Programming Assignment 5:
Running Average and Running Median

You are given a stream of numbers (such as visitor counts to a web-site every minute). You are asked to give a running arithmetic mean and a running median of the numbers in the stream. Of course, you could store all of the numbers and then calculate these two values, but this becomes quickly expensive. Instead, you have to develop a data structure that has enough data to calculate the new mean from its current state and the new number arriving in the stream.

For the arithmetic mean, our data structure consists of the number of elements seen so far and the arithmetic mean so far. To update after receiving one more number, we increment the number of elements. As far as the arithmetic mean with the new element is concerned, we can calculate (with stream $a_1, a_2, \ldots, a_n$):

$$\begin{aligned}\text{new mean} &= \frac{a_1 + a_2 + a_3 + \ldots + a_n + a_{n+1}}{n+1} \\ &= \frac{n\frac{(a_1 + a_2 + a_3 \ldots + a_n)}{n} + a_{n+1}}{n+1} \\ &= \frac{n \cdot (\text{old mean}) + a_{n+1}}{n+1}.\end{aligned}$$

Task 1:

- Create a data structure that contains the number of elements seen so far and the arithmetic mean of the elements.
- Create a function that updates the data structure with the new mean and the new number of elements seen.
- Test your solution by creating a stream of random numbers that (1) are a random shuffle of the numbers between 1 and 30. (2) are exponentially distributed. (In Python, install numpy or scipy).

The median is the value of the middle element in a sorted array of numbers if the array has an odd number of numbers or the arithmetic mean of the two middle elements if the number of numbers in the array is even. For example:

Given [1, 7, 2, 3, 8, 1, 9], we sort the array and get [1, 1, 2, 3, 7, 8, 9]. Because there are seven numbers, we pick the middle one, which is 3. This is the median.

Given [2, 3, 12, 11, 10, 5, 4, 2, 1, 2], we first sort the array and get [1, 2, 2, 2, 3, 4, 5, 10, 11, 12]. The two middle elements are 3 and 4, with mean (3+4)/2 = 3.5. This is the median.

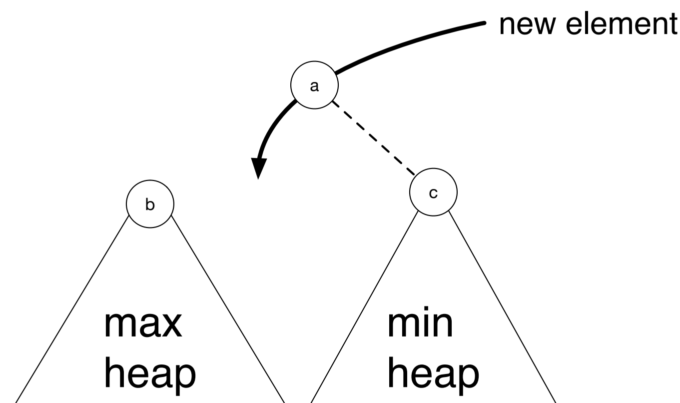
The data structure for the online median calculation is more difficult. The naïve algorithm adds the new number to an array of numbers seen so far. This array is ordered. When a new number arrives, we can use binary search in order to insert the number into the array so that the array remains ordered. This takes $O(\log(n))$ steps. Then we find the middle element or two elements in the array and get the median from them. The draw-back of this data structure is the insertion of an element into the middle of an array. If the array is a linked list, then the insertion is easy, but binary search is a challenge. If the array is implemented as a C-style array, then the binary search is easy, but the insertion is a challenge, as it means shifting elements.

A simpler alternative which you are going to use uses priority queues (or heaps). A heap is essentially a C-style array. In a **min-heap** with n elements, we can insert an element in time $O(\log(n))$ and extract the minimum element of all those inserted in time $O(\log(n))$ and look at the minimum element in constant time. A **max-heap** has the same operations, only we extract the maximum element and inspect the maximum element.

The data structure for online median consists of a min-heap and a max-heap. The elements in the min-heap and the max-heap are exactly the elements previously seen in the stream. The elements in the max-heap are smaller or equal to the elements in the min-heap. The number of elements in the max-heap is the same as the number of elements in the min-heap or one more.

When a new element from the stream arrives, we compare it with the minimum element in the min-heap. If the new element is smaller than the minimum of the numbers in the min-heap, then we insert it in the max-heap, otherwise we insert it in the min-heap. Thus, all numbers in the max-heap are strictly smaller than the minimum number in the min-heap.

It can of course happen, that after the insertion, the numbers of the heaps no longer reflect the requirement of equal or almost equal size. If this happens, we move one element from one heap to the other. In more detail, if the number of elements in the max-heap is larger than the number of elements in the min-heap, then we extract the maximum element of the max-heap (at cost $\approx O(\log(n/2))$ and insert it into the min-heap (at cost $\approx O(\log(n/2))$. Reversely, if the number of elements in the min-heap exceeds the number of elements in the max-heap by more than one, then we extract the minimum number in the min-heap and push it onto the max-heap.



To extract the median, we distinguish two cases. If the number of elements in the heaps are the same, then the median is the mean of the maximum of the max-heap and the minimum of the min-heap. If there is one more element in the max-heap, then the minimum is the maximum of the max-heap.

Task 2:

Implement online median. Use the same tests.

Hints:

Here is the first test, assuming you used a class to implement the online median.

```
rm = Running_Median()
test = list(range(1,31))
random.shuffle(test)
print(test)
for ele in test:
    rm.digest(ele)
    print(rm)
    print(rm.median())
    print(10*'-')
```

And here is the second:

```
rm = Running_Median()
test = np.random.exponential(scale=10, size=10000)
for ele in test:
    rm.digest(ele)
print(rm.median(), test.mean())
```

Heaps and priority queues are implemented in Python. Here are two classes that wrap the heapq class in Python.

```
class Min_Heap:
    def __init__(self):
        self.pq = []
    def __str__(self):
        return str(self.pq)
    def push(self, element):
        heapq.heappush(self.pq, element)
    def pop(self):
        try:
            return heapq.heappop(self.pq)
        except IndexError:
            return None
    def inspect(self):
        if self.pq == []:
            return None
```

```
        return self.pq[0]
    def __len__(self):
        return len(self.pq)


class Max_Heap:
    def __init__(self):
        self.pq = []
    def __str__(self):
        return str([-x for x in self.pq])
    def push(self, element):
        heapq.heappush(self.pq, -element)
    def pop(self):
        try:
            return -heapq.heappop(self.pq)
        except IndexError:
            return None
    def inspect(self):
        if self.pq == []:
            return None
        return -self.pq[0]
    def __len__(self):
        return len(self.pq)
```