BlueBox!

You are the embattled owner of a video rental service known as BlueBox. Business is not going so well, so you turn to your interns for fresh, youthful ideas to improve business.

One of your interns from MIT, who's always grumbling that he should have joined a finance firm like his friends, has come up with a new idea for a promotion: bundle movies and allow users to rent two movies at a time for the cost of one. For lack of any better ideas, you decide that you will implement this promotional offer right away, randomly bundling movies together and marketing your new `Two-for-One` promo offer.

Two weeks after `Two-for-One` goes into effect, you receive many complaints from customers that the good movies they want to watch are almost always paired with some other terrible movie that they're not even remotely interested in. This deal really isn't worth it for them unless they are getting two *good* movies.

Each movie has a rating, a continuous number between 0 and 5. You decide that you will only bundle movies if the sum of their scores is 7.5 or more. The score of each movie is uniformly distributed in \(\bf [0,5]\). Since budgets are low these days, any algorithm to pair movies will have to be run on your intern's laptop computers.

You are given a huge database of \(N = 10^{10}\) movies. All movies in the database are indexed by an unique integer id (\(0\leq id \leq 10^{20}-1\)). Each item consists of movie ID (20 chracters long) and popularity score (a float number between 0 and 5). The popularity score is uniformly distributed. You can access the database by calling `getMovie( index )`.

In [ ]:
```
import string
import random

def getMovie( index ):
    if index >= 10**10 or index < 0:
        return None
    random.seed( index )
    my_id = ''.join(random.choice(string.ascii_uppercase + string.digits) for
_ in range(20))
    my_score = random.random()*5
    return {'id':my_id, 'score':my_score}
```

$$1. The Stanford intern had a bold realization: "After I sort this database by scores, I can easily make good pairs!" Then, he tried to sort the database and his laptop froze. Why?

Solution Here

$$2. (Assume that the intern figured out the reason why his machine froze.) He came with a good solution: "query \(2\) random movies and see if the sum is greater than \(7.5\)!". What is the expected number of movies accessed in order to pick a *good* pair? Implement this algorithm and make sure it matches your math.

Solution Here

In [ ]:
```
# how to get score of random movie
id = random.randint(0,10**10)
getMovie(id)['score']
```

In [ ]:
```
N = 10**4 # number of times to simulate event

for i in range(N):
    # Your code here
```

$$3. While you were satisfied with the new algorithm, another intern from Stanford visited your office. She claims that she can indeed find a pair faster than the current algorithm! She proposes: "Pick one random element, and see whether it's greater than \(3.75\) or not. If it is, keep the element, and search again to find a second element that is greater than \(3.75\)!". What is the expected number of movies accessed in order to find a *good* pair? Implement this algorithm and make sure your math checks out.

Solution here

In [ ]:
```
N = 10**4 # number of times to simulate event

for i in range(N):
    # Your code here
```

$$4. An intern from Cal, who took EE126, comes to you. He claims that one can actually do better than the current algorithm. He proposes: "Pick a random movie, and if its score is lower than \(2.5\), throw it away. If its score is higher than \(2.5\), keep it. Continue picking random movies until you find one which has a score greater than \(2.5\). Now, pick another random movie. If the sum of scores of the two movies you have is higher than \(7.5\), you declare success. If not, keep the movie with higher score. Repeat this until you have two movies with a score greater than \(7.5\)." Implement this algorithm. Don't do the math.

In [ ]:
```
N = 10**4 # number of times to simulate event

for i in range(N):
    # Your code here
```

These algorithms are all pretty simple, and really just obvious improvements over one another. For the first two algorithms, the analysis was pretty straightforward, but the analysis for the last algorithm is non-trivial. Turns out, you can use Markov Chains to determine the expected number of movies you will pick until you get a *good* pair. You'll do the analysis in the written portion of this week's homework.