

Eiffel Programming Course

2003-12-17

1 SEARCH

Search for the largest and the second largest element in an array of integers and write them to the console using **io**.

Try to find the two numbers in one loop.

Write your code in the feature **make** of class **SEARCH**. The local variable **a** is an array filled with randomly chosen numbers from **0** to **1023**.

You can find these files in the directory **search**.

2 SPECIAL COUNTER

Write a simple feature **special_counter (n: INTEGER)** that displays all the numbers from **1** to **n** except for the numbers that are multiples of **3** or contain the digit **'3'**.

Example: n=20: 1, 2, 4, 5, 7, 8, 10, 11, 14, 16, 17, 19, 20

Hints:

- **$i \ \% \ 3$** returns the remainder of **i** divided by **3** (e.g. **$7 \ \% \ 3 = 1$**) with **i** of type **INTEGER**.
- Use **i.out** to get a **STRING** so that you can check whether "3" is one of the characters (**i** is of type **INTEGER**).

3 ROUTE BUILDER

Implement all routines of the classes **PLACE**, **ROUTE**, and **ROUTE_SEGMENT**, so that the feature **build_route** of class **ROUTE_BUILDER** can be correctly executed.

You can find the project in the directory **route_builder**.

4 EWIGER KALENDER

Das Ziel ist eine Applikation zu schreiben, die ein Datum liest und für dieses Datum den entsprechenden Wochentag ausspuckt.

Schaltjahre sollen berücksichtigt werden. Das Ganze kann mit zwei Klassen programmiert werden: **DA-TUM** und **KALENDER**. Ihr könnt selbst entscheiden, wie ihr das Datum einlesen wollt (19.12.2003[enter] oder 19[enter] 12[enter] 2003[enter] oder anders).

Zu Schaltjahren muss man folgendes wissen:

1. Alle 4 Jahre und dabei genau zu den durch 4 teilbaren Jahren gibt es einen zusätzlichen Tag im Februar, den 29. Februar. Diese Jahre werden "Schaltjahre" genannt. 1904, 1984, 1996 sind 3 Beispiele für Schaltjahre.
2. Regelung 1 tritt alle 100 Jahre und dabei genau zu den durch 100 teilbaren Jahren (= "Säkularjahre") ausser Kraft. 1700, 1800, 1900 sind 3 Beispiele für Nicht-Schaltjahre, die laut 1 welche sein müssten.
3. Regelung 2 tritt alle 400 Jahre und dabei genau zu den durch 400 teilbaren Jahren ausser Kraft. 2000, 2400, 2800 sind 3 Beispiele für Schaltjahre, welche laut 2 keine sind.

5 FACTORIAL

Develop an application that calculates the factorial of a number that has been entered on the console. The factorial is the product of all the positive integers from one up to and including a given integer. Factorial of zero is assigned the value of one by definition.

Simple Version

1. Create a new class **FACTORIAL** with a creation procedure **run** that will launch the application.
2. Implement **run** such that the following requirements are met:
 - The application should ask the user to enter a non-negative number, or "quit" to exit the application.
 - If the user has entered a valid non-negative integral number, calculate its factorial and output the result to the console.
 - Otherwise display an appropriate error message.

Hint: Read in a string first, and then ask the string whether it represents indeed an integer.

3. Implement the feature **simple_calculate** that computes the factorial of a non-negative number.

Hint: Use a loop to accumulate the intermediate results.

Advanced Version

The factorial of a number **n** can also be inductively defined:

$$\begin{aligned}\mathbf{factorial(0)} &= \mathbf{1} \\ \mathbf{factorial(n)} &= \mathbf{factorial(n - 1)} * \mathbf{n}\end{aligned}$$

Use this fact and eliminate the need for a loop. Write a feature **advanced_calculate**.

Hint: **advanced_calculate** calls **advanced_calculate** again, but with a different argument.

6 COMPILATION PROBLEM

In its current state, the following text for the class **TINY_COUNTER** will not compile. Find the 5 causes of compilation errors and report on paper what has to be done in order to fix them.

Hints:

- 1 Error "type is based on unknown class"
- 3 Errors "Syntax error"
- 1 Error "unknown identifier"

indexing

description: "Counter that you can increment by one, decrement, and reset"

class TINY_COUNTER

feature

item: INT
// Counter's value.

feature -- Element change

increment is

-- Increase counter by one.

do

item = item + 1

ensure

item = old item + 1

end

decrement is

-- Decrease counter by one.

require

item > 0

do

item := item - 1

ensure

item = old item - 1

reset is

-- Reset counter to zero.

do

items := 0

ensure

item = 0

end

invariant

item >= 0

end

7 CHECKERBOARDS

7.1 Simple Checkerboard

Write a program that takes one command line parameter **N** and prints out a two dimensional **N-by-N** checkerboard pattern with alternating spaces and asterisks, like the following 8-by-8 pattern. Use two loops and one if-else statement.

```
*.*.*.*.*
.*.*.*.*
*.*.*.*
.*.*.*.*
```

7.2 Triangle

Write a program that takes a command line parameter **N** and prints an **N-by-N** triangular pattern like the one below. Use two loops and one if-else statement.

```
* * * * *
. * * * *
. . * * *
. . . * *
. . . . *
. . . . . *
```

7.3 X

Write a program that takes a command line parameter **N** and prints an X like the one below (dimension $(2*N+1)$ -by- $(2*N+1)$). Use two loops and one if-else statement.

```
* . . . . *
. * . . * .
. . * . * .
. . . * . .
. . * . * .
. * . . * .
* . . . . *
```

8 MASTERMIND

The goal of the game mastermind is to correctly guess a sequence of colours. For each guess, the player receives the following feedback:

- one black marker for each colour that he/she guessed correctly
- one white marker for each colour that also appears in the sequence, but at a different position; or, in other words: one white marker for each black marker that could be achieved additionally by rearranging the guessed colours

Examples:

sequence: green red blue red
guess: yellow green blue green
feedback: black white

sequence: red green red red
guess: red red green green
feedback: black white white

The markers do not indicate the position of the correct colours, only their number is relevant. A simplified version of Mastermind only reports the correct colours, i.e. the number of black markers. Since it's a little tedious to calculate with colours, we will work with numbers instead.

1. Write a program that lets you play the simplified version of Mastermind. For this, you need a class that knows the correct sequence (the solution) and reports the black markers for a guess. You can either let the user enter the correct sequence at the beginning of the game, or else define the correct sequence as a once feature. Use one class to implement the Mastermind functionality, and another class to handle all the user input and output with **io**.
2. Generate the solution using a random number generator instead of user input or once feature. A random number generator is implemented in class **RANDOM**. This class has two creation features, **make** and **set_seed**. **make** just calls **set_seed** with a predefined argument. The easiest way to use an instance of class **RANDOM** is to make use of the features **start** to initialize it, **forth** to move to the next random number, and **item** to read the current random number.

In order to get new random numbers every time the program runs, the **RANDOM** instance should be seeded with a changing number. While user input can certainly be used, it is more convenient to use the current time. For this, you need to include the library "time" into your project. Open "Project → Project settings" from the menu and switch to the Clusters tab. After clicking the "Add" button, you are asked to enter a name for the library and its location. The library is the directory "library/time" in your EiffelStudio directory. After closing this dialog window, you need to exclude the sub-clusters 'german' and 'french' (or 'english' and 'french', or ...) to get rid of multiple classes defined for different languages.

I suggest defining your random number generator as a once feature:

```
rng: RANDOM is
  -- random number generator
  local
    time: DATE_TIME
  once
    create time.make_now_utc
    create Result.set_seed (time.seconds * 1000 + time.time.milli_second)
    Result.start
  end
```

3. Extend your program such that it also reports the number of white markers.

9 TIC TAC TOE

In this exercise you will implement four missing features of a Tic Tac Toe. Two human players can play against each other. The players have to enter the coordinates where they want place their mark in the form **x y**. **1 1** is the upper left corner.

Do the following:

1. Open the Ace-file of the project **tic_tac_toe** and compile it.
2. Look at class **BOARD**. It uses **ARRAY2** from the library to store the state of the game.
3. Implement the three missing features of **BOARD**.
4. Look at class **ROOT_CLASS**. **make** (the root feature) does the interaction with the players. It loops until a player has won the game. In each iteration it does the following:
 - Ask current player for **x** and **y**.
 - Place current player's mark at **x, y** and see if he has won.
 - If not, the other player becomes the current player.
5. Implement the missing feature of **ROOT_CLASS**.
6. Compile the application.
7. Play the game.

10 MULTIPLICATION JUST WITH ADDITION

Given the following fragment of a Eiffel feature try program out the feature multiplication, so that it calculates the product of the multiplication **a * b** just with the help of a loop statement and the addition operator **+**. Do not use the multiplication operator ***** to solve this task!

Hint: A multiplication is nothing more than the repeated addition of the same number.

```

multiplication (a, b: INTEGER): INTEGER is
  -- Calculate the product of a times b and return it as the result.
  do
    -- what you have to implement
  end

```

11 HANDLING WITH LISTS

You want to implement a simple database for names. As an internal datastructure you can use a **LINKED_LIST** object which offers the following features:

- **start**: moves to the first element of the list
- **forth**: moves to the next element (one after that one which was selected before)
- **after**: returns true, if there is no more element
- **item**: returns the current element

Your **NAME_DATABASE** class (the one you will implement) should offer a feature which prints out all names which are in that database and which are alphabetically after a given string:

```

print_list_with_names_larger_than (this: STRING) is
  -- goes through the list, compares the item with 'this'
  -- and prints it out if it is larger (uses < operator)
  do
    -- what you have to implement
  end

```

The **LINKED_LIST** is generic. So you need to specify what kind of **LINKED_LIST** you want — since we use here string for the names, you can use something like

```

feature {NONE}
  names: LINKED_LIST [STRING]

```

Necessary steps:

1. Create a new class **NAME_DATABASE** with its features
2. Define it a your root class
3. Add a feature **make**, which is the creation feature. That feature will fill the database with some names and after that will do some printouts of names.

APPENDIX

The Loop Statement

In Eiffel, a loop has the following syntax:

```
-- the code before the loop
from
    -- initialization:
    -- what has to be done before
until
    -- exit:
    -- when we have to stop to repeat
do
    -- body:
    -- the part which should be executed
    -- several times
end
-- the code after the loop
```

Before we start with the body, we execute once the initialization part and check the boolean exit condition. If the exit condition is **False**, we execute the body and check for the exit condition again. If the exit condition is **True**, the loop is finished and the code after the keyword **end** is executed.

The examples were prepared by Volkan Arslan, Till Bay, Susanne Cech, Jörg Derungs, Peter Farkas, Michela Pedroni, Matthias Sala, Sebastien Vaucouleur, and Tobias Widmer.