**Homework** **3**

**1. 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 10th 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:

Graphical user interface, application, table, Excel

Description automatically generated

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. So, you can play with this spreadsheet, which is supplied with the homework (open it using either Excel of 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.

You may ask – why do I have to use a 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 trial and error method to solve it gives a good understanding of the loan’s model. Also, there are many complex financial problems that *cannot* be solved using a formula, and must be solved instead using a similar spreadsheet model. To sum up, playing with this spreadsheet will give you good intuition, and this intuition will help you write the following Java program.

**The LoanCalc program** gets three inputs: a loan amount, an interest rate, and number of payments. It them 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. Inspect the supplied skeletal LoanCalc.java class file, and make sure that you understand its logic and structure.

/bla

**Getting started**: even though these functions do very little work – if any – the overall given version of the Calendar0 program provides a complete, executable, skeleton of all the code that has to be developed. This is an example of good software engineering: The developer can execute the code, and test it – sort of – before he or she actually sets out to write it.

/bla

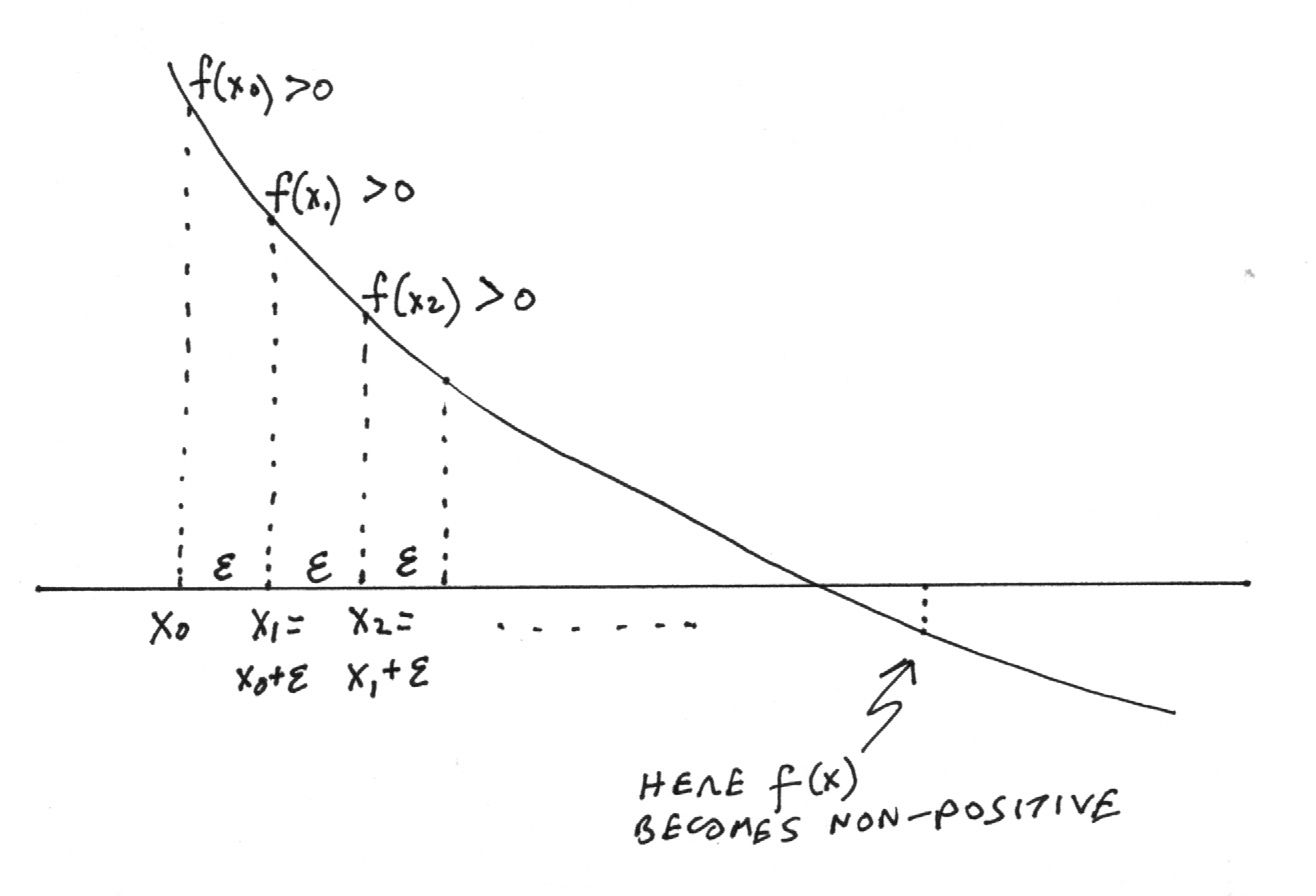
**Computing the ending balance:** Start by implementing the ​endBalance(loan,​ ​rate,​ ​n,​ ​payment)​ function.

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 will bring the loan’s ending balance close to zero? Formally, we wish to find *x* such that , 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 annual 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 ​,​ for which we know that ​. Then, we set to , where is a small value, and check if . We repeat this process until becomes non-positive. At this point we return ​​, which will be an approximation of the correct solution. How good an approximation? The correct solution will be somewhere in the interval . So, the smaller is , the better will be the approximation. In the following image, the value of  in iteration *i* is represented as *xi* :



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 ​​. 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.
* 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 , ,  
//​ 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

while {

//​ Sets ​*L* and ​*H*​ for the next iteration

if

​​ ​//​ 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 () 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).

**2. Lower case**

(15 points) Write a program (LowerCase) that gets a command-line string argument, and prints a string which is identical to the given string, except that all the upper-case letters are converted to lower-case letters. Notice that characters that are not letters are left as is. Here are two examples of the program’s execution:

% **java LowerCase B2C**

b2c

% **java LowerCase "TLV to LA: 15 Hours."**

tlv to la: 15 hours.

Implementation tips: Start by defining an empty string that you will gradually evolve into the answer string. Use a loop that iterates through the given string’s characters, using the str.length() and str.charAt(int) functions. If the character is a letter, convert it. Then add the character to the answer string. How to know if a given character is a letter, lower-case, or upper-case? Consult the ASCII table, which will give you hints how to write your code.

**3. Unique characters**

(15 points) Write a program (UniqueChars) that gets a command-line string argument, and prints a string which is identical to the original string, except that all duplicate characters are removed, unless they are space characters. Here are two examples of the program’s execution:

% **java UniqueChars committee**

comite

% **java UniqueChars "yael played the yokelele"**

yael pd th ok

Implementation tips: Create a new empty string, and grow it iteratively, using the processed characters. You will need to use some String functions, including str.indexOf(char).

**4. Calendar**

We wish to write a program that gets a year, like 2021 or 1912, and prints the *calendar* of this year. We’ll approach this objective gradually. First, we will write and test two key calendar-oriented functions. Second, we’ll write a program that prints all the dates from January 1, 1900 to December 31, 1999 (covering the entire 20th century). Finally, we will write the required calendar program.

**Calendar0**

(20 points) Write a program (Calendar0) that takes a year (like 1492 or 2021) as a command-line argument, and prints two things (1) weather the given year is a common year or a leap year, and (2) The number of days in each month in that year. Here are two separate examples of the program’s execution:

|  |  |
| --- | --- |
| % **java Calendar0 2018**  2018 is a common year  Month 1 has 31 days  Month 2 has 28 days  Month 3 has 31 days  Month 4 has 30 days  Month 5 has 31 days  Month 6 has 30 days  Month 7 has 31 days  Month 8 has 31 days  Month 9 has 30 days  Month 10 has 31 days  Month 11 has 30 days  Month 12 has 31 days | % **java Calendar0 2020**  2020 is a leap year  Month 1 has 31 days  Month 2 has 29 days  Month 3 has 31 days  Month 4 has 30 days  Month 5 has 31 days  Month 6 has 30 days  Month 7 has 31 days  Month 8 has 31 days  Month 9 has 30 days  Month 10 has 31 days  Month 11 has 30 days  Month 12 has 31 days |

In order to perform its operation, the program uses two functions: The isLeapYear(int) function checks if the given year is a leap year. The nDaysInMonth(int,int) function returns the number of days in the given month of the given year. Both functions are defined in the given Calendar0.java file. Each function has a tester function: The isLeapYearTest(int) tester is giby compiven; the nDaysInMonthTest(int,int) tester should be written by you.

**To get started, compile and run the given Calendar0.java program.** Make sure that you understand the program’s logic and structure. As usual, the program produces nonsense, since the skeletal version of the functions return arbitrary values.

**Complete and test the isLeapYear function**. We wrote the basic logic of this function in lecture 1-2. Next, write the nDaysInMonthTest function. This test function uses a loop t call the nDaysInMonth function 12 times. And since the latter function was not yet implemented, you will get the result that every one of the 12 months has 0 number of days.

**Next, complete and test the nDaysInMonthTest function**. The test will print nonsense, since the nDaysInMonthTest function is not yet implemented.

**Finally, implement the nDaysInMonth function**. For your practice, you must use a switch statement. Notice that nDaysInMonthTest must call isLeapYear.

**Calendar1**

(20 points) The 20th century is defined as the period between 1.1.1900 and 31.12.1999, inclusive. Let us start by noting an obscure fact: 1.1.1900 was a Monday. It turns out that this information is all we need in order to print the calendar of every year after this date.

Write a program (Calendar1) that prints the calendars of all 100 years of the 20th century. In addition, the program counts and prints how many Sundays fell on the first day of the month during the 20th century. In other words, you have to handle about 36,500 days; if the current day happens to be a Sunday, and also the 1st day of the month, you have to count it. Here is an example of the program’s execution:

% **java Calendar1**

1/1/1900

2/1/1900

3/1/1900

4/1/1900

5/1/1900

6/1/1900

7/1/1900 Sunday

8/1/1900

...

30/8/1901

31/8/1901

1/9/1901 Sunday

2/9/1901

3/9/1901

4/9/1901

5/9/1901

6/9/1901

7/9/1901

8/9/1901 Sunday

9/9/1901

10/9/1901

...

29/12/1999

30/12/1999

31/12/1999

During the 20th century, 172 Sundays fell on the first day of the month

Notice that 1.9.1901 is one of these special Sundays that we wish to count.

The program generates about 36500 days and prints that many lines. If you want the program to stop after *n* days, you can do it by setting a debugging variable (explained in the code). To check the output’s correctness, you can consult [this website](https://www.timeanddate.com/calendar/?year=1901&country=1). Focus on the end of the output, and check the dates of a few Sundays. If they fall on the same dates as shown on the website, your program seems to be working properly.

Implementation tips: We know that 1.1.1900 was a Monday. Starting from this base value, you have to advance several cyclical counters: The day of the week (1, 2, ..., 7, 1, 2, ..., 7, ...), the day of the month (1, 2, ..., *n*1, 1, 2, ..., *n*2, ...) where *n*1, *n*2 , ... are 28, 29, 30, or 31, depending on which month and year we are in, the month of the year (1, 2, …, 12, 1, 2, …, 12, ...), and the year of the century (1900, 1901, …).

The key function in this program is advance, which advances the counters mentioned above. We recommend starting by writing this function, incrementally: write code that advances one counter, and prints the counter’s value after each change, for testing purpose. Then add another counter, print its values, and so on.

Note that you already wrote the leapYear and nDaysInMonth functions, so you can copy-paste them from your Calendar0 program.

Write your solution by completing the given Calendar1.java class file.

**Calendar**

We are ready to complete the final version of the program (Calendar.java). This program gets a given year, like 2021, and prints the calendar of that year. Here is an example of this program’s execution:

% **java Calendar 2024**

1/1/2024

2/1/2024

3/1/2024

4/1/2024

5/1/2024

6/1/2024

7/1/2024 Sunday

8/1/2024

...

1/11/2024

2/11/2024

3/11/2024 Sunday

4/11/2024

5/11/2024

6/11/2024

7/11/2024

8/11/2024

9/11/2024

10/11/2024 Sunday

11/11/2024

12/11/2024

13/11/2024

14/11/2024

15/11/2024

16/11/2024

17/11/2024 Sunday

...

26/12/2024

27/12/2024

28/12/2024

29/12/2024 Sunday

30/12/2024

31/12/2024

Implementation tips: This program can be implemented easily by making several simple modifications to the Calendar1 program. Start by inputting the given year, as a command-line argument. Then enter a loop that advances the days (without doing anything else) from 1.1.1900 until the last day of the year before the given year. Then enter a second loop that prints the calendar of the given year. There is no need to count and print the number of Sundays that fall on the first day of the month.

For this program we provide no skeleton. Make a copy of your working Calendar1.java file, change the file name and the class name to Calendar.java, and proceed to make the necessary changes.

**Submission guidelines**

Before submitting your work for grading, make sure that your code is written according to our [Java Coding Style Guidelines](https://drive.google.com/open?id=1Yr6FFaSTw07Tp4gU7OYrgW_1Rluo3B4G).

Submit the following files only:

* LoanCalc.java
* LowerCase.java
* UniqueChars.java
* Calendar0.java
* Calendar1.java
* Calendar.java

Submit on Github: <https://classroom.github.com/a/GOfbyh27>