Uppsala University 2021-03-19, 10:00–16:00

5LN715 Advanced Programming for Language Technologists

Instruction: The purpose of the exam is to assess your programming skills and creativity while working individually. You are thus not allowed to interact (cooperate, communicate, etc.) with other people (especially course mates) during the exam. However, you are allowed to use resources available online. By handing in your solution you certify that it is your own personal work, reflecting your individual understanding and skills. Your honesty and integrity are crucial for a correct assessment of your study results; you will be further examined along these dimensions in future courses.

Please adhere to the exam guidelines outlined below:

- submit your answers and code as part of `exam21.py`, provided to you in this assignment package
- edit and implement Python code where necessary, otherwise use multiline comments
- only short answers are necessary
- answers should be exact and concise
- use of Python 3 is presupposed
- use of consistent indentation and formatting is mandatory

The questions in this exam are intended to be clear, well-defined, and straightforward. However, if some task seems ambiguous or open to you, solve it in the way you find most reasonable (given the course content), and write a note about how you interpreted the instruction. We will also monitor the `advprog@cl.lingfil.uu.se` email address throughout the duration of the exam. You are welcome to email us to ask for clarifications, but please don't reveal the solutions you have in mind.

The grade thresholds are as follows: G: 50 % and VG: 75 %. (The examination of the course also involves a series of assignments, as you know.)

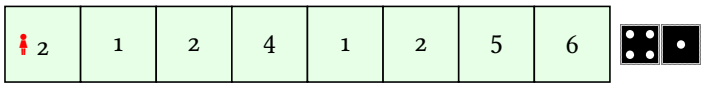
Good Luck!

1 Question 1 [max 2 + 4 + 4 p.] Terrain race with dice

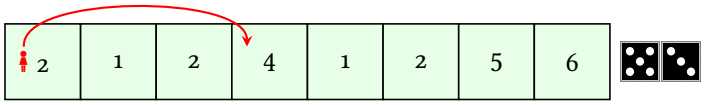
We are doing an abstract race on a "terrain" that is just a list of numbers. A higher number indicates a terrain that is harder to pass.

Each turn we throw two 6-sided dice to see how far we should move. The first die says how many steps we maximally can advance to the right. The second die indicates the highest "terrain" we are allowed to pass. If there is a higher number in the way we need to stop there.

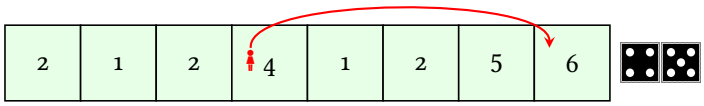
In an example like



we are at the start of the race. The first die means we can go at most 4 steps, but the second die being 1 means we can't pass the 2, so we will not advance at all this turn. Next turn:



We can only move three of the 5 steps. With the 3 we can pass 2, 1, 2, but not 4. Next turn:



We can move 4 steps, to the space with a 6. This time there are two reasons for stopping where we stop. We have no more steps to take, *and* we can't pass that 6 anyway.

So after that we have almost finished the race, just waiting for any throw where the second die is 6, so we can pass that final hurdle.

The class `TerrainRace` is for a race like this. The terrain is just a list of numbers, like `[2, 1, 2, 4, 1, 2, 5, 6]` for the example above. We are interested in how many turns it takes to finish the race, so the class has a method `number_of_moves` for that, and `ave_number_of_moves` that should compute an average number of turns, when this is done several times.

```
15 from random import randint
16
17
18 class TerrainRace:
19     def __init__(self, terrain, debug=False):
20         self.terrain = terrain
21         self._pos = 0
22         self.debug = debug
```

```
23
24      def _move(self):
25          move_length, move_strength = self._random_move_parameters()
26          if self.debug:
27              self._show_position(move_length, move_strength)
28          while (self._pos < len(self.terrain)
29                 and move_length
30                 and move_strength >= self.terrain[self._pos]):
31              move_length -= 1
32              self._pos += 1
33
34      def _random_move_parameters(self):
35          return randint(1, 6), randint(1, 6)
36
37      def _show_position(self, d1, d2):
38          for i, t in enumerate(self.terrain):
39              if self._pos == i:
40                  print('*', end='')
41              else:
42                  print(' ', end='')
43              print(t, end='')
44          print(f'   {d1} {d2}')
45
46      def number_of_moves(self):
47          self._pos = 0
48          counter = 0
49          while self._pos < len(self.terrain):
50              counter += 1
51              self._move()
52          return counter
53
54      def ave_number_of_moves(tries=10000):
55          total = 0
56          for _ in range(tries):
57              total += self.number_of_moves()
58          return total/tries
```

(This is an extract from `exam21.py` as it's given to you.) The code is supposed to be used like this:

```
>>> terrain = [2,1,5,1,1,4,2,5,1,1,1,1]
>>> race = TerrainRace(terrain)
>>> race.number_of_moves()
7
>>> race.number_of_moves()
4
>>> race.debug = True
>>> race.number_of_moves()
*2 1 5 1 1 4 2 5 1 1 1 1   2 4
 2 1*5 1 1 4 2 5 1 1 1 1   2 6
 2 1 5 1*1 4 2 5 1 1 1 1   4 1
 2 1 5 1 1*4 2 5 1 1 1 1   4 5
```

```
 2 1 5 1 1 4 2 5 1*1 1 1   6 5
5
>>> race.debug = False
>>> race.ave_number_of_moves()
7.7329
>>>
```

(a) That last thing with `ave_number_of_moves` doesn't work as it says in the example, though. (Try it!) Fix the definition of that method!

(b) A variant is to not throw *two* dice but just *one*, and use the *same* value both to see how many steps we may move *and* what terrains we can pass.

Write a class `SingleDieTerrainRace` with the existing `TerrainRace` as its superclass. Inherit most things and only change what needs to be changed.

The resulting code could be used like this:

```
>>> srace = SingleDieTerrainRace(terrain, True)
>>> srace.number_of_moves()
*2 1 5 1 1 4 2 5 1 1 1 1   5 5
 2 1 5 1 1*4 2 5 1 1 1 1   1 1
 2 1 5 1 1*4 2 5 1 1 1 1   4 4
 2 1 5 1 1 4 2*5 1 1 1 1   5 5
4
>>>
```

(c) Write another subclass of the existing `TerrainRace`, this one called `HardTerrainRace`. This has the special rule that at the end of each move we must back one step (if we haven't finished the race already). If we are at the start of the race nothing special happens.

Here is an example:

```
>>> lrace = HardTerrainRace(terrain, True)
>>> lrace.number_of_moves()
*2 1 5 1 1 4 2 5 1 1 1 1   6 1
*2 1 5 1 1 4 2 5 1 1 1 1   4 1
*2 1 5 1 1 4 2 5 1 1 1 1   6 6
 2 1 5 1 1*4 2 5 1 1 1 1   5 5
 2 1 5 1 1 4 2 5 1*1 1 1   3 3
5
>>>
```

2 Question 2 [max 4×2.5 p.]

Determine the worst case time complexity for each code snippet below, and support your answer with a short argument. Assume that x and y and are all arrays of integers. You can refer to their sizes via n and m, respectively.

(a)

```
for i in x:
    for j in y:
        if j > 3:
            for k in y:
                k = k // 2
        else:
            break
```

(b)

```
for idx in range(len(x)):
    print(idx)
for i in x:
    for j in y:
        if j == i:
            j = j ** 2
```

(c)

```
n = len(x)
while n > 0:
    for j in y:
        print(j)
    n = n // 2
```

(d)

```
len_y = len(y)

for i in range(len_y):
    for j in range(len_y):
        for k in range(len_y):
            y[i] = i * j
            y[j] = j * k
            y[k] = k * i
```

3 Question 3 [max 10 p.]

Implement methods for `LinkedListStack`, a Linked-list based Stack via the specification below, which is an extract from `exam21.py`. You should assume that the `Node.data` is an integer. You can add attributes to the `LinkedListStack` constructor to make things easier. All operations should be done in $O(1)$ time.

```
81  class Node:
82      def __init__(self, data):
83          self.next = None
84          self.data = data
85
86
87  class LinkedListStack:
88      def __init__(self):
89          self.head = None
90
91      def isEmpty(self):
92          """
93          Returns True if list is empty; False otherwise
94
95          :return: bool
96          """
97
98      def push(self, item):
99          """
100         Pushes 'Node' item on top of stack
101
102         :parameter:item: 'Node' being pushed
103         :return: None
104         """
105
106     def pop(self):
107         """
108         Pops 'Node' off the top of the stack;
109         throws error if stack is empty
110
111         :return: Node
112         """
113
114     def peek(self):
115         """
116         Returns Node on top of stack without removing it;
117         Returns None is stack is empty
118
119         :return: Node
120         """
121
122     def size(self):
123         """
124         Returns the number of nodes that comprise the stack
```

```
125
126         :return: int
127         """
```

4 Question 4 [max 10 p.]

Consider the following situation: you are an IT technician working at a telecommunications company that maintains internet, telephone and cable TV network accounts for millions of customers. You are tasked with implementing a system that connects customers to customer service representatives via a call-in number. Typically, customers call in and are asked to select an pre-determined option that best describes the nature of their call. They are placed on hold until a representative becomes available to address them. Since the call volume is high, each customer call is coded with a *rank*, depending which option they choose. This determines the severity of their inquiry or complaint.

The system you are tasked with implementing should mainstate a call volume state, which keeps track of every customer on hold. It should also determine which customer gets connected to a representative first, given their call rank. A typical call order (in succession) might look like the following:

- Customer `A`: Calling about upgrading their service to premium; `RANK 1`
- Customer `B`: Calling about concern relating to last month's bill; `RANK 2`
- Customer `C`: Internet connection is down, has important video conferencing call in one hour; `RANK 3`
- Customer `D`: Calling about adding a TV channel package to their account; `RANK 1`

At the time of calling, no representative is available to take these customers' calls. However, as representatives eventually do become available, they connect to the customers in the following order: `C, B, A, D`. Your system should be able to add customers to the state and also connect them with representatives given the severity of their rank, as above.

Discuss at least two different options for data structures that you could use for solving this task. Try to come up with as efficient data structure options as you can. Describe how you would store the information using each data structure. Then, go through all operations needed for storing the data and printing the asked for information. For each operation, discuss what the time complexity is of that operation with each of your data structure options. Note that the exact operations might be slightly different for the two data structures. Also note that it is possible, but not necessary, that your proposed data structure solutions use more than one data structure each. Finally, note that you do not have to write any code here; only describe how you would have organised the data structures used in your code, as well as time complexity issues.

5 Question 5 [max 10 p.]

Consider the raw text file `trout.txt` (provided to you on Studium). Implement a function called `read_and_sort`, which takes this file as input, tokenizes it, and writes a new file to disk. This new file should be called `trout.txt.probs`. It should contain all *alphabetic* tokens, their frequencies, and unigram probabilities, sorted by probability and separated by tabs. The output file should look similar to this:

```
token     freq      prob
trout     283       0.02024175666976611
like      194       0.013875974536871468
said      190       0.013589871969100923
fishing   162       0.011587153994707102
america   133       0.009512910378370647
creek     99        0.007081038552321007
went      98        0.007009512910378371
old       83        0.005936628281238824
good      83        0.005936628281238824
...
```

You are allowed to use whatever tokenization method you see fit, though `spacy` is recommended. You are also allowed to use dictionaries or `collections.defaultdict` for tracking frequencies, but not `collections.Counter`. You should also filter out stopwords in your file. These are also provided to you in a separate text file, `stopwords.txt`.

6 Question 6 [max 5 × 2 p.]

Consider the `numpy.array` *A* below:

```
A = np.array([[22, 28, 52],
              [87, 12, 76],
              [44, 61, 81],
              [97,  4, 67],
              [52, 14, 24],
              [34, 82,  7]])
```

1. How can one square every element of *A*, and extract the maximum value that is returned (`9409`)?

2. How can one get the indices of the minimum value per row, and then compute the sum of those values (`8`)?

3. Consider a new array *B* below. How can one combine A and B column-wise (so that first row is `[22, 28, 52, 88, 41, 22]`), and get the average per column (`[56.0, 33.5, 51.2, 34.3, 36.0, 44.3]`)?

```
B = np.array([[88, 41, 22,  1, 14, 70,  7, 48, 64],
              [60, 10, 17, 34,  5, 57, 16, 98, 36]])
```

4. Permute B accordingly and join it to the end of A (leaving you with 12 rows). Call this array C. Now, create another array, with the same shape as A, consisting of random numbers ranging between 1 and 99. Join this to the end of C, like you did with B. You should end up with 18 rows.

   How can one make a `pandas` data frame, using C as input data, with the column names `X, Y, Z`? Show all of your intermediate code.

5. How can one visualize the contents of the data frame in a scatter plot, similar to the one shown here? Note that your plot will not look identical to this one, due to the random number generation. The title, however, should remain the same.