COVER PAGE FOR A WRITTEN EXAMINATION/TEST

Name of subject	: Data Structures & Algorithms
Subject code	: 822188-B-6
Date of examination	: 15/12/2021
Length of examination	: 3 hours
Lecturer	: Pieter Spronck ANR: 447823

Telephone number of departmental secretariat: 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.

1. This exam is a programming exam. Students submit their answers in the form of Python code files. Students may use editors installed on the university computers to create their code, including notebooks. You may use Anaconda to start an editor or local notebooks.
2. Students should only use modules that come standard with Python, as discussed in the first 32 notebooks. Thus, numpy is not allowed.
3. Students may consult the following written materials on the computer or on paper:
   - 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
4. The students may use note paper during the exam. They may also use a calculator.
5. 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!
6. This exam contains 5 questions. Each question has its own code file. Students fill in their code in these code files, and submit them as their answers. Please submit five separate code files, and 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! If a student wants to add remarks to submitted code, please do so as comments in the code files that are submitted.
7. Students should add their name and student number at the top of each file as a comment.
8. Students may add extra tests in the main() function that is found in each of the code files.
9. 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.
10. Functions never need to ask the user for input. All inputs for the functions are supplied via parameters.

11. 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. Test the file before submitting it, to make sure that nothing went wrong after copying!
12. Make sure that code is easy to understand. When in doubt, add comments.
13. Each question is worth 2 points. The points are added up to form the exam grade. Standardized rounding applies.
14. Not all exercises are equally difficult. We tried to order them from easy to hard, but this may vary between students.
15. Naturally, students may not consult other people during the exam, nor are they allowed to use digital devices to consult anything but the materials listed above.

You can start the exam now, good luck!

Exam Data Structures & Algorithms 2021/2022

Please read the exam instructions on the cover page carefully!

Note that academic fraud amounts to handing in code that you did not write yourself, either because you got someone to help you, or because you copied someone else's answers, or because you consulted internet sites which you are not allowed to consult during the exam!

However, you may use code which you find in the books or notebooks to base your answers on.

What academic fraud is and the fact that you are not allowed to commit it, has been pointed out to you now multiple times during the class, via a lecture, via an announcement, via a video, via the cover page of this exam, and right here on this page.

When academic fraud is detected, the exam board will be notified immediately. Consequences may be severe. Denial of knowledge of academic fraud will not be accepted as an excuse.

If you need to download the exam because you received this page on paper, then please go to the Canvas course, and find the exam (it will be either under Assignments or Quizzes). You find all the files there, and you can also submit your answers there.

Make sure that you place any `.txt` file which is provided in the same folder as where you edit the code for the third exercise. If you do not, your code may not be able to see them. In particular, if you use the Jupyter server for coding, upload them too.

If you have questions during the exam, you can ask them via the Discussion forum for the Canvas course. You may read each other's questions, but you may not answer them – this would actually be a violation of the rule that you cannot communicate with others during an exam. Leave the answering to the instructors.

We wish you good luck.

Working in notebooks

For your convenience, we have added a notebook file named `DS&A2122Exam.ipynb` to the template files. This file contains all the questions, and code blocks with the template answers already in them. You can upload this file to the Jupyter server, or load it in a local Jupyter server, and use it to create answers to the exercises. Remember that you should place the `transcript.txt` file which is needed for one of the exercises in the same folder as where you place the notebook.

Even if you work like this, **you must copy your code to the `.py` Python files and upload those as your answers**! Do not try to upload your notebook! We cannot grade a notebook.

Exercise 1: Approximate e

File: `E01_approximate_e.py`

One-line summary

In this exercise, you approximate the value of *e* by calculating a continued fraction.

Context

There are many different ways to express the number *e* (Euler's constant), which is 2.718281828459... One way is the following formula, which has an infinite number of fractions:

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{2}{3 + \cfrac{3}{4 + \cfrac{4}{5 + \cfrac{5}{\ddots}}}}}}$$

Recursively, you can express this as: `e = 2 + 1/cf(1)`

where `cf` stands for "continued fraction," which is defined as: `cf(n) = n + n/cf(n+1)`

For large `n`, this can be approximated by: `cf(n) = n+1`

Exercise description

Write a function `approximate_e()` which calculates the value of *e* using the formula given above. The function gets one parameter, which is the depth of the calculation (an integer valued zero or higher), i.e., the number of fractions that you include in the calculation. At the given depth, you approximate `cf(n)` by `n+1`. The function returns a float, which is the approximation of *e*.

At depth 0, you approximate `cf(1)` as `1+1`, thus e is approximated by `2 + 1/(1+1) = 2.5`

At depth 1, you approximate `cf(2)` as `2+1`, and `cf(1)=1+1/cf(2)`, thus e is approximated by `2 + 1/(1+1/(2+1)) = 2.75`

Etcetera. You may choose whether to implement this function recursively or iteratively. The depth will not be higher than 100, so a recursive implementation should work.

Examples

`approximate_e( 0 )` returns `2.5`
`approximate_e( 1 )` returns `2.75`
`approximate_e( 100 )` returns `2.7182818284590455`

Exercise 2: Substrings

File: `E02_substrings`

One-line summary

In this exercise you find in a list the string which has the most substrings which are also in the list.

Exercise description

The function `find_superstring()` gets one parameter: a list which contains strings. These strings only contain letters of the alphabet, and are all lower-case. Some of the strings in the list are substrings of other strings in the list. Your goal is to find the "superstring," which is the string which has most of the other strings in the list as substrings.

The function should return the "superstring." If there is more than one string which has the most substrings, then return the one earliest in the alphabet.

You are allowed to change the list in the function.

Examples

If the list is:

```
["broth","brother","chemotherapy","her","moth","mother","other","rot",
"smother","the"],
```

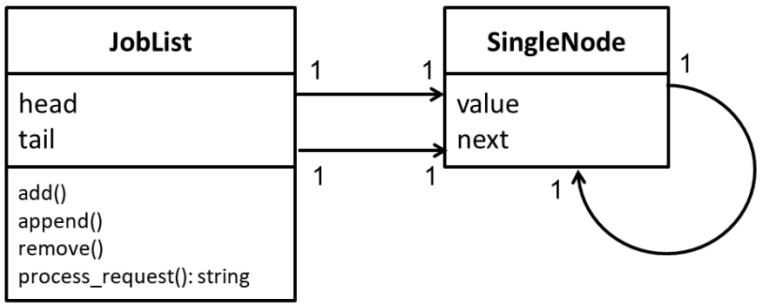
the superstring is: `brother`, which has five substrings (`broth`, `her`, `other`, `rot`, `the`). The strings `chemotherapy` and `smother` also have five substrings (`her`, `moth`, `mother`, `other`, `the`), but since `brother` is earlier in the alphabet than either of them, that is the string that should be returned.

Exercise 3: Process Jobs

File: `E03_process_jobs.py`

One-line summary

In this exercise you process commands to place jobs in a linked list and remove them again.



Exercise description

The template for this exercise contains a class `SingleNode` and a class `JobList`. `JobList` is a `SingleLinkedList`, exactly as it appears in the notebooks, consisting of `SingleNode`s. The only extra method in `JobList`, which you have to create, is `process_request()`.

`process_request()` gets three parameters (apart from `self`): `request` (string), `job` (string), and `priority` (string). There are three possible values for `request` (for which there are constants in the template file) namely “add”, “remove”, and “list”. There are two possible values for priority (also represented by constants in the template file), namely “low” and “high”.

If the `request` is “add”, you add the string contained in `job` to the linked list. If the `priority` is “low”, you add the `job` to the end of the linked list. If the `priority` is “high”, you add the `job` to the start of the linked list. The method returns a string “added: “ plus the job (see the example below). Note that when a high-priority job is added to the list, you can just attach it to the head – you do not have to insert it after all other high-priority jobs. Therefore you do not need to store the priority with a job.

If the `request` is “remove”, you remove one job from the list, namely the one at the start. The method returns a string “removed: “ plus the job that was removed (see the example below). If there were no jobs in the list, the method only returns the string: “removed: Nothing”.

If the `request` is “list”, the method returns the string: “list: “ plus all the jobs in the linked list, starting at the start, separated by commas (see example below).

Examples

```
jl = JobList()
jl.process_request( ADD, "banana", LOW ) returns “added: banana (low priority)”
jl.process_request( ADD, "apple", HIGH ) returns “added: apple (high priority)”
jl.process_request( LIST ) returns “list: apple, banana”
jl.process_request( REMOVE ) returns “removed: apple”
jl.process_request( REMOVE ) returns “removed: banana”
jl.process_request( REMOVE ) returns “removed: Nothing”
```

Exercise 4: Searching

Files: `E04_searching.py, transcripts.txt`

One-line summary

In this exercise, you will parse a large and complex text file to be able to search for strings.

Context

Large text files that contain "real world" data are typically messy and time consuming to search. You pre-process them to make searching faster.

Exercise description

The template for this exercise contains a class `Data`. When a `Data` object is created, you give it a filename (string). The class has two methods which you need to create: `parseFile()` and `search()`. You are allowed to make changes to the `__init__()` method, and to add extra methods.

`parseFile()` is called once, and pre-processes the file. It gets no parameters. It returns `False` if the file does not exist, otherwise it returns `True` (you do not need to check for exceptions). The preprocessing should make sure that you can search the file contents for words, consisting of letters from the alphabet. The letters can be upper-case or lower-case, and the search should be case-sensitive (so distinguish between upper-case and lower-case). The `parseFile()` method should have a time complexity which is at worst `O(n*log(n))`, with `n` being the number of separate words that can be found in the file.

After `parseFile()` has been called, you can search for words (which only consist of letters from the alphabet). You do this with the `search()` method. This method gets the `target` (string) that you are searching for as parameter. It returns `True` if the target can be found, and otherwise `False`. The search should be at worst `O(log(n))`, with `n` being the number of separate words that can be found in the file.

Note: the `parseFile()` method already contains a statement to open the file. It gets a special parameter with key "encoding" which indicates the encoding method of the file, otherwise it may give problems on some systems.

Examples

```
myData = Data( "transcripts.txt" )
if myData.parseFile():
    print ( myData.search( "sdafa" ) )       # False
    print ( myData.search( "illuminati" ))   # True
    print ( myData.search( "Masters" ))      # False
    print ( myData.search( "masters" ))      # True
```

Exercise 5: Determine Longest

File: `E05_determine_longest.py`

One-line summary

In this exercise you have to determine the time complexity of five functions which determine the longest word in a list of words.

Exercise description

Each of the functions, which are all named `determine_longest_`*x*`()` (*x* being a number), gets one parameter: `slist` which is a list of strings. The function returns the longest string in the list (if there are multiple, it will return one of them).

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 list.

Background information

You may assume the following:

- Taking a sublist from a list is `O(n)`, where `n` is the length of the sublist
- The list method `sort()` is `O(n*log(n))`, where `n` is the length of the list
- The statement `del list[i]` is `O(n)`, where `n` is the length of the list

Here are descriptions of the algorithms:

- `determine_longest_1()` processes each string in the list, keeping track of the longest.
- `determine_longest_2()` sorts the list by length, then returns the last element.
- `determine_longest_3()` compares the first two elements of the list and removes the shortest, until only one element remains.
- `determine_longest_4()` compares the first element of the list with the longest element of the rest of the list. The longest element of the rest of the list is determined by a recursive call to the function itself with the rest of the list as parameter.
- `determine_longest_5()` splits the list in two, and recursively calls itself with the two sublists. It determines the longest string that is returned from those two calls. If it gets called with a list with only one element, it returns that element.