```
54      def ave_number_of_moves(self, tries=10000):
```

**1a.** Correct. 2 2

```
75  class SingleDieTerrainRace(TerrainRace):
76
77      def _move(self):
78          move_length = self._random_move_parameters()
79          move_strength = move_length
80          if self.debug:
81              self._show_position(move_length, move_strength)
82          while (self._pos < len(self.terrain) and move_length and move_strength >= self.
                ↪ terrain[self._pos]):
83              move_length -= 1
84              self._pos += 1
85
86      def _random_move_parameters(self):
87          return randint(1, 6)
```

**1b.** This works, but is not ideal, since you were asked to only change what needs to be changed, but this has a lot of duplication from the original class. 2 2

```
61  class HardTerrainRace(TerrainRace):
62      def _move(self):
63          move_length, move_strength = self._random_move_parameters()
64          if self.debug:
65              self._show_position(move_length, move_strength)
66          while (self._pos < len(self.terrain)
67                 and move_length
68                 and move_strength >= self.terrain[self._pos]):
69              move_length -= 1
70              self._pos += 1
71              if move_length == 0:
72                  self._pos -= 1
```

**1c.** Ideally this should use `super` so you don't have to copy so much code from the original code. But also this doesn't work. See this example:

```
>>> t = HardTerrainRace([2,2,2,1,4], True)
>>> t.number_of_moves()
*2 2 2 1 4   3 6
 2 2*2 1 4   2 6
 2 2 2*1 4   5 1
 2 2 2 1*4   1 1
 2 2 2 1*4   5 3
 2 2 2 1*4   4 2
 2 2 2 1*4   4 2
 2 2 2 1*4   6 6
8
```

It backs up one step after the first two moves, but not after the later ones. 2 2

```
"""

A. O(nm^2) - the first for loop is the array x so that's going to be variable n,
the next for loop is array y so that's going to be m and inside that one has another
for loop traversing through the same array y, this will be another m. the inner-most
for loop is going to have a constant of 1/2 yielding a more accurate big O as O(n*(m^2/2),
but that 1/2 constant doesn't matter, so we're left with n(m^2)

B. O(nm) - the top for loop is going over array x (variable n). the outer for loop below is
going over the same, n, while the inner for loop is going over array y (variable m). the more
accurate big O, I suppose is O(n+nm) / factored out as O(n(1+m)), but the nm is a higher order so we don't care about
the lone n.

C. O(mlogn) - the while loop is traversing over array x (variable n) and the for loop is traversing
array y (variable m). But the while loop has n = n//2 which will give it a logarithmic complexity,
making it log n. so in total we have mlogn

D. O(m^3) - this one has all 3 for loops traversing through the same array, array y (as variable m).
this yields m^3. the statements in the inner-most for loop with reassigning the array doesn't add to the
complexity.

"""
```

**2a** 2.5 2.5
**2b** 2.5 2.5
**2c** 2.5 2.5
**2d** 2.5 2.5

```
121 class Node:
122     def __init__(self, data):
123         self.next = None
124         self.data = data
125 
126 
127 class LinkedListStack:
128     def __init__(self):
129         self.head = None
130         self.count = 0
131 
132     def isEmpty(self):
133         # will see if there's nothing in the first node/head
134         if self.head is None:
135             return True
136         else:
137             return False
138         """
139         Returns True if list is empty; False otherwise
140 
141         :return: bool
142         """
143 
144     def push(self, item):
145         # What I'm doing is under the assumption I got from
146         # the instructions which is that the newest node
147         # created will be sent to the end of the linked list.
148         # The wording threw me off a bit.
149 
150         if self.head is None:
151             self.head = Node(item)
152         else:
153             new_node = Node(item)
154             new_node.next = self.head
155             self.head = new_node
156         self.count += 1
157         """
158         Pushes 'Node' item on top of stack
159 
160         :parameter:item: 'Node' being pushed
161         :return: None
162         """
163 
164     def pop(self):
165         # can call isEmpty method instead of rewriting same
166         # conditionals to check if the linked list is empty
167         if self.isEmpty():
168             return 'Error'
169         else:
170             delNode = self.head
171             self.head = self.head.next
```

```
172             delNode.next = None
173             self.count -= 1
174             return delNode.data
175         """
176         Pops 'Node' off the top of the stack;
177         throws error if stack is empty
178 
179         :return: Node
180         """
181 
182     def peek(self):
183         # do same isEmpty check first as before
184         if self.isEmpty():
185             return None
186         # just return the top node's data as it'll be the
187         # self.head here
188         else:
189             return self.head.data
190         """
191         Returns Node on top of stack without removing it;
192         Returns None is stack is empty
193 
194         :return: Node
195         """
196 
197     def size(self):
198         return self.count
199         """
200         Returns the number of nodes that comprise the stack
201 
202         :return: int
203         """
```

**3** Excellent. **10** **10**

```
"""

Write your answer here.

I would use 1 outer dictionary as my main access point to the rest of the data. I believe this will work with the caveat
that the telecommunications company has either a finite number of rankings, or if they come up with a new ranking then
update the dictionary with it, as you'll see later on.
The length of the dictionary, meaning the number of keys minus 1, would be predetermined by the number of rankings the
telecommunications company has. So if they have rankings 1,2,3,4, and 5 then the the keys of the dictionary would be the
same, (1,2,3,4,5). Next the values of these keys would be structured with singly linked lists , with the self.head in
each singly linked list constantly being updated wth the customer next in line. The linked list will be filled out in
order of first come first serve FOR SPECIFICALLY THAT RANKING, not just for ALL customers with varying rankings. And
just as the example presented in the instructions portrayed, if there are two customers in rank 3 (which is the highest
ranking defined by the telecommunications company), one customer with rank 2, and one customer with rank 1 then the
operators will be give both customers in rank 3 priority over the customers with rank 2 and 1, and the customers with
rank 3 will be dealt with in order of which called in first, as customers are added to the tails of linked lists as they
come in.
Operations of storing data and printing are as follows:
customers with ranking are fed into the dictionary and their ranking is matched against the dictionary's keys, then the
customer is added to the tail-end of the linked list corresponding to that specific key. Now, once a customer is taken
care of, in any linked list, they are removed as the self.head of that linked list and the self.head.next is assigned as
the new self.head, continuously updating as callers are dealt with or if they drop from the call. All customers in the
highest priority ranking (in this case rank 3) must be dealt with before allocating representatives to lesser rankings.
This can be accomplished with a simple isEmpty() method, commonly made in linked list classes (or even from the previous
question in this exam), for the highest ranking key's value's linked list.
For time complexities:
Insertion and deletion of nodes in the singly linked list both have average and worst case time complexities of O(1)
which is nice for this example. The dictionary's time complexity on average is O(1) as well as it functions as a hashmap
in python, requiring the inner parameters to be has functions. This is advantageous as other times collisions can occur
and it'd make the worst time complexity be O(n). Next, the operation of the previously mentioned isEmpty() method to
```

```
check if the higher ranking's linked list has customer waiting is a constant. All in all, I think this is a
semi-efficient solution with a minimized time complexity.

Another thing I thought of which might be cool, but not mentioned by the question, is if the self.head customer is
talking to a call representative and their call is deemed a different ranking as the customer and representative talk.
In such a case, the customer put back into the stream of data inputted to the dictionary and put at the tail end of the
newly designated, appropriate ranking.

Solution 2:
A doubly linked list can be used instead of a singly linked list as the pointer to the previous node can be utilized
effectively in this example. Also, doubly linked lists can be used to implement binary trees unlike singly linked lists.
I feel as thought this would be advantageous for something like a call center, or a telecommunications company in our
case, to account for further features than proposed than in this question, such as calls dropped or elevation of
ranking. The customers are inputted with their ranking in the conventional .data parameter. A isEmpty() will be used
again here for the same purpose. This is constant time so we don't mind that in the grand scheme of things. Now, once
the doubly linked list is not empty, the next newly created Node will be appended and it's .data parameter (its
ranking) will be compared against the previous node's data and if it's greater (higher priority as defined by the
company) then its .prev parameter will point to the .prev.prev's node and in turn the .prev.prev's .next parameter will
point to the the node at hand. And the node at hand's .next will point to the former .prev (the node that began the
comparison. This comparison will happen until the node at hand meets attains self.head status, or reaches a node with
the same ranking priority as its data parameter. Additionally, this also extends to if the node at hand has a lower
rank priority it will stay at the tail end of the doubly linked list, pointing to None, until more new Nodes are
created for further comparison, etc. And similarly to solution 1, the customer/node at the self.head position is deleted
once dealt with and prompts the self.head.next Node to become the first node/self.head .
For time complexities:
Initially inserting the customers with their rankings into the doubly linked list will have at worst O(1) (this is the
same at best, too). Deleting the customer after they're dealt with also has a wost time complexity of O(1). However,
searching through the doubly linked list for a node to compare its ranking data will come with it a time complexity of
O(n) as it must traverse through the nodes one at a time (or 'n' at a time).


```

```
270 The same idea of reassigning customer ranking applies to this solution, but instead of the
        ↪ data reentering the input
271 stream seen in solution 1, it will be either elevated or demoted to a position closer or
        ↪ further away from the self.head
272 iff (if and only if) it's ranking is reassigned by the representative. This would cause O(n)
        ↪ in the worst case.
273
274 """
```

**4** Great discussion overall; good join. A small comment: I think what you're describing in the first method, in so many words, is a queue no? **10** **10**

```python
def read_and_sort():
    from spacy.lang.en import stop_words, English

    nlp = English()

    # trout file
    with open('trout.txt') as trout_file:
        raw_text = trout_file.read().lower()
    # add stopwords to nlp's defined stop word vocabulary
    with open('stopwords.txt') as stopwords_file:
        lines = stopwords_file.read().split()
        # print(lines)
        for word in lines:
            term = nlp.vocab[word]
            term.is_stop = True

    tokenizer = nlp.tokenizer
    doc = nlp(raw_text)
    tokenized_words = [tok.text for tok in doc if tok.is_stop == False if tok.is_alpha]
    # tokenized_words

    from collections import defaultdict
    r = defaultdict(int)
    for token in tokenized_words:
        r[token] += 1

    values = r.values()
    # Return values of a dictionary
    total = sum(values)
    print("Token\tfreq\tprob")

    with open('trout.txt.probs', 'w') as file:
        file.write("Token\tfreq\tprob\n")
        for key, value in r.items():
            prob = value / total
            file.write(f"{key}\t{value}\t{prob}\n")
```

5 Code looks great. Two minor things you didn't do, however: 1. sort the output by descending order of probability and 2. add the input file as an argument to the function. 6.5 6.5

```
322 ##########################################################################
323
324 import numpy as np
325 import pandas as pd
326 import matplotlib.pyplot as plt
327 import seaborn as sns
328
329 A = np.array([[22, 28, 52],
330               [87, 12, 76],
331               [44, 61, 81],
332               [97, 4, 67],
333               [52, 14, 24],
334               [34, 82, 7]])
335
336 B = np.array([[88, 41, 22, 1, 14, 70, 7, 48, 64],
337               [60, 10, 17, 34, 5, 57, 16, 98, 36]])
338
339 # Question 6.1
340
341 # Write your code here.
342 arr = np.max(np.power(A,2))
343 print(arr)
344 # Question 6.2
345
346 # Write your code here.
347 summation = np.sum(A.argmin(axis=0))
348 print(summation)
349 # Question 6.3
350
351 # Write your code here.
352 B = np.reshape(B, (6,3))
353 # print(B)
354 #np.concatenate(np.reshape(A,B),axis=1)
355 X = np.concatenate((A,B), axis=1)
356 print(np.mean(X, axis = 0))
357 # Question 6.4
358
359 # Write your code here.
360 C = np.concatenate((A,B))
361 Y = np.random.randint(1,99, size=(6,3))
362 C = np.concatenate((C,Y))
363 #[random.uniform(low,high) for _ in range(size)]
364 #np.random.Generator.uniform(low=0.0, high=1.0, size=None)
365 # Question 6.5
366
367 # Write your code here.
368 import pandas as pd
369 df = pd.DataFrame(data=C, columns = ['X', 'Y', 'Z'])
370 print(df)
371 import seaborn as sns
372 import matplotlib.pyplot as plt
```

```
373 sns.scatterplot(data=df, x='X', y='Y', size = 'Z')
374 plt.title("A Very Informative Plot")
375 plt.show()
```

**6.1** 2 — 2
**6.2** 2 — 2
**6.3** 2 — 2
**6.4** 2 — 2
**6.5** 2 — 2

Total: 52.5 / 60 = 88 %, VG