Kaito Minami
DS4300: Large-scale Information Storage and Retrieval

I created two functions: postTweets (for Strategy 2) and postTweetsStr1 (for Strategy 1).
Strategy 1 for postTweets ran much faster than strategy 2 because I only set tweet and managed the tweets by tweet_id key stored in a set, tagged with user_id.
Strategy 2, on the other hand, I had to first retrieve all the followers from user_id and for each followers, I then added tweets for their timeline.
If strategy 1 runs 2 million times for 2 processes, strategy 2 could run 1 million * (average follower numbers + 1). And most people have more than 1 followers, making it significantly slower.

```
Function postTweets ran 586.9965207461987 times in 1 second.
Average execution time: 1703.58761 seconds per function call
1
Function postTweetsStr1 ran 1941.5350900327123 times in 1 second.
Average execution time: 515.05636 seconds per function call
1
```

I created two functions: getHomeTimeline (for Strategy 2) and getHomeTimelineStr1 (for Strategy 1).
Here, strategy 2 is much faster than strategy 1 because all the hard work is done in postTweets process. For strategy 2, I only randomly picked 10 tweets and sorted by timestamp from already-created timeline set.
Meanwhile, in strategy 1, I first had to pick a followee randomly, retrieved random tweet from them, append it on a virtual timeline, and sort by timestamp, rinse and repeat for 10 times.

```
Function getHomeTimeline ran 3208 times in 1 second.
Average execution time: 0.00031 seconds per function call
3208
Function getHomeTimelineStr1 ran 232 times in 1 second.
Average execution time: 0.00432 seconds per function call
232
```

Overall, I think this result is not very surprising, as one process bears more tasks than another. And that's how it appears in their clear strength and weakness for both strategies. But compared to Homework 1 Twitter RDB (postTweets: 637, getHomeTimeline: 6), it clearly is better and it shows how efficient Redis is as a database.