CS1010E: Programming Methodology

PE1 (Saturday, 20 February 2021)

Instructions

Read **ALL** instructions carefully. Read **ALL** questions carefully first before attempting them.

1. **Submission Content**:
   - Each question will be given a template file (*e.g.*, `QuestionN.py`).
   - Comment out all the given **test cases** and `import` statements before submission.
   - If any implementation is given, you are not allowed to change them.
   - You are not allowed to *add*, *remove*, or *modify* any parameters from functions.
   - You are not allowed to change the *name* of the given function.
   - Unless otherwise stated, you are not allowed to add/change `import`.
   - General restrictions should not be violated or you will be **heavily penalised**.
2. **LumiNUS Submissions**:
   - You are NOT allowed to change the filename.
   - You should submit within the given 2 hours. NO additional time will be given.
   - You should submit to the correct folder with your name on LumiNUS.
   - Late submission will be **heavily penalised**.
3. **Coursemology Submissions**:
   - You are responsible for finalizing your submission to avoid being **heavily penalised**
   - You should not submit at the last minute since Coursemology is not designed to handle large workload.
   - If you see that Coursemology is *under maintenance*, it is due to large workload and not an actual maintenance. No extension will be given.
   - Your LumiNUS and Coursemology submission should be **identical**. Any changes will be heavily penalized.
   - There are no test cases on Coursemology and our testing will use test cases that are *different* from the sample runs given.
4. **Screen Recording Submissions**:
   - Failure to submit your screen recording, missing parts of screen recording or failure to keep your proctoring camera will result in your PE being marked **ZERO**.
   - Late submission will be **heavily penalized**.
5. **Testing**: *No change from usual instructions.*
6. **Good Programming Practice**: *No change from usual instructions.*
7. **Plagiarism Warning**:
   - Any form of plagiarism including (*but not limited to*) copying of code, viewing online source, *etc.*will be marked **ZERO** and reported to the University immediately.
     - Any attempt at modifying plagiarised code including (*but not limited to*) adding comments, changing variable names, changing order of lines/functions, adding redundant statements, *etc. will not be able to fool the system*.

Failure to follow any of the instructions above will result in deduction of your marks.

Files

- `PE1.pdf`
- `Question1.py`
- `Question2.py`
- `Question3.py`

Coursemology

- `PE1`

LumiNUS

- `PE1 File Submission`
- `PE1 Video Submission`

Questions

1. DNA Transcription
   - 1.1 Iterative Transcription [*20 marks*]
   - 1.2 Recursive Transcription [*20 marks*]
2. RNA Translation
   - 2.1 RNA Segment [*35 marks*]
   - 2.2 Polypeptide Property [*25 marks*]
3. Auspicious Numbers
   - 3.1 Simple Auspicious [*40 marks*]
   - 3.2 Efficient Auspicious [*10 marks*]

Question 1: DNA Transcription

Deoxyribonucleic acid (DNA) is the building block of life on Earth. DNA transcription is the first step in the creation of protein, essential for all living organisms on Earth. It involves copying a particular segment of DNA into ribonucleic acid (RNA). Afterwards, the RNA is translated into polypeptide which will eventually form proteins. The entire process can be summarised using the following image from Khan Academy.

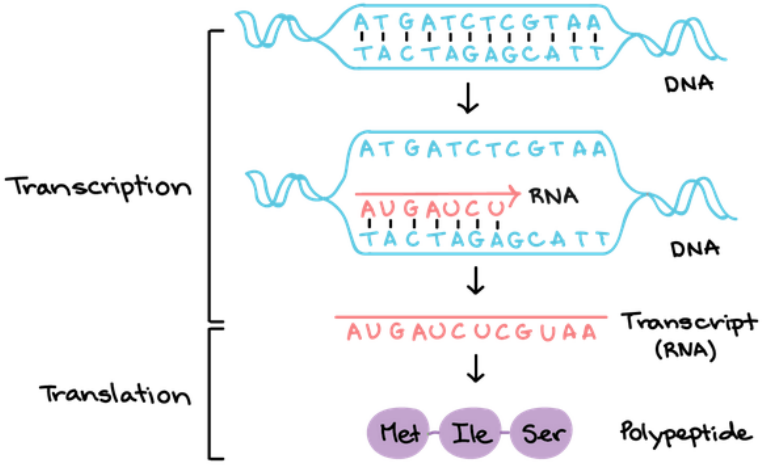

Figure 1: DNA transcription and RNA translation.

DNA is composed of 4 bases adenine, guanine, thymine and cytosine (coded as `A`, `G`, `T` and `C` respectively). In DNA, `A` is always paired with `T` and `G` is always paired with `C`. However, there is no `T` in RNA. It is replaced with uracil (coded as `U`) instead. This gives us the following pairing tables:

DNA Base	RNA Base Pair
A	U
G	C
T	A
C	G

continue on the next page...

General Restrictions

You are NOT allowed to use the the following Python ***built-in*** string (`str`) or list (`list`) methods/functions in Question 1:

- `encode()`
- `decode()`
- `replace()`
- `partition()`
- `rpartition()`
- `chr()`
- `split()`
- `rsplit()`
- `translate()`
- `map()`
- `count()`
- `ord()`
- `find()`
- `join()`
- `rfind()`
- `rsplit()`
- `rstrip()`
- `filter()`
- `index()`
- `strip()`
- `sort()`
- `sorted()`
- `pop()`

1.1 Iterative Transcription [20 marks]

Question

Write the *iterative* function `dna_transcription_I(dna)` to transcribe the given DNA segment (`dna`) given as a string (`str`) into the corresponding RNA as string (`str`).

Restrictions

- You may not use recursive function(s) to solve this.

Assumptions

- `dna` is in uppercase and will only be A, G, T or C

Sample Run #1

```
>>> dna_transcription_I('AGCTGACGTA')
UCGACUGCAU
```

Sample Run #2

```
>>> dna_transcription_I('AGCAGGACT')
UCGUCCUGA
```

Sample Run #3

```
>>> dna_transcription_I('AAGGCCTT')
UUCCGGAA
```

continue on the next page...

1.2 Recursive Transcription [20 marks]

Question

Write the *recursive* function `dna_transcription_R(dna)` to transcribe the given DNA segment (`dna`) given as a string (`str`) into the corresponding RNA as string (`str`).

Restrictions

- You may not use iterative constructs (*e.g.*, loop, list comprehensions, *etc.*) to solve this.
- The function `dna_transcription_R` must be *recursive* (*i.e.*, it calls itself). The use of any recursive helper functions will not be counted as being recursive.

Assumptions

- `dna` is in uppercase and will only be A, G, T or C

Sample Run #1

```
>>> dna_transcription_R('AGCTGACGTA')
UCGACUGCAU
```

Sample Run #2

```
>>> dna_transcription_R('AGCAGGACT')
UCGUCCUGA
```

Sample Run #3

```
>>> dna_transcription_R('AAGGCCTT')
UUCCGGAA
```

Question 2: RNA Translation

The next step is the the RNA translation. Unfortunately, the translation process is not that straightforward. RNA segments may contain junk that are not translated. Additionally, we are also interested in the overall biochemical properties of the polypeptide generated.

General Restrictions

You are NOT allowed to use the the following Python ***built-in*** string (`str`) or list (`list`) methods/functions in Question 2:

- `encode()`
- `decode()`
- `replace()`
- `partition()`
- `rpartition()`
- `chr()`
- `split()`
- `rsplit()`
- `translate()`
- `map()`
- `count()`
- `ord()`
- `find()`
- `join()`
- `rfind()`
- `rsplit()`
- `rstrip()`
- `filter()`
- `index()`
- `strip()`
- `sort()`
- `sorted()`
- `pop()`

2.1 RNA Segment [35 marks]

RNA translation will only start from what is called as the initiation codon until the termination codon. In our simplified simulation, there are two initiation codons: UU and UG. Furthermore, there are two termination codons: AC and AA.

The translation process will start from the *leftmost* initiation codon. This can be either UU or UG depending on their position in the RNA. Any RNA before the leftmost initiation codon will be ignored. Searching for the initiation codon will be done bases by bases. Therefore, the segment `AUUA` contains one initiation codon UU even though it does not start at even indices. However, during the translation process, the RNA will be read in groups of two bases. Since the translation is reading the RNA in groups of two bases, `UUGAAG` has no termination codon.

As soon as the translation process encounters any termination codons, the translation process ends. Any RNA after the first termination codon will be ignored. Note that given this restriction, the length of the RNA need not be even number. For instance, the RNA may be `AUGUUAUAAACUU` which has 13 bases. Normal translation will not work on this RNA.

However, since we only translate from UG up to AA, the translated regions are: `UGUUAUAA`. The rest of the RNA are ignored. There are other possible translated regions since there is UU and AC in the RNA but we are only interested in the leftmost initiation codon with its corresponding leftmost termination codon. You may assume that the translated regions will have even number of bases.

Question

Write a function `rna_segment(rna)` to return the translated region as string (`str`) from the given RNA (`rna`) given as string (`str`).

Assumptions

- `rna` is in uppercase and will only be A, G, U or C
- `rna` will have at least one translated region

Sample Run #1

```
>>> rna_segment('AUGUUAUAAACUU')
UGUUAUAA
```

Sample Run #2

```
>>> rna_segment('GUUGAAGUACAAAG')
UUGAAGUACAAA
```

2.2 Polypeptide Property [25 marks]

Each of the amino acid produced by the RNA can be categorised into non-polar, polar, basic or acidic. If the amino acid is categorised as either non-polar or polar, we assume that they are neither basic nor acidic. What we want to know the general property of the polypeptide by using the following category:

- `Acidic` : If there are more acidic amino acids than basic amino acids.
- `Basic` : If there are more basic amino acids than acidic amino acids.
- `Polar` : If neither `Acidic` nor `Basic` and there are more polar amino acids than non-polar amino acids.
- `Neutral` : If it does not fall into either one of the above categories.

Amino	Property	Amino	Property	Amino	Property	Amino	Property
F	*acidic*	S	*polar*	Y	*polar*	Q	*acidic*
L	*non-polar*	P	*acidic*	O	*non-polar*	C	*basic*
M	*basic*	T	*polar*	A	*basic*	R	*basic*

As such, the polypeptide `FM` is `Neutral` while the polypeptide `YSF` is `Acidic`. However, note that RNA translation is not perfect. Sometimes you get an unknown amino acid. In that case, you have to ignore them. For instance, the polypeptide `FXM` is still `Neutral` because the amino acid X is unknown. If everything is unknown, then we treat it as `Neutral`.

Question

Write a function `poly_property(poly)` that accepts a polypeptide (`poly`) as string (`str`) and returns the expected general property of the polypeptide based on the categories above.

Assumptions

- `poly` is in uppercase with length at least 1

Sample Run #1

```
>>> poly_property('FM')
Neutral
```

continue on the next page...

Sample Run #2

```
>>> poly_property('YSF')
Acidic
```

Sample Run #3

```
>>> poly_property('FXM')
Neutral
```

Question 3: Auspicious Numbers

In some culture, some digits are considered to be bad luck. For instance, the digit 4 is considered bad luck in the far east. When a digit is considered bad luck, some people do not want them to appear in any number such as phone numbers or license plate numbers.

3.1 Simple Auspicious [40 marks]

Instead of checking if a number is auspicious or not, what we want to do is *count* how many auspicious numbers greater than `0` are there with a given number of digits. If we consider the digit 4 to be bad luck, then there are 8 auspicious numbers with 1 digit: `1`, `2`, `3`, `5`, `6`, `7`, `8` and `9`. With 2 digits, we have 72 auspicious numbers shown below:

```
10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29,
30, 31, 32, 33, 35, 36, 37, 38, 39, 50, 51, 52, 53, 55, 56, 57, 58, 59,
60, 61, 62, 63, 65, 66, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 79,
80, 81, 82, 83, 85, 86, 87, 88, 89, 90, 91, 92, 93, 95, 96, 97, 98, 99
```

If we consider the digits 1 and 3 to be bad luck, then there are 7 auspicious numbers with 1 digit: `2`, `4`, `5`, `6`, `7`, `8` and `9`. With 2 digits, there are 56 auspicious numbers. Note that we do not consider the number `0` or any number with leading `0`s.

Question

Write the function `auspicious_number(n, bad)` that returns how many auspicious numbers are there with exactly `n` digits and do not contain any numbers given in the list (`list`) of integer (`int`) `bad`.

Assumptions

- `n > 0`, `0 < len(bad) < 9`
- Each entry in `bad` is *unique*

Sample Run #1

```
>>> auspicious_number(2, [4])
72
```

Sample Run #2

```
>>> auspicious_number(3, [4])
648
```

Sample Run #3

```
>>> auspicious_number(2, [1, 3])
56
```

continue on the next page...

Sample Run #4

```
>>> auspicious_number(3, [1, 3])
448
```

3.2 Efficient Auspicious [10 marks]

You will immediately get this mark if your answer for Question 3.1 can finish within 1 second for the very large inputs such as in sample runs below:

Sample Run #1

```
>>> auspicious_number(50, [4])
458113351761787849810187670902774464624095575112
```

Sample Run #2

```
>>> auspicious_number(1000, [1,3])%1000000000
951240704
```

Sample Run #3

```
>>> auspicious_number(100000, [4])%1000000000
39111112
```

– End of Paper –