Homework 4

1. Loan calculations

Suppose you take a loan of 100,000 ILS (שקלים) 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-year period the loan will be fully paid? For example, suppose that the annual payment is 10,000 ILS. 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 ILS. At the end of the second year, the balance will be (94,500 – 10,000) * 1.05 = 88,725 ILS. In general, if you have to make *n* equal payments, we can make *n* such calculations, and check the balance at the end of the *n*-year loan period. If the ending balance is positive (meaning that you still owe money), it implies that we should have paid more each year. If the ending balance is negative, it implies that we paid too much. This logic is illustrated in the following spreadsheet:

Loan:	100000	
Interest rate:	0.05	
Periods:	10	
Periodical payment:	**10000**	Change this value
	Ending balance	
Period 0	100000	
Period 1	94500	
Period 2	88725	
Period 3	82661	
Period 4	76294	
Period 5	69609	
Period 6	62589	
Period 7	55219	
Period 8	47480	
Period 9	39354	
Period 10	**30822**	And observe the impact on this value

Start by playing with this spreadsheet, which is supplied with the homework. Change the value of the periodical payment, and observe the impact on the last period's ending balance. Using trial and error, try to come up with a periodical payment that brings the ending balance close to 0.

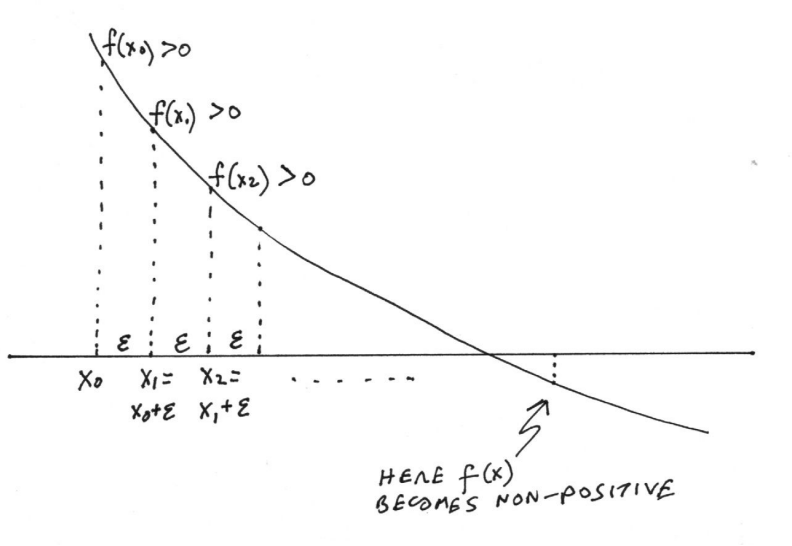
Inspect the spreadsheet formulas that compute the periodical ending balances, and make sure that you understand the model. If you are not familiar with spreadsheet modeling, this is a good opportunity to become familiar with this important tool.

A. Implement the function `endBalance(loan, rate, n, payment)` in the supplied `LoanCalc.java` class. Implementation tip: Use a loop to carry out the same computation done by the spreadsheet. Test your implementation by evaluating this function on several different `payment` values, and compare the returned values to those computed by the spreadsheet.

The key question in this exercise is as follows: How to compute the periodical payment that will bring the loan's ending balance close to zero? Formally, we wish to solve $f_{loan, rate, n}(x) = 0$, where *f* is the `endBalance` function, *loan* is the initial loan sum, *rate* is the interest rate, and *n* is the number of payments. The only variable is *x*, the annual payment. The goal is to find an *x* value for which the function is close to 0. In other words, find an *x* value such that $f(x) \sim 0$.

Note that *f* is monotonically decreasing in *x*: As *x* increases, $f(x)$ decreases, the more you pay each year, the lower will be your ending balance. The solution of monotonic functions can be approximated in several different ways, of which we'll focus on two: *brute force search*, and *bisection search*.

Brute force search: we wish to compute *x* such that $f(x) = 0$, where *f* is a monotonically decreasing function. We can start with some initial value x_0 for which we know that $f(x_0) > 0$. Then, we can add a small value, say ε, to *x*, obtaining a value that we call x_1. We can then compute the value of $f(x_1)$, and repeat the process $x_t = x_{t-1} + \varepsilon$ until $f(x_t)$ becomes non-positive. At this point we can return x_t, which will be an ε-approximation of the correct solution.

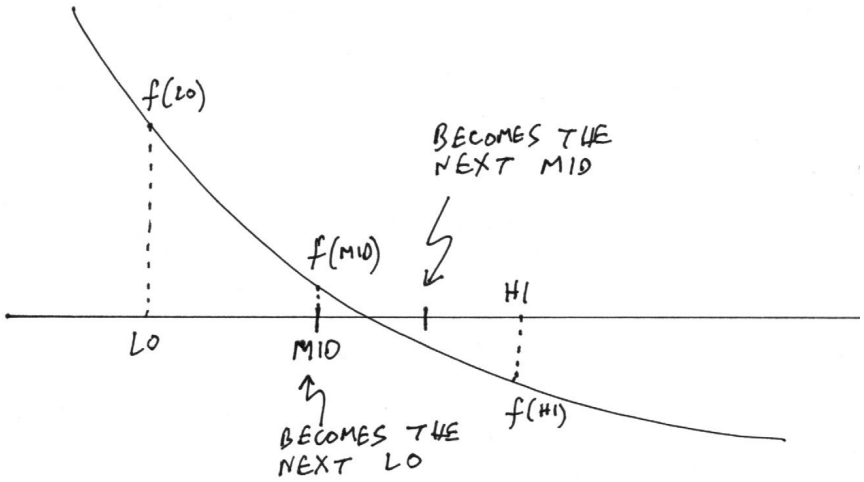


B. Implement the `solveByBruteForceSearch` function, designed to find *x* such that $f(x) \sim 0$ by the iterative process just described. Implementation tips:

- *f* is implemented by the `endBalance` function.
- Since the function computes the ending balance of an *n*-period loan, it is reasonable to start the search for a solution with $x = loan / n$. Why? Because if in each period we pay $loan / n$, it means that we are paying no interest. Therefore, the ending balance will surely be positive (in other words, we know that $f(loan/n) > 0$, as required by the algorithm, so $loan / n$ can serve as a starting point).
- Implement the function using a loop that starts with this value, computes *f*, and increases *x* iteratively, until $f(x)$ becomes non-positive.
- We wish to keep track of how many iterations the algorithm used. Do this by incrementing the static variable `iterationCounter` in each iteration.

Bisection search: A significantly more efficient algorithm for finding a good approximation of x such that $f(x)$~0 can be described as follows:

```
// Assumption: f(x) is a monotonic and continuous function
// Select initial lo and hi values such that f(lo) · f(hi) < 0
// (implying that the function becomes zero somewhere between lo and hi)
mid = (lo + hi) / 2
while (hi − lo) / 2 > ε {
    // Sets lo and hi for the next iteration
    if f(lo) · f(mid) > 0
        lo = mid    // the solution must be between mid and hi, so set the next lo to the current mid
    else
        hi = mid    // the solution must be between lo and mid, so set the next hi to the current mid
    // Computes the mid-value for the next iteration
    mid = (lo + hi) / 2
}
return lo    // lo (or hi) provide an ε-approximation of the correct solution.
```



C. Implement the function `solveByBisectionSearch`, designed to solve $f(x) = 0$ using the above algorithm. Implementation tips:

- f is implemented by the `endBalance` function.
- Choose initial *lo* and *hi* values, using similar considerations to what we did in the brute force search.
- We wish to keep track of how many iterations the algorithm used. Do this by incrementing the static variable `iterationCounter` in each iteration.

2. Prime numbers

A prime number is a number greater than 1 which is only divisible by 1 and by itself. There is an infinite number of prime numbers, and here are the first few of them: 2, 3, 5, 7, 11, 13, 17, 19, ... Write a program `Primes` that reads a command-line integer *N*, and prints all the prime numbers up to *N*. Your program should implement the *Eratosthenes algorithm* presented in lecture 4-2.

Here is an example of the program's execution:

```
% java Primes 40
The prime numbers up to 40 are:
2
3
5
7
11
13
17
19
23
29
31
37
```

Implementation note: When implementing the algorithm, you can stop the "skipping process" at $\sqrt{N}$. The reason is related to the following observation: if $x = a \times b$, then $a \leq \sqrt{x} \leq b$.

3. Letter frequencies

Many text analysis programs need to count how many times the letters of the alphabet appear in a given text. The `LetterFreq` program does just that, and shows the results using a histogram. The program can be executed in two ways. If the user provides a command-line argument, the program treats it as the text to analyze. For example:

```
% java LetterFreq "As Easy as A, B, C"

a:****
b:*
c:*
d:
e:*
f:
...   (the program actually prints 26 lines, one per letter, but we skip a few to save space)
r:
s:***
t:
...
x:
y:*
z:
```

Alternatively, if the user supplies a text file (instead of a command-line argument), the program performs the same analysis on the text in the file. For example:

```
% more EnglishText.txt        // the "more" command can be used to list the contents of a file

Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of
the Galaxy lies a small unregarded yellow sun. Orbiting this at a distance of roughly
ninety-two million miles is an utterly insignificant little blue green planet whose
ape-descended life forms are so amazingly primitive that they still think digital watches
are a pretty neat idea. This planet has - or rather had - a problem, which was this: most
of the people on it were unhappy for pretty much of the time. Many solutions were suggested
for this problem, but most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't the small green pieces
of paper that were unhappy. And so the problem remained; lots of the people were mean, and
most of them were miserable, even the ones with digital watches. Many were increasingly of
the opinion that they'd all made a big mistake in coming down from the trees in the first
place. And some said that even the trees had been a bad move, and that no one should ever
have left the oceans.

% java LetterFreq < EnglishText.txt

a:************************************************
b:***************
c:*******************
d:*********************************
e:*****************************************************************************************************
f:**************
g:********
h:***************************************
i:************************************************
j:**
k:***
l:****************
m:*****************
n:************************************************
o:*************************************************************
p:******************
q:****
r:*****************************************
s:****************************************
t:*************************************************************************
u:*************************
v:*******
w:************
x:*
y:**************
z:
```

Complete the supplied `LetterFreq.java` program.

We recommend to first test the program with small strings, using the command-line argument execution option. You can then switch to testing the program on the supplied `EnglishText.txt` file, or use some other texts of your own.

4. Encryption

In cryptography, "cyclic shift encoding" is a simple encryption technique in which each letter in a given text is replaced by a letter some fixed number of positions down the alphabet. The fixed offset is called *key*. Without loss of generality, let's assume that we shift to the right, cyclically. To illustrate, suppose we have an English alphabet consisting of 5 letters only (instead of 26), and we use *key* = 2. This implies that the letters will be shifted as follows:

	letter:	position:	
Decoded input:	`a b c d e`	0 1 2 3 4	$N = 5$, $key = 2$
Encoded output:	`c d e a b`	2 3 4 0 1	$(x + key)\ \%\ N$ // where x represents the position

The reverse decoding operation, i.e. going from an encoded input "back" to a decoded output using a given key, can be performed using a similar algebraic operation.

Given this encryption scheme, the text `"bad deed"`, for example, will be encoded as `"dca abba"`. For simplicity, we assume that the text contains only lowercase English letters, and possibly space characters. The encoding operation ignores the space characters, i.e. leaves them as is.

Note the relationship between the letters and their positions, or index values. In the above example, the index values of `'a'`, `'b'`, `'c'`, `'d'`, `'e'` are 0, 1, 2, 3, 4, 5. In ASCII, the numeric values of the `char` values `'a'`, `'b'`, `'c'`, ..., `'z'` are 97, 98, 99, … 122. Thus, if we subtract 97 (the ASCII value of `'a'`) from each one of these `char` values, we get that `'a'`, `'b'`, `'c'`, ..., `'z'` will be represented by 0, 1, 2, …, 25 (altogether, 26 letters). We emphasize, once again, that in Java, a `char` value like `'a'` is actually represented by the numeric value 97. Therefore, a statement like `System.out.print('a' + 2)` will print 99. Taken together, all these tips give you everything you need in order to write a program that encodes and decodes texts using the method described above. And, after writing this program you will understand perfectly well how `char` values are handled, and how they can be manipulated.

Here is how the program operates:

```
% java CodeOps "defend the northern wall" 4
Encoded: hijirh xli rsvxlivr aepp
Decoded: defend the northern wall
```

The program receives a string and a key, and prints the encoded string and then its decoded version. To clarify, the second line prints the string resulting from decoding the encoded version of the given string, using the given key. Of course, this is done for testing purposes.

Write a program, `CodeOps`, that performs these operations. Implementation tip: you will need three public functions: `main`, `encode`, and `decode`. The output shown above should be implemented in the `main` function.

5. Encryption II (bonus exercise)

As it turns out, the `CodeOps` program can be made to be smarter. For example:

```
% java CodeOps "cnozk krkvngtzy gxk xgxk"
Decoded: white elephants are rare
```

We see that the program knows how to decode encoded texts without getting the key!

For a 10-points bonus, extend the `CodeOps` program to handle this capability.

Implementation tips:

1. The same `CodeOps` program should handle both the basic version and the bonus version. The program checks how many command-line arguments the user entered. If there is one argument, the program assumes that it is an encoded text, and it goes on the decode it (that's the bonus version). Otherwise, the program assumes that the user entered a string and a key, and it goes on to encode and decode the text, as we did before (that's the basic version). Assume that the program always gets either one or two command-line arguments.
2. Use overloading to define two `decode` functions. The basic version gets a string and a key as parameters; the bonus version gets one string parameter.
3. How does the bonus version of `decode` work? You have to figure it out yourself, but here is a big tip: the most frequent letter in English texts that are large enough is `'e'`.
4. In order for the bonus version of `decode` to work, we have to make two assumptions: (i) the given text was encoded from a text written in the English language, and (ii) the most frequent letter in the original text was indeed `'e'`. To illustrate the importance of these assumptions, note that the program will have no problem decoding the encoded version of `"far out in the uncharted backwaters of the unfashionable end of the western spiral"`, but it will fail to decode correctly the encoded version of `"far out in the uncharted backwaters"`. Why? Because in the former text, the most frequent letter is `'e'`, and this is not the case in the latter text. In general, if the text is sufficiently long, and is in English, we can be quite sure that the bonus decoding will work correctly.

Submission

Before submitting your solution, inspect your code and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Both documents can be found in the Moodle site, it is your responsibility to read them. Any deviations from these guidelines will result in points penalty. Submit the following files only:

- `LoanCalc.java`
- `Primes.java`
- `LetterFreq.java`
- `CodeOps.java`

If you submit the basic version of `CodeOps` you can get a maximum of 100 points. If you submit the bonus version you can get a maximum of 110 points.

Deadline: Submit Homework 4 no later than November 22, 23:55. You are welcome to submit earlier.