CS1010E Practical Exam I

Instructions:

- If you failed to keep your proctoring camera, your PE will result in **ZERO mark** no matter what you submitted onto Exemplify or Coursemology.
- You ***cannot import*** any additional packages or functions, or you will get zero mark.
- This PE consists of **THREE** parts and each part should be submitted seperately in Examplify and Cousremology. It's better for you to organize your files into `part1.py`, `part2.py` and `part3.py` and each file is independent, e.g. you cannot expect that the answer in `part2.py` calls a function in `part1.py` that is not in `part2.py`. If you want to reuse some function(s) from another file/part, you need to copy them (as well as their supporting functions) into that file/part. For example, you just need to copy the function you want to call in `part1.py` into `part2.py`.
- You are allowed to write extra functions to structure your code better. However, remember to submit together mentioned in the point above.
- No mark will be given *if your code cannot run*, namely any syntax errors or crashes.
    - We will just grade what you have submitted and we will **not** fix your code.
    - Comment out any part that you do not want.
- Workable code that can produce correct answers in the given test cases will only give you *partial* marks. Only good and efficient code that can pass ALL test cases will give you full marks. Marks will be deducted if your code is unnecessarily long, hard-coded, or in poor programming style, including irrelevant code or test code.
- You must use the same function names as those in the template given.
- Your code should be efficient and able to complete *each* example function call in this paper within 2 seconds.
- In all parts, you should **return** values instead of **printing** your output. In another word, you should not need any "`print()`" in this entire PE for submission.
- You should either delete all test cases before submission or comment them out by adding # before the test cases. If you submit more code than required, it will result in **penalty** because your code is "unnecessarily long".
- You should save an **exact** copy of your submission in your computer for submitting in Coursemology again ***after*** the PE. Any differences between Exemplify and Coursemology submissions will be severely penalized, except for indentation differences.
- Reminder: Any type of plagiarism like copying code with modifications will be caught, that include adding comments, changing variable names, changing the order of the lines/functions, adding useless statements, etc.

Constraints for Part 1

You cannot use the following Python ***built-it*** functions/methods in Part 1:

- `encode, decode, replace, partition, rpartition, split, rsplit, translate, map, filter.`

Also, your code cannot be too long. Namely, the function in *each* task must be fewer than 400 characters and 15 lines of code including all sub-functions that called by it.

You cannot use any list, tuple, dict or set and their functions in this part.

Part 1 Run-length Decoding (15 + 15 = 30 marks)

Run-length *encoding* (RLE) is a form of lossless data compression in which runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This technique is extremely useful especially in data compression.

Assuming we have a string with ASCII characters. Each run of consecutive data item is represented as that item with its count. Here is an example of RLE.

`'AAAAAAAAAHHHEEM'` → `'A9H3E2M1'`

You can see that the string with 15 characters on the left is converted to a string with 8 characters only.

However, we want you to write a function to perform the reverse, namely, run-length decoding (RLD) as the following example:

`'A9H3E2M1'` → `'AAAAAAAAAHHHEEM'`

In this part, you can assume that the input is a string with even number of characters. Each ASCII character is followed by its count in an integer between 1 to 9. So, you will not have an input like `'B11C2'`.

Task 1 Iterative Run-length Decoding (15 marks)

Write an ***iterative*** version of the function `RLD_I(s)` to decode a string `s` mentioned above. In this task, you cannot use any recursion. Here is some sample output.

```
>>> print(RLD_I('H3e2l3o1W1o3r4l2d1'))
HHHeellloWoooorrrrlld
>>> print(RLD_I('A9H3E2M1'))
AAAAAAAAAHHHEEM
```

Task 2 Recursive Run-length Decoding (15 marks)

Write a ***recursion*** version of the function `RLD_R(s)` with the same functionality in Part 1 Task 1. However, you cannot use any loops or list comprehension in this task.

Part 2 Divisors (20 + 10 + 5 = 35 marks)

Given an integer N > 0, d is a divisor of N if N is divisible by d. A divisor is also called a factor. E.g. 7 is a divisor of 28. N can have many divisors. The ***number of divisors of N*** counts how many different divisors does N has. For example, if N = 16, it has 1, 2, 4, 8 and 16 as its divisors. So the number of divisors of 16 is 5.

N	Divisors	Number of Divisors of N
1	1	1
2	1,2	2
3	1,3	2
4	1,2,4	3
5	1,5	2
6	1,2,3,6	4

Task 1 List of Numbers of Divisors (20 marks)

Write a function `num_divisor_list(N)` to output a list `L` that contains the numbers of divisors from 0 to N. Each item `L[i]` in `L` contains the number of divisors of $i \leq N$. Here are some sample outputs.

```
>>> print(num_divisor_list(6))
[0, 1, 2, 2, 3, 2, 4]
>>> print(num_divisor_list(10))
[0, 1, 2, 2, 3, 2, 4, 2, 4, 3, 4]
```

For simplicity, we assume `L[0]=0`. Namely, we claim that there is no divisor for the integer 0.

Task 2 Who has/have the Max Number of Divisors? (10 marks)

Given an integer N, what is/are the numbers that has/have the most number of divisors? For example, from the sample outputs above, you can see that if N = 10, there are three numbers that have their number of divisor equals to 4 as the maximum of the list. So the numbers that have the most number of divisors under 10 are 6,8 and 10. Your output must be a list in the ascending order. Again, here are some sample outputs:

```
>>> print(list_of_max_divisor(10))
[6, 8, 10]
>>> print(list_of_max_divisor(25))
[24]
>>> print(list_of_max_divisor(98))
[60, 72, 84, 90, 96]
```

Task 3 How Many Prime Numbers? (5 marks)

Given an integer N, how many prime numbers are smaller than or equal to N? E.g. if N = 25, the primes numbers ≤ N are 2, 3, 5, 7, 11, 13, 17, 19 and 23, so the answer is 9. In this question, the integer 1 and 0 are not considered as prime numbers.

```
>>> print(how_many_prime(25))
9
>>> print(how_many_prime(100))
25
>>> print(how_many_prime(1000))
168
```

Part 3 Alice's Magic Height Potion Treatments (25 + 10 = 35marks)

Remember Alice in Wonderland? She drank a potion and made her shrink. The magic potion can change her height!



Now, these magic potions are available for everyone! And not only by shrinking, it can also make you grow taller! However, because of the limitation of our current "technology", we only have two types of potions now:

- Potion A: Make someone to grow 1 cm taller
- Potion B: Make someone's height shrink by half, but this potion is **only effective** if the height is an even number in cm.

Let's have an example, if Alice is 4 cm tall now, and she wants to change her height to 3 cm, we can either

- Drink one Potion B (4 cm □ 2 cm) then one Potion A (2 cm □ 3 cm), or
- Drink two Potions A (4 cm □ 6 cm) then one Potion B (6 cm □ 3 cm).

The first choice needs ***two*** potions and the second choice needs ***three*** potions.

However, the potions are still new, we tried not to take as many as possible because we are afraid if there is any side effect. So in the above example, the optimal way to take the potions is the first choice (Potion B then Potion A).

Task 1 Magic Potion Treatment (25 marks)

Given the initial and final heights as two integers `h1` and `h2` in cm respectively, write a function `magicPotionTreatment(h1,h2)` to return the string of the potion process with the minimal number of potions that change the height from `h1` to `h2`. Here are some sample output:

```
>>> print(magicPotionTreatment(4,3))
BA
>>> print(magicPotionTreatment(9,2))
ABABAB
>>> print(magicPotionTreatment(123,5))
ABBABBBA
```

You can assume the inputs are non-zero positive integers in cm.

Task 2 Long Magic Potion Treatment (10 marks)

Optimize your function that you can finish computing the treatment with inputs as large numbers within 2 seconds.

```
>>> print(magicPotionTreatment(134217729,3))
ABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
>>> print(len(magicPotionTreatment(10**100+17,11)))
555
```

Code Template

Part 1

```
'''
Please be reminded that the followings
are not allowed in this part
- import packages
- Use these functions: encode, decode, replace, partition,
  rpartition, split, rsplit, translate, map, filter.
- List comprehension
- Dictionary and set
'''
def RLD_I(s):
    return ''
def RLD_R(s):
    return ''
# You should comment out the following when you submit your code
#print(RLD_I('H3e2l3o1W1o3r4l2d1'))
#print(RLD_I('A9H3E2M1'))
```

template for the next two parts are on the next page...

Part 2

```
'''
Please be reminded that the followings
are not allowed in this part
- import packages
'''
def num_divisor_list(N):
    return []
def list_of_max_divisor(N):
    return []
def how_many_prime(N):
    return 0
# You should commment out the following test
# cases when you submit your code to examplify
# and cousremology
'''
print(num_divisor_list(5))
print(num_divisor_list(10))
print(num_divisor_list(50))
print(list_of_max_divisor(10))
print(list_of_max_divisor(20))
print(list_of_max_divisor(50))
print(how_many_prime(10))
print(how_many_prime(20))
print(how_many_prime(100))
'''
```

Part 3

```
def magicPotionTreatment(h1,h2):
    return ''
# You should commment out the following test cases when you submit your code
#print(magicPotionTreatment(4,3))
#print(magicPotionTreatment(9,2))
#print(magicPotionTreatment(123,5))
```