Roll No: EE18B067 Name:Abhishek Sekar

Date: September 10, 2021

1 Reinforcement of concepts / methods

1.1. [Python Function]

(a) Write a Python function that takes a positive integer n, and returns the sum of the squares of all the positive integers smaller than n.

Solution:
Code:

```
def square_sum(n):
    if(n <= 0):
        return -1
    elif(type(n) != int):
        return -1
    else:
        s = 0
        for i in range(1,n):
            s += i**2
        return s
```

Input and Output

- **Input:** Positive integer n.
- **Output:** Sum of squares of all positive integers below n.

Sample Input and Sample Output:

- Please input a positive integer.
  10
  The sum of squares of all the positive integers below n is 285
- Please input a positive integer.
  50
  The sum of squares of all the positive integers below n is 40425

Explanation
Note: The codes throughout the latex report have been left uncommented so as to improve the ease of reading the report as the comments make it look cluttered.

The function first checks if the given input is positive and an integer. If it isn't positive or is not an integer, it returns an error value of -1.
There is an integer s which is initialized to 0 and this variable is used to store the result. If the input is a valid one, the function has a for loop which iterates through all integers from 1 to n. For every iteration i, i^2 is added to the sum variable s. Therefore, as we iterate through the loop, s adds the square of all integers part of the iteration,thereby

capturing the required output.
Finally, this sum s is returned from the function.
Time Complexity:
As we iterate through 1 to n in the for loop, our time complexity is of the order of n,ie, O(n).
Alternate Approach:
Instead of utilizing a function with time complexity O(n), we can try performing the same operation using a derived formula, shown in the below equation.

$$\sum_{i=0}^{n-1} i^2 = \frac{(n)(n-1)(2n-1)}{6} \tag{1}$$

This will reduce the time complexity to O(1) which is evident when we look at the time taken by both the codes for the same input.
0.00099754 seconds is the time taken using the iterated approach while the time taken by the one-shot approach discussed above is too short to be calculated accurately.
Code:

```
def alter_square_sum(n):
    if(n <= 0):
        return -1
    elif(type(n) != int):
        return -1
    else:
        return int(n*(n-1)*(2*n - 1)/6)
```

(b) Write a Python function that takes a positive integer n, and returns the sum of the squares of all the odd positive integers smaller than n.

Solution:
Code:

```
def odd_square_sum(n):
    if(n <= 0):
        return -1
    elif(type(n) != int):
        return -1
    else:
        s = 0
        for i in range(1,n,2):
            s += i**2
        return s
```

Input and Output

- **Input:** Positive integer n.
- **Output:** Sum of squares of all odd positive integers below n.

Sample Input and Sample Output:

- Please input a positive integer.
  10
  The sum of squares of all the odd positive integers below n is 165
- Please input a positive integer.
  50
  The sum of squares of all the odd positive integers below n is 20825

Explanation
This function is very similar to that discussed in the previous subdivision.The only difference being in the iteration stage. Where we iterated from 1 through n, here, instead, we iterate through 1 through n in steps of 2, employing the range constructor.This will iterate through all odd integers from 1 through n and as in the previous case, we keep adding the square at every iteration to our sum variable to produce the desired result. **Time Complexity:**
As we iterate through 1 to n in the for loop, our time complexity is O(n).
Alternate Approach:
Like in the previous case, we can employ a one shot approach using a derived formula for the sum which is shown below.

$$\sum_{i=1}^{n}(2i-1)^2 = \frac{(n)(2n-1)(2n+1)}{3} \qquad \text{(For n odd integers)} \qquad (2)$$

For the given integer n, there will be $k = n//2$ odd integers below n, where // is integer division.This k can be plugged in the expression above to generate the result. As in the previous case, this will reduce the time complexity to O(1).
Code:

```
def odd_alter_square_sum(n):
    if(n <= 0):
        return -1
    elif(type(n) != int):
        return -1
    else:
        k = n//2
        return int(k*(2*k+1)*(2*k - 1)/3)
```

1.2. What parameter values should be sent to the range constructor to produce a range with values:
(a) 60,70,80

Solution:
Code:

```
l = []
for i in range(60,81,10):
    l.append(str(i))
print(",".join(l))
```

Output:
60,70,80

Explanation
The range constructor takes three integer arguments as follows:
range(start,stop,step)
The range constructor increments or decrements by an amount of step from start to stop-1 provided stop is greater than start if we're incrementing or vise versa if we're decrementing (ie, step is negative)
Now using this, by looking at the given series, we see that it starts from 60 and goes on till 80 in steps of 10. Therefore comparing the syntax of the range constructor, we have,

- start = 60
- stop = 80 + 1 = 81
- steps = 10

If we implement the above on code, we do get the given series as our output as shown above.

(b) 4,2,0,-2,-4

Solution:
Code:

```
l = []
for i in range(4,-5,-2):
    l.append(str(i))
print(",".join(l))
```

Output:
4,2,0,-2,-4
Explanation:
We observe that the given series starts with 4, reduces by 2 to produce every subsequent term and ends with -4. Comparing this with the syntax of the range constructor described in the previous part (for the decrementing case), we see that,

- start = 4
- stop = -4 - 1 = -5 (as we're decrementing, stop is negative)
- steps = -2

2 Creativity

2.1. Write a Python function that takes a sequence of integer values and determines if there is a distinct pair of numbers in the sequence whose product is odd.

Solution:
Code:

```
def prodd(n):
    count = 0
```

```
    for i in range(len(n)):
        count = 0
        if(n[i] % 2 == 1):
            count += 1
            for j in range(i+1,len(n)):
                if(n[j] % 2 == 1):
                    if(n[j] != n[i]):
                        count += 1
                        break
        if(count == 2):
            return "Yes"
    return 'No'
```

Input and Output

- **Input:** Integer array n.
- **Output:** string:'Yes' if there exists a distinct pair of numbers whose product is odd and 'No' otherwise.

Sample Input and Sample Output:

- input: [3,2,3,4,3,3,4] output: "No"
- input:[1,2,3,4,5] output: "Yes"

Explanation
The function checks if the given input n has a distinct pair of numbers whose product is odd. As we probe the given constraint, we see that the product of two numbers is odd, only when the two numbers themselves are odd. Using this as our approach, the function iterates through the array to check if it can find two distinct odd integers. It first iterates through the elements of the array searching for an odd integer.Should it find one, it updates the count to 1 and searches for another distinct odd integer from the next index of the array till the end of the array. Should it not find any , count is reset to 0 and this process is repeated till all elements of the array are exhausted. However,if it finds another distinct odd number, count is updated to 2 and it breaks out of the inner loop. As our requirement is to find two distinct odd integers, the function returns "Yes" as soon as count reaches two.If this condition is not met, at the end of all the iterations, "No" is returned.
Time Complexity:
As we iterate through 1 to x,where x is the length of the array, in the outer for loop and from i+1 to x in the inner for loop, we perform $\frac{x^2}{2}$ computations for the worst case.
Therefore, the time complexity of this algorithm is O(x^2).

2.2. Write a Python function that counts the number of vowels in a given character string.

Solution:
Code:

```
def vowel_count(s):
    s = s.lower()
    count = 0
```

```
    for i in range(len(s)):
        if(s[i] == 'a'):
            count += 1
        elif(s[i] == 'e'):
            count += 1
        elif(s[i] == 'i'):
            count += 1
        elif(s[i] == 'o'):
            count += 1
        elif(s[i] == 'u'):
            count += 1
    return count
```

Input and Output

- **Input:** string s.
- **Output:** integer count with the number of vowels in s.

Sample Input and Sample Output:

- input: 'Algorithms' output: 3
- input: 'Myth' output: 0

Explanation
This function first converts all characters of the string to lowercase, in order to accommodate strings mixing both upper and lower case characters.The count variable is initialised to 0.After this, it iterates through all characters of the string and compares this character with the set of vowels $\{a, e, i, o, u\}$ in a series of if-elif statements. If a match is found, the count variable is increased by 1.Once all the characters of the input string have been exhausted, the count is returned.
Time Complexity:
As we iterate through 1 to n,where n is the length of string s and perform comparisons, the time complexity of this algorithm is O(n).

2.3. Write a Python program that takes as input three integers, "a", "b" and "c", from the console and determines if they can be used in the following arithmetic formulas: (i) "a+b=c", (ii) "a=b-c", (iii) "a*b=c".

Solution: Code:

```
print('Enter three numbers a,b,c in a space separated manner. \n')
a,b,c = map(int , input().split())

if(c == a+b):
    print('The numbers follow the pattern, {} + {} = {} '.format(a,b,c))
if(c == b-a):
    print('The numbers follow the pattern, {} - {} = {} '.format(b,a,c))
```

```
if(c == b*a):
    print('The numbers follow the pattern, {} * {} = {}'.format(a,b,c))
else:
    print('These numbers can not be used in the above arithmetic formulas')
```

Input and Output

- **Input:** Three integers a,b,c.
- **Output:** Prints if the input satisfies the appropriate condition.

Sample Input and Sample Output:

- Enter three numbers a,b,c in a space separated manner.
  0 0 0
  The numbers follow the pattern, 0 + 0 = 0
  The numbers follow the pattern, 0 - 0 = 0
  The numbers follow the pattern, 0 * 0 = 0
- Enter three numbers a,b,c in a space separated manner.
  1 5 7
  These numbers can not be used in the above arithmetic formulas

Explanation
The program inputs three integers from the user and allocates it to variables a,b and c using a map. After this, the inputs are checked if the follow the given properties $\{(a+b=c),(a=b-c),(a*b=c)\}$ using a series of if statements and if it does, the corresponding statement acknowledging the same is printed for the user.If the inputs don't satisfy the given properties, a statement stating the same is printed.

2.4. Write a Python function that takes a sequence of numbers and determines if all the numbers are different from each other (that is, they are distinct).

Solution: Code:

```
def naive_unique(n):
    for i in range(len(n)):
        for j in range(i+1,len(n)):
            if(n[i] == n[j]):
                return 'Non-unique'
    return 'Unique'
```

Input and Output

- **Input:** integer array n.
- **Output:** string: "Unique" of all elements of the input n are distinct else "Non-unique".

Sample Input and Sample Output:

- input: [1,2,3,4,5,6] output: Unique
- input: [1,3,2,5,4,1] output: Non-unique

Explanation
This function is very similar to the one we implemented in question 2.1. We iterate through the given input using a nested for loop with the inner loop starting from the next index after current value of the outer loop. It searches for matches in the array, in this nested fashion by comparing two elements from the array. Should a match be found,the elements of the array are not unique and therefore it returns an output "Non-unique". If no match is found upon exhausting the elements of the input array, it returns an output "Unique".
Time Complexity:
Using the same logic as in 2.1, the time complexity of this algorithm is O(x^2) where x is the length of the input array. However, using an intermediate data structure called set, this time complexity can be reduced to O(x). This approach is shown below.
Code:

```
def optimized_unique(n):
    check = set()
    for i in range(len(n)):
        check.add(n[i])
    if(len(check) == len(n)):
        return 'Unique'
    else:
        return 'Non-unique'
```

Explanation
Here, we keep on appending elements to the set variable check.Check only retains the unique elements present in the input array. Therefore we compare the length of check and that of the input array. If the lengths are equal, we can ascertain that there is no copies of any element present in the input and return "Unique". On violation of this condition, we can see that the input array is not distinct.
Surprisingly, on inspecting the time it takes to run these codes, this "optimized" version of the function is not that very fast when compared to the original one. The reason for this could be attributed to the fact that the data structure list is processed much faster than set in python.

3 Project-based / Cross-Contextual

3.1. Design a program that can test the Birthday problem, by a series of experiments, on randomly generated birthdays which test this paradox for n = 5,10,15,20,25,30...200.

Solution: Code:

```
import numpy as np
import matplotlib.pyplot as plt

N_exp = 1000
```

```
probabilities = []
indices = []
for i in range(5,201,5):
    count = 0
    indices.append(i)
    for n in range(N_exp):
        arr = np.random.randint(1,365,i)
        if(optimized_unique(arr) == 'Non-unique'):
            count += 1
    probabilities.append(count/N_exp)

print(probabilities)
plt.plot(indices,probabilities)
plt.xlabel('n')
plt.ylabel('P[n]')
plt.title('The birthday paradox plot for different values of n')
plt.grid()
plt.show()
```

Input and Output

- **Input:** integer N_{exp}.
- **Output:** A plot of the probabilities, the probabilities themselves.

Sample Input and Sample Output:

- input: N = 1000
- output: [0.03, 0.12, 0.249, 0.416, 0.557, 0.707, 0.824, 0.883, 0.938, 0.97, 0.988, 0.992, 0.995, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

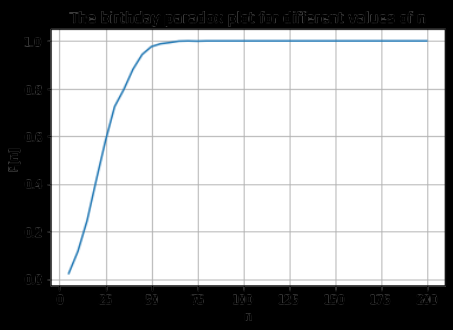

Figure 1: Plot of the probabilities for $N_{exp} = 1000$

- input: N = 10000

- output:[0.0279, 0.1182, 0.2562, 0.4172, 0.5789, 0.7032, 0.8154, 0.8901, 0.9412, 0.9724, 0.9877, 0.9932, 0.9985, 0.9993, 0.9996, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

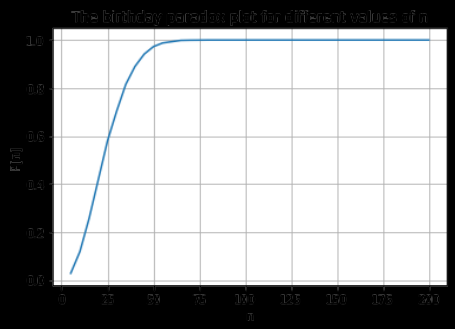

Figure 2: Plot of the probabilities for $N_{exp} = 10000$

Explanation
From wikipedia, we learn that the birthday paradox is a problem concerning with the probability of atleast 2 people amongst sharing their birthday. The reason this is a paradox is because it assumes that all days are equally likely when it comes to a person's birthday while in actually it isn't ie, the birthday paradox states that it requires at least 23 people for the probability of 2 people sharing their birthday to cross 50 percent, in reality it is 23 *or fewer*.
So to test the birthday paradox, we assume the same, that all birthdays are equally likely. We number the 366 unique birthdays from 1 to 366 and use a random number generator to incorporate the randomness involved for a birthday. So for each value of n, we obtain n random integers in the range $[1, 365]$ and use the function derived in the previous question to check if the array containing these integers is unique.
We repeat this experiment N_{exp} times and compute the probability as the number of experiments we get a non distinct array divided by the total no of experiments.

$$p[n] = \frac{\text{number of non-unique arrays}}{N_{exp}} \tag{3}$$

We compare the values of this probability with the accurate values as per the paradox computed using binomial random variables and plot the probabilities vs n. We can see that the probability values obtained through the code is very close to that of the paradox and the similarity of the graph just reiterates this. As we'd expect, a higher number of experiments come closer to the actual values.

3.2. Write a Python program that outputs all possible strings formed by using the characters 'c', 'a', 't', 'd', 'o', and 'g' exactly once.

Solution:
Code:

```
def all_possible_strings(input_str, start_ind = 0):
    if (start_ind == len(input_str)-1):
        print (" ".join(input_str))
    for i in range(start_ind, len(input_str)):
        str_copy = [c for c in input_str]
        str_copy[start_ind], str_copy[i] =str_copy[i], str_copy[start_ind]
        all_possible_strings(str_copy,start_ind + 1)
```

```
all_possible_strings('catdog')
```

Input and Output

- **Input:** string 'catdog'.
- **Output:** The 6! possible permutations of 'catdog'.

Explanation
Here we've implemented a function that generates all possible strings using the input characters in a recursive manner. Let us use a smaller example to explain how the code works.

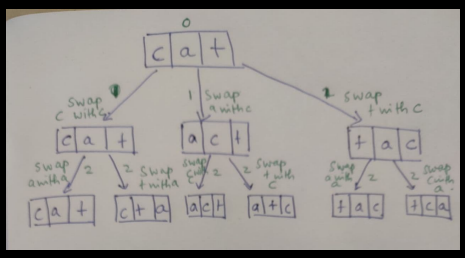

Figure 3: Working of the code using the example 'cat'

So first, the function is initialized with the input string cat and a start index term 0. The function works in three steps:

- First it creates a copy of the input string and names it str copy.
- Next it swaps the i^{th} element of str copy with the start ind element.
- It recursively calls the function, but this time with input str copy and start ind +1.

The crux of the function lies in the last two steps. The function keeps swapping some elements and retaining the others. What the recursive function call does is, it repeats this operation on the portion of the string that hasn't been swapped yet. So once all these combinations are exploited, which will take x iterations where x is the length of the string, the combinations are printed. This is explained by the above diagram.

- In the second iteration, in the string 'cat', c is swapped with c, the first element and that is sent back recursively to the function which leads us to a swapping with c and t swapping with c.
- In the final iteration, the second element of all the above strings are swapped with with itself and all the elements following it employing this recursive fashion and since start ind will now be equal to length -1 , it prints all the outputs.

The resulting structure resembles a tree with x leaves at the first step, x-1 at the second step etc.
Time Complexity:
As we iterate through 1 to x where x is the length of the string to produce an output and since there are x! outputs, the time complexity of this algorithm is O(x·x!).

3.3. Write a Python program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

Solution:
Code:

```python
n = int(input())
i = 0
if(n < 2):
    i = 0
else:
    i = len(list(bin(n))[2:])-1

print('Using binary approach',i)
i = 0
while(n >= 2):
    n = n/2
    i += 1

print("Using iterative approach",i)
```

Input and Output

- **Input:** integer n.
- **Output:** integer i which is the number of times we've to successively divide the input.

Sample Input and Sample Output:

- input: 10 output:3
- input:16 output:4

Explanation
Here, we've tried the same program in two different ways. In the first method, we use the fact that the numbers are stored in binary on the machine. So, for an integer n which takes d bits to be represented, we know that 2^d is greater than n. We also know that $2^(d-1)$ is lesser than or equal to n. Using these two, we see that,

$$2^{d-1} \leq n < 2^d$$
$$\Rightarrow 1 \leq n < 2 \qquad \text{(Dividing by } 2^{d-1} \text{ on both sides)}$$

Which directly tells us that we need to divide a number n, d-1 times to achieve the required result. This is an elegant manner to do this. In the second approach, we do it in a brute force iterative manner using a while loop, where we repeatedly check if the quotient on dividing by 2 is lesser than 2 or not. On every division, we update a count variable.
This count variable finally produces the result.
Time Complexity:
Whether the first approach or the second the time complexity of this program is O(log(n)) as the binary digit retrieval process in the computer is of the same order as our iterative approach of dividing successively by 2.