Homework 3

1. Algebraic operations

(30 points) The purpose of this exercise is to practice writing and calling simple functions. You will develop a little world in which the basic algebraic operations are expressed as *functions*:

- $a + b$ is realized as `plus(a,b)`
- $a - b$ is realized as `minus(a,b)`
- $a * b$ is realized as `times(a,b)`
- $a \wedge b$ is realized as `pow(a,b)`
- the integer part of a / b is realized as `div(a,b)`
- the modulo $a \% b$ is realized as `mod(a,b)`
- the integer part of $\sqrt{x}$ is realized as `sqrt(a)`

Since all these functions operate on `int` values and return `int` values, they can be easily *composed*. For example, the expression `2 * (4 + 3)` can be realized as `times(2,plus(4,3))`.

Your task is to implement all the functions shown above, without using the Java operators `+`, `-`, `*`, `/`, `%`, and without using Java's functions `Math.pow` and `Math.sqrt`.

The only algebraic operations that you are allowed to use are `++` (add 1), `--` (subtract 1), `<`, `<=`, `>`, `>=`, `==`, and `!=`. You can also use any other Java element that we learned, like `while` and `for`.

Inspect the given `Algebra.java` class, and implement all the functions. You are welcome to add more tests to the `main` function, if you want.

Implementation tips

(1) To compute `a + b`, we can add `1` to `a`, `b` times. Implementing minus is a similar idea.

(2) When writing a function, always try to do it using other functions that you've already implemented. For example, `times` can be implemented using `plus`.

(3) Implement the functions in the order in which they appear in the class.

(4) In this exercise we don't worry about efficiency. We only care about elegance.

2. Loan calculations

(30 points) Suppose you take a loan of 100,000 Shekels at an annual interest rate of 5%, for 10 years, with equal annual payments. How much should you pay each year, so that at the end of the 10^{th} year the loan will be fully paid? For example, suppose that the annual payment is 10,000 Shekels. In this case, your balance (the sum that you still owe) at the end of the first year will be (100,000 – 10,000) * 1.05 = 94,500. At the end of the second year, the balance will be (94,500 – 10,000) * 1.05 = 88,725. In general, if you make n equal payments, we can make n such calculations, and check the balance at the end of the n-th year. This model is implemented in the following spreadsheet:

	A	B	C	D	E	F
1	Loan:	100000				
2	Interest rate:	5				
3	Periods:	10				
4	Periodical payment:	10000		Change this value		
5						
6		Ending balance				
7	Period 0	100000				
8	Period 1	94500				
9	Period 2	88725				
10	Period 3	82661				
11	Period 4	76294				
12	Period 5	69609				
13	Period 6	62589				
14	Period 7	55219				
15	Period 8	47480				
16	Period 9	39354				
17	Period 10	30822		And observe the impact on this value		
18						

If the ending balance is positive (meaning that you still owe money), it implies that the periodical payment is too low. If the ending balance is negative, it implies that you paid too much.

Getting started: Start by playing with the given spreadsheet: Load it into Excel, or into Google Sheets, and experiment with various periodical payment values, until the ending balance will be close to zero (say, a few Shekels). The periodical payment that brings the ending balance close to zero is the problem's solution.

Why do we use spreadsheet modeling for doing this calculation? Isn't there some *financial formula* that, given the loan amount, the number of periods, and the interest rate, gives the periodical payment that settles this loan? The short answer is that, yes, there is such a formula. But, building a spreadsheet model and using a trial-and-error method to solve it gives a good understanding of the problem. Also, there are many complex financial and scientific problems that *cannot* be solved using a formula, and must be solved instead using simulation, or some spreadsheet model.

The `LoanCalc` program gets three inputs: a loan amount, an interest rate, and number of payments. It then computes the periodical payment that pays out the given loan. This value is computed using two alternative algorithms: *Brute force* search, and *bi-section* search. Notice that for each function that you have to write we provide the function signature, as well as a default return value. The result is a complete, executable, skeleton of all the code that has to be developed. This is an example of good software engineering: The system architect builds the program skeleton, and the developer can run and test the program before fully implementing it.

Computing the ending balance: Start by implementing the `endBalance(loan, rate, n, payment)` function.

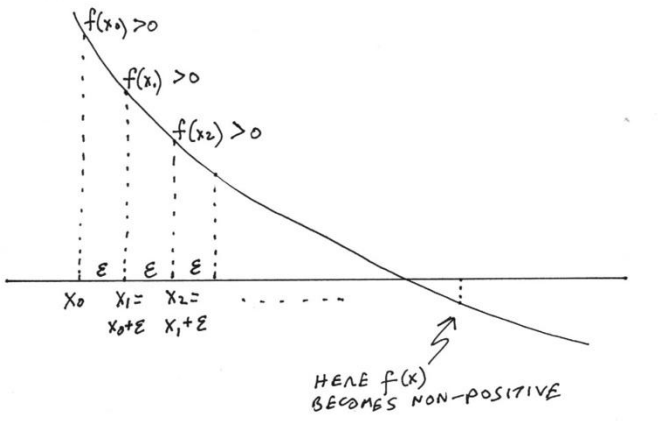
Implementation tip: Use a loop to carry out the same computation done by the spreadsheet. Test your implementation by evaluating this function on several possible payment values, and compare the returned values to those computed by the spreadsheet.

Moving along, how to compute the periodical payment that brings the loan's ending balance close to zero? Formally, we wish to find x such that $f_{loan,rate,n}(x) = 0$, where f is the

`endBalance` function, *`loan`* is the initial loan sum, *`rate`* is the interest rate, *`n`* is the number of payments, and *x* is the periodical payment. We treat the first three values as fixed parameters, so *x* is the only variable of this function. The goal is to find an *x* value for which the function is close to 0.

Note that f is monotonically decreasing in *x*: As *x* increases, f decreases: The more you pay each year, the lower will be your ending balance. As we learned in lecture 3-1, the solution of monotonic functions can be approximated using brute force search, and bisection search.

Brute force search: We start with an initial value g, for which we know that $f(g) > 0$. Then, we set g to $g + \epsilon$, where ϵ is a small value, and check if $f(g) > 0$. We repeat this process until $f(g)$ becomes non-positive. At this point we return g, which will be an approximation of the correct solution. How good an approximation? The correct solution will be somewhere in the interval $[g - \epsilon, g + \epsilon]$. So, the smaller is ϵ, the better will be the approximation. In the following image, the value of g in iteration *i* is represented as x_i:



Implement the `bruteForceSolver` function.

Implementation tips:

- In this application, *f* is the `endBalance` function.
- Since the function computes the ending balance of an *n*-period loan, it is reasonable to set the initial guess of the brute force to $g = loan/n$. Why? Because if in each period we pay
  loan / *n*, it means that we are not paying interest. Therefore, the ending balance will surely be positive, i.e. $f(loan/n) > 0$
- Keep track of the number of iterations by incrementing the static variable `iterationCounter` in each iteration (and make sure to set it to 0 before starting the search).

Bisection search: As we learned in lecture 3-1, the solution of a monotonic function can be found using an elegant and efficient algorithm – *bisection search*. Here is a version of this algorithm, adapted to this application:

```
// Sets L and H to initial values such that f(L) > 0, f(H) < 0,
// implying that the function evaluates to zero somewhere between L and H.
// So, let's assume that L and H were set to such initial values.
// Set g to (L + H)/2
while (H − L) > ϵ {
    // Sets L and H for the next iteration
    if f(g) · f(L) > 0
        // the solution must be between g and H
        //  so set L or H accordingly
    else
        // the solution must be between L and g
        //  so set L or H accordingly
    // Computes the mid-value (g) for the next iteration
}
return g
```

(In lecture 3-1, we used the notation *M*, for "middle", instead of g)

Implement the `bisectionSolver` function.

Implementation tips:

- Use the algorithm described above. Note that the algorithm is missing some details. See the bi-section algorithm presented in lecture 3-1, and complete the algorithm accordingly.
- As before, *f* is the `endBalance` function.
- Choose initial *lo* and *hi* values, using similar considerations to what we did in the brute force search.
- Keep track of the number of iterations by incrementing the static variable `iterationCounter` in each iteration (and make sure to set it to 0 before starting the search).

3. Anagram

(40 points) An *anagram* is a word or a phrase formed by rearranging the letters of a different word or phrase, using every original letter exactly once. For example, the word "`listen`" can be rearranged into "`silent`". As we did with palindromes, we disregard spaces, punctuation marks, and upper/lower case letters. For example, "`anagram`" and "`Nag a Ram`" are anagrams.

Inspect the given `Anagram.java` class, and implement all its functions.

Implementation notes:

1. Start by reading the `main` function. Make sure that you understand all the tests.

2. Implement the `preProcess` function, and test it. You can add testing code to the `main` function, as you see fit.

3. Implement the `isAnagram` function. Start by pre-processing the two strings. Then check if the two resulting strings form an anagram. Tip: Use a nested loop for iterating over all the characters of both strings.

4. Implement `randomAnagram`. Note that this function is not supposed to return a word or phrase in the English language. Rather, it should return some random permutation of the characters in the given string. For example, a random anagram of the string "`java`" may be, say, "`ajva`".
Tip: One way to implement this function is to use a loop that draws a random character from the string and then deletes the selected character from the string (so that we will not select it again).

Submission guidelines

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines.

Submit the following files only:

- `Algebra.java`
- `LoanCalc.java`
- `Anagram.java`

Submit on Github.