**COVER PAGE FOR DIGITAL ON CAMPUS EXAMINATION IN CANVAS**

**Name of the course**: Data Structures & Algorithms
**Course code**: 822188-B-6
**Date of the examination**: 25-01-2023
**Duration of the examination**: 3 hours
**Lecturer**: Pieter Spronck **ANR**: 447823
**Lecturer's telephone number available during examinations**: 8118

**Students are expected to conduct themselves properly during examinations and to obey any instructions given to them by examiners and invigilators. Firm action will be taken in the event that academic fraud is discovered.**

All mobile devices (telephone, tablet, etc.) must be switched off in the exam room and should be stored in your jacket or bag. You are not allowed to carry these during the exam. It is also not allowed to wear a watch (both analog and digital) in the exam room.

**Exam instructions**

1. Only the following examination aids are permitted during the exam:
   a. Books, lecture notes (either in printed form, book form, or digitally supplied on the university computers):
      - Python documentation as supplied with Python
      - The book "The Coder's Apprentice"
      - The book "Think Python"
      - Notebooks, either local notebooks or on the JupyterLab server (jupyterlab.uvt.nl)
      - A Python cheat sheet as supplied on Canvas
      - Lecture sheets as supplied on Canvas
      - Anything else that is supplied on Canvas for this course
   b. Blank scrap paper and pens. Scrap paper and other hard copy materials may be taken home by the students at the end of the exam.
   c. Student's own calculator (any).
   d. Internet sources: https://jupyterlab.uvt.nl (this is the Jupyterlab server of the university) and https://docs.python.org.
   e. Computer programs: Any of the programs in the default installation, including Python, Anaconda, Spyder, Jupyter, PyCharm, Visual Studio Code, and IDLE.
2. In case a student does not comply with instructions paragraph 1, this can be qualified as (a suspicion of) fraud. The EER, Rules and Guidelines of the Examination Board of the School and the General Guidelines apply.
3. The exam has 5 questions.
4. For the course, the expected passing score (the score to pass the exam) is 6. The midterm counts for 40% of the final grade for the course. The score is calculated by taking 40% of the midterm score and 60% of the exam score, unless the exam

score is higher than the midterm score (or the student has no midterm score), in which case the exam score counts for 100%. Standard rounding to half points applies, except that 5.5 or higher will be rounded to 6, while 5.4 or lower will be rounded to 5.

5. Each question is worth 2 points.
6. The main requirement for getting all the points for the answer is that the student's code processes the tests given by the instructors correctly. In degenerative cases (such as code being unintelligible, code being exceptionally slow, code only being able to handle the test cases supplied in the template files but not other test cases, or a student replacing a loop with a long list of conditional statements) points will be subtracted. Code that crashes may not get points at all.
7. Each question has its own template code file, in which the students have to fill in a function, methods, create classes, or add comments. A question which needs input files, will have these supplied as well, and these input files should be placed in the same folder as where the template code files are placed. The code files are to be submitted as answers. Please meet the following requirements:
   a. Students should add their name and student number at the top of each file as a comment.
   b. Only the standard Python modules may be used to develop code (as discussed in the first 32 notebooks), so no numpy or pandas.
   c. Students may add extra tests in the main() function that is found in each of the code files.
   d. If a student needs to add print() statements in a function that they are developing for debugging purposes, these print() statements should be removed before submitting the final code.
   e. Functions never need to ask the user for input. All inputs for the functions are supplied via parameters.
   f. If a student wants to use the notebooks as editor, then they should copy the code from the Python files to the notebook in which they want to do the editing, and after having finished their code, copy the code back to the original file. They then submit the original file.
   g. Code files should be tested before submitting, especially if notebooks were used to develop the code (to catch copying mistakes).
   h. If a student wants to add remarks to submitted code, please do so as comments in the code files that are submitted.
   i. Five separate code files must be submitted. Make sure that they are Python (`.py`) files! Do not change the names of the files! Do not pack them in a ZIP file! Do not submit `.ipynb` files! Do not submit screenshots! Renaming a `.ipynb` file to a `.py` file will not work, as it will not magically turn a `.ipynb` file into a Python file.
8. Not all questions are equally hard, and it tends to vary between students what they consider to be hard. If you are stuck on a question, turn to the next one, and return to the question you got stuck on later. Think before you start coding – finding a good approach tends to save a lot of time.
9. If a student has questions during the exam, they should ask those to an instructor present, or in the Discussion forum on Canvas. The instructors will keep track of

those questions and answer them. Students should NOT post code in the forum! Students may see each other's questions, but should NOT answer each other's questions! Communicating on the contents of the exam with anyone but the instructors is not allowed!

10. The instructors will supply a discussion of each of the questions on Canvas after the grading has been completed. If any questions remain after you have viewed these discussions, please contact the instructors via the Canvas Inbox to arrange an exam inspection. This may be arranged via Zoom or Teams.

You can start the examination now, good luck!

**Exercise 1: Perfect numbers**

File: `E01_perfect.py`

**One-line summary**

In this exercise, you create a function that determines whether a number is “perfect.”

**Context**

A perfect number is a positive integer that is equal to the sum of its proper divisors. A “proper divisor” is any positive integer which is lower than the given number, which divides the given number without remainder. For instance, the proper divisors of 6 are 1, 2, and 3. As 1+2+3=6, 6 is a perfect number. The proper divisors of 24 are 1, 2, 3, 4, 6, 8, and 12. 1+2+3+4+6+8+12=36, therefore 24 is not a perfect number.

**Exercise description**

The function `perfect()` is given an integer as parameter, and returns `True` if the integer is a perfect number, and `False` otherwise. You may assume that the function is called with an integer which is greater than zero.

The `main()` function calls the function `perfect()` with all the integers between 1 and 10,000. There are four perfect numbers in that range. The first three are 6, 28, and 496. The fourth one you may discover by yourself.

**Hint**

While testing, it may be smart to let the test loop only run up to 500, to reduce the time needed for the search. Once you can find the first three numbers, increase the number of trials to 10,000 again.

**Examples**

`perfect( 6 )` returns `True`.

`perfect( 24 )` returns `False`.

**Exercise 2: Coauthors**

File: `E02_coauthors.py`
Input files: `papers1.txt, papers2.txt`

**One-line summary**

In this exercise, you build a dictionary of authors of scientific papers and their coauthors.

**Exercise description**

The function `build_coauthors_dictionary()` gets one parameter, which is the name of a file. This file contains, on each line, the title of a paper, several names of authors, and finally a number which indicates how many times the paper has been cited. There are commas between each of the items. For instance, the line "`My First Paper,Anton,Bas,Corry,5`" describes a paper titled "My First Paper", which has three authors (Anton, Bas, and Corry), and has been cited 5 times. The number of authors varies between the lines, but there is at least one author per paper.

You use the file to build a dictionary of authors and their coauthors. The keys are the author names, and the values are lists of coauthors in alphabetical order. So, for each author you need to create a sorted list containing all of the coauthor names in alphabetical order, and without repetition. You may assume that the file only contains lines which are as described above. You may also assume that all the commas in the line distinguish between items, e.g., there cannot be a comma in the title of the paper.

The code template contains a test function which prints the authors in alphabetical order of author names, and with each author all their coauthors as indicated in the dictionary.

**Example**

The file `papers1.txt` contains:

```
My First Paper,Anton,Bas,Corry,5
My Second Paper,Corry,Anton,4
This I wrote by myself,Anton,11
Nobody cares about this,David,Bas,Anton,0
I have no co-authors,Eddington,77
```

The function returns the following dictionary:

```
{ 'Anton':['Bas', 'Corry', 'David'], 'Bas':['Anton', 'Corry', 'David'],
'Corry':['Anton', 'Bas'], 'David':['Anton', 'Bas'], 'Eddington':[] }
```

The output is displayed as:

```
Anton: Bas, Corry, David
Bas: Anton, Corry, David
Corry: Anton, Bas
David: Anton, Bas
Eddington:
```

**Exercise 3: Shopping basket**

File: `E03_shopping_basket.py`

**One-line summary**

In this exercise you add items to a shopping basket and calculate the final price of the whole basket.

**Exercise description**

The template file for this exercise contains four classes: `Basket`, `Item`, `WeightedItem`, and `UnitsItem`. In the exercise you need to implement the class `Basket`, and the two item classes `WeightedItem` and `UnitsItem` which inherit from `Item`. The `Item` class is already complete and should not be changed. A UML diagram is given on the next page.

`WeightedItem` is used for products with prices calculated by weights, like rice or beans. Objects of this class have a name and the price per kilogram. The method `calculate_price()` receives as argument the amount in grams and will return the price of the total amount.

`UnitsItem` is used for products with prices calculated by units, like eggs or bread buns. Objects of this class have a name and the price per unit. The method `calculate_price()` receives as argument the number of units and will return the price of the total amount.

For the `Basket` class you need to implement three methods: `__init__()`, `add_item()`, and `create_receipt()`.

Objects of the classes `WeightedItem` and `UnitsItem` can be added to a Basket. For this, you need to implement the `add_item()` method in the Basket class. The method receives an item and an amount. The amount should correspond to the kind of object, so the number of units or grams. You may decide for yourself how you keep track of the items in the basket.

The total price of all the items in the basket is calculated using the `create_receipt()` method. This method has to create a string containing lines (separated by "\n"). Each line must contain the item name, amount in the basket, and the total price. Every item should only be once on the receipt, so if you first add two buns to the basket, and later add another three buns to the basket, on the receipt there should be only one line with buns, namely five buns.

The code includes an example and the expected output.

**Hint**

It makes sense to first develop the `UnitsItem` and `WeightedItem` classes and test them, before you implement the `Basket` class.

**Example**

If you add the following four items to a basket

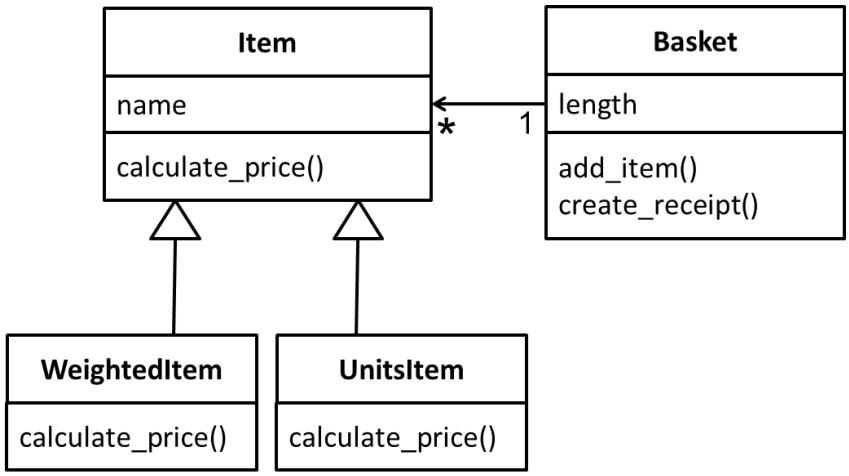
```
basket = Basket()
item_1 = WeightedItem("Oats", 3.5)
item_2 = UnitsItem("Bun", 0.7)
item_3 = UnitsItem("Baguette", 1)
item_4 = WeightedItem("Rice", 4)

basket.add_item(item_1, 500)
basket.add_item(item_2, 2)
basket.add_item(item_2, 3)
basket.add_item(item_3, 3)
basket.add_item(item_4, 1500)
```

The return value from the Basket's create_receipt function, when printed, should produce something like the following (exact formatting may differ):

```
Rice        1500   6.00
Oats         500   1.75
Bun            5   3.50
Baguette       3   3.00
Total             14.25
```

**UML diagram**



**Exercise 4: Sorted Ring**

File: `E04_sorted_ring.py`

**One-line summary**

A ring is a double-linked list in which the last node loops back to the first one. In this exercise you implement a method to insert nodes into a ring maintaining the numerical order.

**Exercise description**

This exercise uses the `Ring` class which you had to create in the December exam.

Consider a double-linked list. Normally, such a list has a head and a tail, and the double-linked list itself has references to that head and tail. A "ring" is a double-linked list, in which the head's "previous node" refers to the tail, and the tail's "next node" refers to the head. The ring has a reference to one of the nodes, which is called the `current` node. A schematic representation of a ring with four nodes is given on the next page. The code for the exercise contains a `Ring` class, which implements a ring consisting of `RingNode`s.

The `Ring` has five methods (all of which are implemented already):

- `__init__()` which creates the ring
- `__len__()` which ensure that when you ask for the length of the ring using the `len()` function it gives you the number of nodes in the ring
- `add()` which adds a node to the ring. The value for the new node is given as parameter. The new node has the `current` node as nextnode, and after insertion of the new node, the new node is the `current` node.
- `rotate()` which rotates the ring by moving the `current` node a number of `steps`, given as a parameter. If `steps` is positive, the current node is moved to the `nextnode` for the given number of steps. If `steps` is negative, the ring rotates in the opposite direction by following the `prevnode` references.
- `__repr__()` which returns a string which represents the ring, showing the values stored in the ring starting with the `current` node.

You have to implement a `SortedRing` class, which inherits from `Ring`. An UML diagram is shown on the next page. You override the `add()` method so that when you add a node to a `SortedRing`, it is placed in the ring so that the ordering of the values is preserved. E.g., if a sorted ring contains nodes with values 3, 6, and 7 (in that order, whereby the node with value 7 points to the node with value 3 as next node), and you add the value 5, the new node is inserted between the node with value 3 and the node with value 6. After insertion of a new node, the `current` node is this new node.

You may assume that the `SortedRing` will be created with *only* integer values.

**Hint**

One possible way of solving this exercise is by letting the `add()` method of `SortedRing` rotate the ring until the current node is the place where the new value must be inserted, and then use the `add()` of `Ring` to insert the new node.

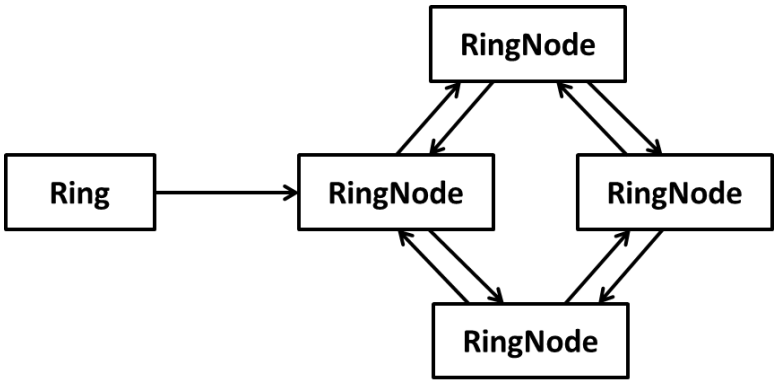
**Example**

```
ring = SortedRing()
ring.add(3)
ring.add(1)
ring.add(5)
ring.add(2)
ring.add(4)
print( ring )
```

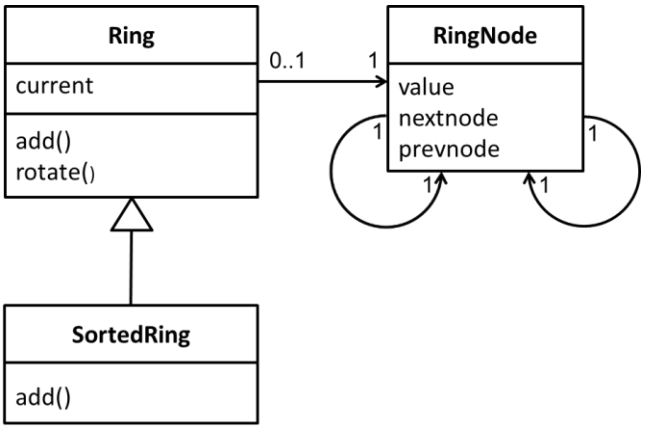
Gives as output: `Ring(4,5,1,2,3)`

**Schemas**

A schematic representation of a ring with four nodes:




UML diagram for this exercise:



**Exercise 5: Ascending letters**

File: `E05_ascending_letters.py`

**One-line summary**

In this exercise you determine the time complexity of five functions which determine whether or not the letters in a word appear in ascending order.

**Exercise description**

Each of the functions, which are all named `ascending_x()` (*x* being a number), gets one parameter: a string which is a word. The function returns `True` if all the letters in the word appear in ascending order, and `False` otherwise.

For each of the functions, write as a comment in the file what the time complexity is (immediately below the function), and give a brief explanation on how you determined this. If you do not add an explanation, the answer is automatically considered to be wrong. In your big-O notation of the time complexity, you may use `n` to refer to the length of the word.

**Background information**

You may assume the following:

- Making a copy of a string is `O(n)`, where `n` is the length of the string
- Casting a string to a list is `O(n)`, where `n` is the length of the string
- The `range()` iterator is `O(1)`
- The function `len()` is `O(1)`
- The list method `sort()` for a list of length `n` is `O(n*log(n))`
- The string method `join()` with a list of length `n` as argument is `O(n)`

Here are descriptions of the algorithms:

- `ascending_1()` compares every letter of the word to the next one.
- `ascending_2()` compares every letter of the word with all the letters in the part of the word which comes after that letter.
- `ascending_3()` turns the word into a list, sorts the list, and then compares the sorted list with the original list of letters.
- `ascending_4()` turns the word into a list, then compares each letter with the next and swaps them if they are not in ascending order; after that it turns the new list into a string again and compares it with the original word.
- `ascending_5()` recursively checks whether the first letter of a word is lower than the first letter of the rest of the word, and if the rest of the word is also in ascending order.