Assignment 04

Programming with Python (for Bioinformatics)

Johanna Schmitz

05.06.2024; Submit before **18.06.2024, 23:59**

Assignment 04

General

For the next assignments, you can and should use **numpy**.
To install **numpy**, you simply do

```
mamba install numpy
```

in your **activated** pyprog environment.

Assignment 04 - Task 1

Assignment 04 - Task 1 (4 points)

In this task you will implement a **counting Bloom filter**. A counting Bloom Filter is similar to a Bloom filter, which is a space-efficient probabilistic data structure that may be used to test whether an element belongs to a set (with possible false positives, but no false negatives). Due to its properties, elements can only be added but not removed. In contrast, a counting Bloom filter allows insertion and deletion of elements. As long as only elements are deleted that were previously inserted, the counting Bloom filter can guarantee that the returned count of an element is an upper bound on its true count. The more elements were inserted into the counting Bloom filter, the higher gets the probability that the true count is lower than the filter suggests.
An empty counting Bloom filter is an array of m counters, all set to 0. For each element, k hash functions map the element to k positions in the filter, that are incremented or decremented during insertion and deletion.

Assignment 04 - Task 1

Task

Create a class `CountingBloomFilter` that stores the counts (**uint8**) in a **numpy array**. In addition, implement the following methods:

1. `__init__(self, m, k, hash)`: class constructor with `m` being the filter size, `k` the number of hash functions and a list of integers `hash` to compute the hash values in the function `__hash__`.
2. `__hash__(self, x, i)` computes the i-th hash function for an element `x`. The formula for the hash function is `(x * hash[i]) % m`.
3. `insert(self, x)` inserts element `x` by incrementing the counters by 1.
4. `delete(self, x)` deletes element `x` by decrementing the counters by 1 (if contained).
5. `contains(self, x)` checks whether an element is in the filter or not (returns `True` or `False`).
6. `count(self, x)` returns the count the filter currently suggests for the element `x`.

You may assume that only integers are inserted into the filter.

Assignment 04 - Task 2

Assignment 04 - Task 2 (3 points)

You take part in a treasure hunt and have to solve a puzzle to find the room number in a very large building in which you can find a new hint.
The room number is encrypted as a list of integers that you have to rearrange before computing the room number. To rearrange the list, you have to move each number **forward or backward by its value** in the order that they **originally appeared** in the list. Numbers moving around during the rearrangement process do not change the order in which the numbers are moved. In addition, the list is **circular**, which means that if a number is moved over one end of the list it wraps back around to the other end as if the ends were connected.
After the rearrangement process, the room number can be found by computing the sum of the **1000th, 2000th, and 3000th** numbers **after the value 0**, wrapping around the list as necessary.

Assignment 04 - Task 2

Example

For example:

Initial arrangement:

`1, 2, -3, 3, -2, 0, 4`

1. Move 1 : `2, 1, -3, 3, -2, 0, 4`
2. Move 2 : `1, -3, 2, 3, -2, 0, 4`
3. Move -3 : `1, 2, 3, -2, -3, 0, 4`
4. Move 3 : `1, 2, -2, -3, 0, 3, 4`
5. Move -2 : `1, 2, -3, 0, 3, 4, -2`
6. Move 0 : `1, 2, -3, 0, 3, 4, -2`
7. Move 4 : `1, 2, -3, 4, 0, 3, -2`

In the above example, the 1000th number after 0 is 4, the 2000th is -3, and the 3000th is 2; adding these together gives 3.

Assignment 04 - Task 2

Task

Write a function `get_treasure(filename)` that reads the initial list from a file (one number per line), performs the rearrangement steps and then returns the room number. The input file always contains exactly on zero, to guarantee a unique starting position to find the room number. All other numbers are allowed to occur multiple times.

Hint

You have to find a way to keep track of which elements you already moved and which are next according to the original order!

Assignment 04 - Task 3

Assignment 04 - Task 3 (3 points)

In this task, you should implement **quantile normalization** for a matrix, e.g., a `gene x samples` matrix (see also **How to do quantile normalization correctly for gene expression data analyses**).

Steps:

1. Rank the genes (rows) of each sample (columns) by magnitude (keeping original order for tied values).
2. Calculate the average value for each rank.
3. Substitute all original values by the average value with the same rank.
4. Reorder the genes of each sample back into the original order.

Assignment 04 - Task 3

Example:

$$\begin{pmatrix} 2 & 4 & 4 & 5 \\ 5 & 14 & 4 & 7 \\ 4 & 8 & 6 & 9 \\ 3 & 8 & 5 & 8 \\ 3 & 9 & 3 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 4 & 3 & 5 \\ 3 & 8 & 4 & 5 \\ 3 & 8 & 4 & 7 \\ 4 & 9 & 5 & 8 \\ 5 & 14 & 6 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} 3.5 & 3.5 & 3.5 & 3.5 \\ 5 & 5 & 5 & 5 \\ 5.5 & 5.5 & 5.5 & 5.5 \\ 6.5 & 6.5 & 6.5 & 6.5 \\ 8.5 & 8.5 & 8.5 & 8.5 \end{pmatrix} \rightarrow \begin{pmatrix} 3.5 & 3.5 & 5 & 3.5 \\ 8.5 & 8.5 & 5.5 & 5.5 \\ 6.5 & 5 & 8.5 & 8.5 \\ 5 & 5.5 & 6.5 & 6.5 \\ 5.5 & 6.5 & 3.5 & 5 \end{pmatrix}$$

Task:
Write a function `quantile_normalize` that gets a matrix (NumPy 2D array) as function argument returns the matrix after quantile normalization.