Emil Jógvan Svensmark

2022

ALG Assignment – Hand In 1

Experiment 1.4.44: Birthday Problem.

First, we will create a program that can generate an array of randomly generated numbers between 0 and N – 1. I have chosen to create this solution in python, as seen in the code shown below.

```
def sequencing(N, r):
    sequence = []
    for _ in range(r):
        sequence.append(random.randint(0,N-1))
    return sequence
```

As shown in the picture above, the first method designed is a method that takes in two parameters: N and r. N is the given number that should represent the range of 0 to N – 1 and the range is the length of the array.

Next, we should create a method that can calculate the value of the following equation:

$$\sim \sqrt{\frac{\pi N}{2}}$$

This equation represents the hypothesis of the number of integers that should be generated before a repeated number is found. The solution looks like this:

```
def hypothesis(N):
    print("Value of N: " + str(N))
    print("The expected value is: ~" + str(math.sqrt(math.pi*N/2)))
    r = math.ceil(math.sqrt(math.pi*N/2))
    print("Ceiling of value is: " + str(r))
    match = validateArray(sequencing(N,r))
    if (match):
        print("A repeated number was found, which has proven that the hypothesis was correct in this case")
    else:
        print("No repeated numbers were generated in this sequence.")
```

We use a ceiling to calculate the absolute value of the range, as it isn't possible to have a range of fractions.

Next and last thing for this program to function to create a way to validate repeated numbers in the generated array. As seen in the picture above, we expect a Boolean with a value of True if there is a match found, and false if there is no match found. The solution looks like this:

```
def validateArray(array):
    print("Validating given array for repeated numbers: " + str(array))
    count = 1
    for x in array:
        for y in array[count:]:
            if (x == y):
                return True
        count += 1
    return False
```

We use a nested loop in which we check every number (represented by x) and comparing it to every number after x in the same array (represented by y). This way we only compare new combinations, and no time and power will be wasted with redundant comparisons.

Running the code looks like this:

```
N = 20
hypothesis(N)
```

```
Value of N: 20
The expected value is: ~5.604991216397929
Ceiling of value is: 6
Validating given array for repeated numbers: [5, 4, 9, 5, 4, 14]
A repeated number was found, which has proven that the hypothesis was correct in this case
```

From the above result we can conclude the following:

With an N of 20, the hypothesis is that the number of integers generated before the first repeating number should be ~5.6 (rounding up to use the absolute number).

Doing this only once, however, does not prove anything. We would need to do this at a bigger scale, which is why we should create a method that can run this method n times and return the amount of success and failures of the hypothesis for each time run.

We modify our current code to the following:

```
def validateArray(array):
    #print("Validating given array for repeated numbers: " + str(array))
    count = 1
    for x in array:
        for y in array[count:]:
            if (x == y):
                return True
        count += 1
    return False

def hypothesis(N):
    #print("Value of N: " + str(N))
    #print("The expected value is: ~" + str(math.sqrt(math.pi*N/2)))
    r = math.ceil(math.sqrt(math.pi*N/2))
    #print("Ceiling of value is: " + str(r))
    return validateArray(sequencing(N,r))
    #match = validateArray(sequencing(N,r))
    #if (match):
        #print("A repeated number was found, which has proven that the hypothesis was correct in this case")
    #else:
        #print("No repeated numbers were generated in this sequence.")
```

And implement our “mass-production” method:

```
def massHypothesis(N, n):
    print("Running the hypothesis " + str(n) + " times ..")
    successes = 0
    failures = 0
    for _ in range(n):
        if(hypothesis(N)):
            successes += 1
        else:
            failures += 1
    print("Result:")
    print("Amount of successes in hypothesis: " + str(successes))
    print("Amount of failures in hypothesis: " + str(failures))
```

Running the code hypothesis 1000 times looks like this:

```
N = 20
#hypothesis(N)
n = 1000
massHypothesis(N,n)
```

With the result of 5 runtimes:

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 577
Amount of failures in hypothesis: 423
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 582
Amount of failures in hypothesis: 418
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 555
Amount of failures in hypothesis: 445
```

Let’s try to modify the hypothesis a bit now. We can for example either add more numbers to the range than the equation suggests or make it less and see how much it impacts the result. For example, let’s try to add another number to the range. For this, we simply have to multiply the range by 1:

```
return validateArray(sequencing(N,r+1))
```

Let’s see the results then:

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 698
Amount of failures in hypothesis: 302
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 691
Amount of failures in hypothesis: 309
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 703
Amount of failures in hypothesis: 297
```

By doing this we increased the chance by nearly 15%. Let's see if we add 4 additional, 5 in total:

```
return validateArray(sequencing(N,r+5))
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 970
Amount of failures in hypothesis: 30
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 969
Amount of failures in hypothesis: 31
```

```
Running the hypothesis 1000 times ..
Result:
Amount of successes in hypothesis: 983
Amount of failures in hypothesis: 17
```

There is now approximately only 1,5 - 3% chance of an array NOT containing repeating numbers.

In conclusion, there was only about a 55% to see an array with a repeating number when following the amount of numbers calculated by the equation. However, slightly modifying the result from the equation greatly impacts the result.

Experiment 1.4.45: Coupon collector problem.

For this problem we can reuse some of the code written from the previous experiment (1.4.44). We have chosen to reuse the same sequence generating method: **sequencing(N, r)**

However, first of all, we need a way to calculate the n-th harmonic number, so that we can get the result from the given equation:

$$\sim N \cdot H_N$$

The the n-th harmonic number is calculated by the following method:

```
def calculateHarmonicNumber(N):
    H_n = 0
    for x in range(N):
        x += 1
        H_n += 1/x
    return H_n
```

Now for the hypothesis, in which we calculate the number of integers that should be generated before all possible values should have been generated:

```
def hypothesis(N):
    r = math.ceil(N*calculateHarmonicNumber(N))
    return validateArray(N,sequencing(N,r))
```

The validation of the hypothesis is done with the **validateArray()** method, as shown below:

```
def validateArray(N, array):
    result = [None] * N
    count = 0
    for x in array:
        if (x != result[x]):
            result[x] = x
            count += 1
            if (count == len(result)):
                return True
    return False
```

In this method we check every single value in the given array, until we meet the requirements of the hypothesis; that every possible value has been generated. The way we validate each value is by adding it to an array with the size of N (the amount of different numbers available, in given situation) and add it to the same values index of the empty array, if it hasn't already been added. Every time a number is added to the result array, we know we have found a new unique number and the count value is counted up once. When the count value reaches the length of the result array, we know that every possible number has been added, and therefore has achieved the requirements.

For a mass production of the hypothesis, we have created a similar method to the other **massHypothesis()** function:

```
def massHypothesis(N, n):
    print("Running the hypothesis " + str(n) + " times ..")
    print("The number of intergers generated is: " + str(math.ceil(N*calculateHarmonicNumber(N))))
    successes = 0
    failures = 0
    for _ in range(n):
        if(hypothesis(N)):
            successes += 1
        else:
            failures += 1
    print("Result:")
    print("Amount of successes in hypothesis: " + str(successes))
    print("Amount of failures in hypothesis: " + str(failures))

massHypothesis(50,1000)
```

When running this method three times with a given range from 0 to 50 (N = 50) and a mass production of 1000 (running **hypothesis()** 1000 times), we receive the following results:

```
Running the hypothesis 1000 times ..
The number of intergers generated is: 225
Result:
Amount of successes in hypothesis: 571
Amount of failures in hypothesis: 429
```

```
Running the hypothesis 1000 times ..
The number of intergers generated is: 225
Result:
Amount of successes in hypothesis: 558
Amount of failures in hypothesis: 442
```

```
Running the hypothesis 1000 times ..
The number of intergers generated is: 225
Result:
Amount of successes in hypothesis: 569
Amount of failures in hypothesis: 431
```

From the results we can see that there is approximately a 57% chance of success, which is very similar to the other experiment.

Create problem 2.1.13 - Make the program

For this problem we started creating two classes to represent each a card and card deck, which is done with the following code:

```
class Card:
    def __init__(self, suit, value):
        self.suit = suit
        self.value = value

    def getCardValue(self):
        if(self.suit == "Spades"):
            return self.value
        if(self.suit == "Hearts"):
            return self.value + 13
        if(self.suit == "Clubs"):
            return self.value + 26
        if(self.suit == "Diamonds"):
            return self.value + 39

    def __repr__(self):
        return (str(self.suit) + " " + str(self.value))
```

```
class Deck:
    def __init__(self):
        self.cards = []

    def fillDeck(self):
        if not self.cards:
            suits = ["Spades", "Hearts", "Clubs", "Diamonds"]
            for suit in suits:
                for value in range(13):
                    card = Card(suit, value+1)
                    self.cards.append(card)

    def shuffleDeck(self):
        random.shuffle(self.cards)
```

Now that we both have a fill method aswell as a shuffle method we can now begin working on a method that sorts the deck once it has been shuffled. The exercise says that we are to use a insertion sorting method.

```
37      def sortDeck(self):
38          for i in range(1, len(self.cards)):
39              key = self.cards[i]
40              j = i-1
41              while j >=0 and key.getCardValue() < self.cards[j].getCardValue():
42                      self.cards[j+1] = self.cards[j]
43                      j -= 1
44              self.cards[j+1] = key
```

From line 38 I represents every card in the deck, except for the first card, as we are always comparing the card to the left, which is represented by j (which has a start value of I-1; the card to the left). In line 41 we now shift the hovered card to the left until the card has a higher value than the card left of it, and then we i+1.

In action the code runs like this:

```
deck = Deck()
#print(deck.cards)
deck.fillDeck()
print("")
print("")
print("Deck will now be filled with a full deck of cards:")
print(deck.cards)
deck.shuffleDeck()
print("")
print("")
print("Deck is now shuffled, and has the following array:")
print(deck.cards)
print("")
print("")
print("Deck will now be sorted:")
deck.sortDeck()
print(deck.cards)
```

```
Deck will now be filled with a full deck of cards:
[Spades 1, Spades 2, Spades 3, Spades 4, Spades 5, Spad
es 6, Spades 7, Spades 8, Spades 9, Spades 10, Spades 1
1, Spades 12, Spades 13, Hearts 1, Hearts 2, Hearts 3,
Hearts 4, Hearts 5, Hearts 6, Hearts 7, Hearts 8, Heart
s 9, Hearts 10, Hearts 11, Hearts 12, Hearts 13, Clubs
1, Clubs 2, Clubs 3, Clubs 4, Clubs 5, Clubs 6, Clubs 7
, Clubs 8, Clubs 9, Clubs 10, Clubs 11, Clubs 12, Clubs
 13, Diamonds 1, Diamonds 2, Diamonds 3, Diamonds 4, Di
amonds 5, Diamonds 6, Diamonds 7, Diamonds 8, Diamonds
9, Diamonds 10, Diamonds 11, Diamonds 12, Diamonds 13]
```

When filled the cards are already in order.

```
Deck is now shuffled, and has the following array:
[Diamonds 10, Hearts 9, Clubs 8, Spades 1, Diamonds 6,
Clubs 10, Diamonds 7, Clubs 6, Spades 10, Spades 5, Clu
bs 2, Spades 4, Hearts 8, Hearts 2, Spades 2, Clubs 11,
 Spades 7, Spades 3, Clubs 7, Hearts 10, Clubs 3, Clubs
 5, Hearts 5, Hearts 3, Spades 12, Spades 8, Hearts 6,
Clubs 13, Hearts 12, Hearts 4, Diamonds 5, Spades 11, C
lubs 1, Diamonds 8, Diamonds 12, Spades 13, Hearts 11,
Clubs 4, Diamonds 4, Diamonds 13, Diamonds 9, Spades 9,
 Clubs 9, Diamonds 11, Hearts 1, Hearts 7, Clubs 12, Di
amonds 3, Diamonds 2, Diamonds 1, Hearts 13, Spades 6]
```

After shuffling the deck, the cards are now in a random order.

```
Deck will now be sorted:
[Spades 1, Spades 2, Spades 3, Spades 4, Spades 5, Spad
es 6, Spades 7, Spades 8, Spades 9, Spades 10, Spades 1
1, Spades 12, Spades 13, Hearts 1, Hearts 2, Hearts 3,
Hearts 4, Hearts 5, Hearts 6, Hearts 7, Hearts 8, Heart
s 9, Hearts 10, Hearts 11, Hearts 12, Hearts 13, Clubs
1, Clubs 2, Clubs 3, Clubs 4, Clubs 5, Clubs 6, Clubs 7
, Clubs 8, Clubs 9, Clubs 10, Clubs 11, Clubs 12, Clubs
 13, Diamonds 1, Diamonds 2, Diamonds 3, Diamonds 4, Di
amonds 5, Diamonds 6, Diamonds 7, Diamonds 8, Diamonds
9, Diamonds 10, Diamonds 11, Diamonds 12, Diamonds 13]
```

And after sorting the deck, we expect it to be the same as when we filled the deck; which we can see that it is by printing it out again.