Exam Computer Programming II 2021-08-26

Exam hours: 14:00 – 19:00

Note: This exam is intended for those students who have taken the course in the fall 2020 or later. The exam for those who have studied the course earlier with Java can be found here.

Practical details and help during the exam:

- The teachers will often be available in this Zoom-room during the exam. You are welcome to spend the exam in that room, as during regular lessons.

  Do not ask questions in the ”main room”, but go to a breakout room and sign up in the document mentioned below.

- We will post any late information about, for example, errors in the exam or answers to common questions in this document. Keep it open during the exam!

  You can also use this document to contact a teacher for questions. Fill in your name and a breakout room number and a teacher will show up.

- You can also put question on Slack, per email (sven-erik.ekstrom@it.uu.se or tom.smedsaas@it.uu.se), or by phone 0720575820 (Sven-Erik) or 0702544244 (Tom). Do not call but send a SMS. We will call you.

Submission:

- Submission of the exam takes place by uploading files as a regular assignment in Studium.
- The files to be uploaded should be named `m1.py`, `m2.py`, `m3.py`, and `m4.py`. You should start with downloading ”skeletons” for these from Studium.
- The exam consists of A- and B-tasks. An A-task must work correctly (submitted programs must be able to run and solve the task) to be considered as passed. An incomplete A-task gives zero points. B-tasks can give ”points” even if they do not solve the problem completely.
- We prefer that you upload the files to Studium in a single operation (”drag-and-drop” all files to ”Upload file”). **Don’t forget to press ”Upload”!** You can chose a zip-fil or, if you have some trouble, upload one file at a time.
- You can upload files several times during the exam by clicking ”Try again”, then the last version uploaded will be considered as the one you submitted.
- If there is a problem with Studium can you email the files to sven-erik.ekstrom@it.uu.se.
- **No submissions after 19:00 will be accepted!**

Rules

- You may not collaborate with anyone else during the exam. You may not copy code or text but you must write your answers yourself.
- You may use the Internet.
- You have to write your solutions *in designated places* in the files `m1.py`, `m2.py`, `m3.py`, and `m4.py`. Write your name and code (if you have received one) in the comments at the beginning of each file. You must keep the names of files, classes, methods and functions.
- You may not use packages other than those already imported into the files unless otherwise stated in the task. (The fact that a package is imported *does not* mean that it needs to be used!)
- Before your exam is approved, you may need to explain and justify your answers orally to a teacher.

PLEASE NOTE THAT WE ARE OBLIGATED TO NOTIFY ANY SUSPICION OF ILLEGAL COOPERATION OR COPYING AS A POSSIBLE ATTEMPT TO CHEAT!

Components of the exam:

The exam contains A-tasks and B-tasks.

The exam is divided into four sections, each with tasks corresponding to the four modules.

To module 1 belong the tasks:

A1: Use recursion to count instances of a specified value.
A2: Use *memoization* to make a given function more efficient.
B1: Time estimate for a given function.

To module 2 belong the tasks:

A3: An operator for modulos.
A4: A command to clear the variable list.

To module 3 belong the tasks:

A5: A method for removing all elements with a certain value from a linked list.
A6: A calculation for the complexity of repeated additions.
A7: A method of counting leaves in a binary search tree.
A8: Overloading the `==`-operator for binary searchtrees.
B2: Insertion in linked lists.
B3: Complexity analysis of a given function.

To module 4 belongs the tasks:

A9: Parallelization.
A10: Understand and modify code.
B4: Higher order functions.

Betygskrav:

3: At least eight A-tasks passed.
4: At least eight A-tasks and two B-tasks passed.
5: At least eight A-tasks and all B-tasks passed.

Tasks in connection with the module 1

The solution to this task must be written in the designated places in the file `m1.py`. The file also contains one `main`-function which tests the code. Feel free to add more tests!

A1: Write the function `count_all(lst, d)` which counts how many times `d` occurs on *all* levels in `lst` The function should handle sublists with *recursion*.

Exempel:
`count_all([], 1)` = 0
`count_all([1], 1)` = 1
`count_all([1], 0)` = 0
`count_all([1,2,1,3,[[1],[1,2,3]]], 1)` = 4

You can assume that the first argument `lst` is a list (possibly empty). It is allowed to iterate over the list, but sublists must be treated recursively. It is not permitted to flattenthe list.

A2: The function below can not handle very large values of the argument n. Write the function `c_mem(n)` that still uses *recursive* calls but also the technology with *memoization* (also called *dynamic programming*) to handle large values of n. Calculate `c_mem(100)` and include that in your answer.

```
1 def c(n):
2     if n <= 2 :
3         return 1
4     else:
5         return c(n-1) - c(n-3)
```

B1: How long time would it take to calculate `c(100)` on your computer with code A2 unchanged. Motivate the answer!

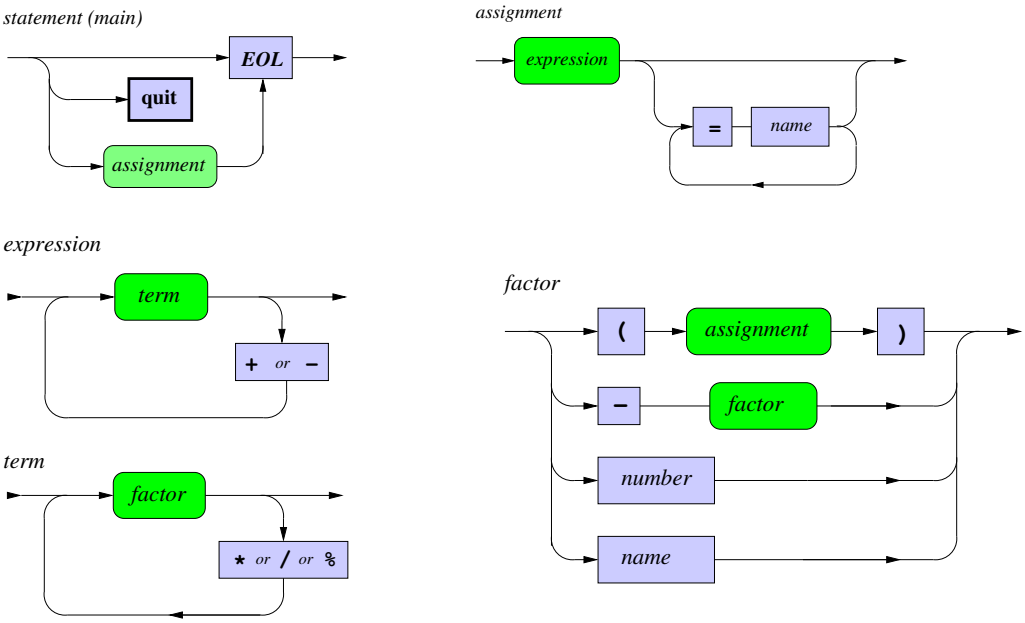
Since you have to answer for what time it would take on your computer, you will probably need to do some time measurements. Include the code you use for this in the `main` function. Explain your reasoning in the multi-line stringät the end of the file.

Note: It should be possible to run this code on another computer!

Tasks related to module 2

The downloaded code `m2.py` implements a calculator similar to the one in the second mandatory task. Some items have been removed and some need to be added. Note that the file also contains the `TokenizeWrapper` class.

The syntax of the expressions is described by the following charts:



In the given code all functions (green boxes) are included but not everyone works quite as they should. The tasks thus consist of supplementing or modifying the functions and, if necessary, add some help functions.

The program starts by reading commands from a file named `init.txt` (which you downloaded in the beginning). The file starts like this:

```
1 (2+4=x) - x                          # 0.0
2 2*(3*x + 2)/(16-x)                   # 4.0
```

The expected result is after the comment sign `#`.

In the rest of `init.txt`, there are test cases for the code you are supposed to implement.

A3: Add code to handle the remainder operator `%`. Example:

```
Input : 1%2
Result:  1.0
Input : 3%3
Result:  0.0
Input : 10 + 23%17 + 2
Result:  18.0
Input : 4%0
EvaluationError:  Division of 4.0 by zero
Input :  3%4%2
Result:  1.0
Input : 5%(3*2)
Result:  5.0
Input : (5+3)%(2*3)/2
Result:  1.0
Input : (5+3)%(2*3/2)
Result:  2.0
Input : (5+3)%(2*3/2 - 1.5*2)
EvaluationError:  Division of 8.0 by zero
```

The operator should have the *same* priority as `*` and `/`. With equal priority, the operations are performed from left to right.

A4: Add the `clear` command `main` which deletes all defined variables.

Example:

```
Input : 1 = x = y = z
Result:  1.0
Input : z + x
Result:  2.0
Input : clear
Input : y
EvaluationError:  Undefined variable: y
```

Tasks related to module 3

In this section you will work with the file `m3.py` which contains the classes `LinkedList.py` and `BST.py` and the function `bst_sort`.

There is also a `main` function with some simple tests.

The `LinkedList.py` class contains code for handling *linked lists* of objects. The lists are *not* sorted and the same value can occur several times. In the examples, integers are used as data, but the code must work for all types of objects.

The class `BST` contains code for ordinary binary search trees.

A5: There is a method in the given code for `LinkedList`

```
    def remove_all(self, x):
        self.first = self._remove_all(x, self.first)
```

which should remove all nodes that contain `x` from the list.
Write the *recursive* help method `_remove_all(self, x, f)` which removes all nodes that contain `x` from the list starting with node `f` and returns the first node in the remaining list.

A6: How does the run time for the code below depend on n?

```
lst = LinkedList()
for x in range(n):
    lst.add_last(x)
```

Reply with a Θ- expression and motivate! Write the answer in the text string at the end of the downloaded file!

A7: In the class `BST` there is the method `count_leaves(self)` which should return the number leaves in the tree (i.e. the number of nodes without children). The method uses the help method
`_count_leaves (self, r)`. Write the help method!

A8: Overload the operator `==` in the class `BST` so it returns `True` if the two trees contain the same data (regardless of shape), otherwise `False`.

Example:

Code:

```
    bst1 = BST([2, 10, 1, 7, 5, 8, 3])
    bst2 = BST([10, 1, 5, 3, 7, 8, 2])
    print(bst1 == bst2)
    bst1.add(4)
    print(bst1 == bst2)
    print(BST()== BST())
    print(BST()== BST(1))
```

Printouts:

```
True
False
True
False
```

B2: Write a method `insert(self, x, index =0)` in the class `LinkedList` that adds a new node with `x` as data at position `index` in the list.

Example (the comments show resulting list):

```
lst = LinkedList()
lst.insert(3)           # <3>
lst.insert(5, 1)        # <3, 5>
lst.insert(5)           # <5, 3, 5>
lst.insert(4,1)         # <5, 4, 3, 5>
lst.insert(1, 99)       # Index out range: 99
```

B3: How does the time for the function below of depend on the length n of the list `aList` (default Python list)? The list can be assumed to contain random numbers.

```
def bst_sort(aList):
    bst = BST()
    for x in aList:
        bst.add(x)
    result = []
    for x in bst:
        result.append(x)
    return result
```

Answer with Θ- or (realistic) $\mathcal{O}$-expressions.

Motivate the answer either theoretically or with reported experiments or both.

Write the answer in the designated text string at the end of the file!

Tasks related to module 4

A9: The file `customer.json` (that you have in your downloaded files) is a so-called JSON-file (a text based file format to store data of different types).

This file contains customer files for a fictitious company; a number of people that all have an index (0,1,2,...) with different properties/fields.

In the file `m4.py` you have the method `get_balance(index)` that uses the module `json` to extract the field `balance` from `customer.json` for the person of a given index (0,1,2,...,111), since there are 112 customers. If you call `get_balance(0)` you will get the first person's (index 0) `balance` which is 1801.02. If you open tile file `customer.json` in an editor you will see that for person 0, there is

```
"balance": "$1,801.02"
```

but the method `get_balance(index)` removes `$` and `,`.

Modify the method `get_total_balance()` in `m4.py` so that it returns the total balance of all customer's balance (the sum). The retrieval of all customer's `balance` should be done in parallel. You can use any parallelization you want to solve this, the goal is not a faster code, but to show that you can parallelize code.

A10: In this task you will again use the file `customer.json`, that you used in A9.

Modify the code `get_mean_balances()` so that it returns the mean balance for male and female customers. Every customer has a field 'gender', that is either 'male' or 'female'. It does not matter if you return a list or tuple.

B4: A leap year (a year when you add an extra day) is defined as

Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400. For example, the years 1700, 1800, and 1900 are not leap years, but the years 1600 and 2000 are.

Modify the method `leapyears(years)`, which has an input argument 'years' which is assumed to be a list of years, that it returns all leap years in the list. Use a one line comprension that creates a list of all leap years.