of addition is:
1000000000087654322345678987654276790123402345678909876543212345678908765432123456 7
8908765432345678876543212345678654345678998765531000000022222221111112

Time for addition =**0ms**

7. Performance evaluation of Subtraction:

Number1:

-
98765432123456789876543212345678987654321234567898765432123456789765432123456789087
3123456789842345678998765432234567899543223456 7

Number2:

9876543212345678987654321234567898765432345678987654323 4

Result:

-
98765432123456789876543212345678987654321234567898765432123456789765432124444443408
546913577749666691356666419766691357853087778 01

Time for Subtraction =**1ms**

8. **Performance evaluation of Power**

Number1:

2

Number2:

1025

Result of power is:
3595386269724631815458610381578049467235953957884613145468601623154653516110019262
6541695464481507204224022775974278671531757953762883324498569486127894824875553578
6849730970552604439202492188238906165904170011537676301364684925762947826221081654
4743267010213691725964798944918769594326096707126592484482744 32

Time for power -**15ms**

9. **Performance evaluation for factorial**

Number1:

1025

Result of factorial is:
555399201596032871515384499330095110645159281262646378059494202745029087737521101882
646950873961995138705616816851705272780294019914442052466965486667064987586076716213
084118614012499829861338444899572676155341645997381365008844614370028318411693632494
019206385012516539387056035941154643957091840647075946057493456350214268350584753613
005462690113400151316876215523571193592741519445269308518233187066646307423301831735
802578669983406647452698848692497866829470500032672913380216738076335941366135552107
116260282144063412871785339645922504773858859175049117906076196232140268609967638775
097211625513090953980427925873053256264809116849716899791555265527716487955825982320
031636909660498686351893087947558554259997602335338461835166205862203478903042732569
180275155595406320482511175461222554129420474952198603840176708932238412066343172610
707256281798486519920468585730745408710972996326892694967934905931018591787976427612
315769477417309952891003511795373548831234418807283333746709721003731627784775830167
397022463807530985272057873264177079050428316930765688846923468315654859595551366767
619460103161547194980479754678164394058728676295754353583675409697463030122654048354
101465536608477197105445400872539878500362156718512646379782169425871295807350161688
603443356393283422285523111094787112656076027774126548414085758092931527906672197274
195817628759428251498835890782743239556382151241831967107131981997267396937379069572
252786357818720141790358604019764998521484630075270576994306547673717331609661989209
592696745626714782149207663538859163345375482955857268837394881290730337575142978393
054380423099891567452696666675065700941068731963873783208971349465340507540223031424
373057915206431095181862390578153125749250055303622254343222219762914025822238945532
407967298071278496696840977344475911230829590299861426149877257726691111384410326331
599658135060725854079725506719808014481253337401156717910838952733137566262225806409
733959803418926043958965986461843976828833825239160610695381810305626755469170436326
469851446958040967810601538155031110679196066088300054012410245772266137767722601894
482364556468413447168071202138126775165880052217191862153168411514416309725217265241
14722718989398537654442844711472459785698311916594371409 9

38837876925171668941276511882070768125888452470333130991385415441093098609269330659
91734747559935810228510418908871698095698273054667445462819143680000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000

Time for factorial =**213ms**

Discussion of results

We have implemented level 1 and level 2 of the project. Subtraction, addition, product, division and factorial is implemented for negative numbers and positive numbers.

If factorial is called for negative numbers the result will be 0 and output the message factorial cannot be called for negative numbers.

Average running time for test cases above can be summarized as follows.

Operation	Running Time(ms)
Addition	0
Subtraction	1
Product	6
Factorial	213
Power	15
Division	328
Mod	319
Square Root	273

Conclusion

We can use this program for the arbitrary precision integer arithmetic. Tested for the sample inputs given and all functions are working correctly with good performance.

References

Javas BigInteger Library

http://developer.classpath.org/doc/java/math/BigInteger-source.html