Санкт-Петербургский Национальный Исследовательский Университет

Информационных Технологий, Механики и Оптики

Факультет Технологического Менеджмента и Инноваций

**Лабораторная работа №6**

Выполнила:

Савинова А.К.

Проверил:

Мусаев А.А.

**Задание №1 (Игра с костями)**

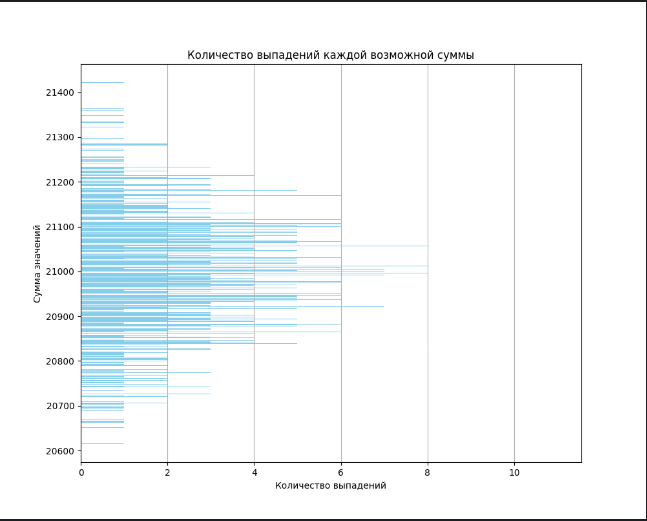
***Условие***: Вы подбрасываете 6000 кубиков, у которых 6 граней, 1350 раз и записываете сумму выпавших значений в список. Посчитайте сколько раз выпала каждая возможная сумма. Постройте график типа barh, где по горизонтальной оси будет количество выпаданий, а по вертикальной сумма значений.

***Код решения***:

import random  
import matplotlib.pyplot as plt  
  
# Подбрасываем кубики и записываем суммы в список  
results = []  
for \_ in range(1350):  
 dice\_sum = sum(random.randint(1, 6) for \_ in range(6000))  
 results.append(dice\_sum)  
  
# Подсчет количества выпадений каждой возможной суммы  
frequency = {}  
for result in results:  
 if result in frequency:  
 frequency[result] += 1  
 else:  
 frequency[result] = 1  
   
# Вывод графика  
def write\_graph():  
 # Построение графика

plt.figure(figsize=(10, 8))  
 plt.barh(list(frequency.keys()), list(frequency.values()), color='skyblue')  
 plt.xlabel('Количество выпадений')  
 plt.ylabel('Сумма значений')  
 plt.title('Количество выпадений каждой возможной суммы')  
 plt.grid(axis='x')  
 plt.show()

***График***:



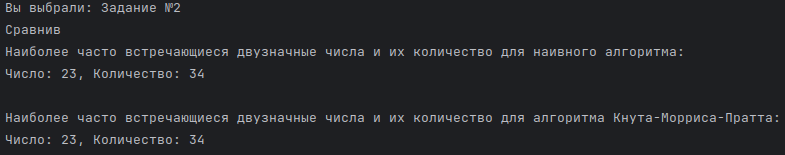
**Задание №2 (Сравнение или выбор?)**

***Условие***: Заполнить массив 500 простыми числами. Записать их в одну строку. Используя алгоритм поиска подстрок (Наивный+Кнута-Морриса-Пратта), посчитать количество наиболее часто встречающихся двузначных чисел в образовавшейся строке. Сравните изученные алгоритмы поиска подстрок. Сделайте вывод о их достоинствах и недостатках

***Код программы***:

# Реализация алгоритма Кнута-Морриса-Пратта для поиска подстроки в строке  
def kmp\_search(text, pattern):  
 def compute\_lps(pattern):  
 lps = [0] \* len(pattern)  
 length = 0  
 i = 1  
 while i < len(pattern):  
 if pattern[i] == pattern[length]:  
 length += 1  
 lps[i] = length  
 i += 1  
 else:  
 if length != 0:  
 length = lps[length - 1]  
 else:  
 lps[i] = 0  
 i += 1  
 return lps  
  
 def search(text, pattern):  
 count = 0  
 lps = compute\_lps(pattern)  
 i = j = 0  
 while i < len(text):  
 if pattern[j] == text[i]:  
 i += 1  
 j += 1  
 if j == len(pattern):  
 count += 1  
 j = lps[j - 1]  
 elif i < len(text) and pattern[j] != text[i]:  
 if j != 0:  
 j = lps[j - 1]  
 else:  
 i += 1  
 return count  
  
 return search(text, pattern)  
  
# Реализация наивного алгоритма для поиска подстроки в строке  
def naive\_search(text, pattern):  
 count = 0  
 for i in range(len(text) - len(pattern) + 1):  
 if text[i:i + len(pattern)] == pattern:  
 count += 1  
 return count  
  
  
# Реализация алгоритма Кнута-Морриса-Пратта для поиска подстроки в строке  
# (код KMP из предыдущего ответа)  
  
# Генерация простых чисел  
def generate\_primes(n):  
 primes = []  
 num = 2  
 while len(primes) < n:  
 for i in range(2, num):  
 if (num % i) == 0:  
 break  
 else:  
 primes.append(num)  
 num += 1  
 return primes  
  
  
# Получение строки из массива простых чисел  
primes = generate\_primes(500)  
primes\_str = ' '.join(map(str, primes))  
  
# Поиск наиболее часто встречающихся двузначных чисел в строке с использованием обоих алгоритмов  
frequencies\_naive = {}  
frequencies\_kmp = {}  
  
for i in range(len(primes\_str) - 1):  
 two\_digit\_num = primes\_str[i:i + 2]  
 if len(two\_digit\_num) == 2 and int(two\_digit\_num) >= 10 and int(two\_digit\_num) <= 99:  
 count\_naive = naive\_search(primes\_str, two\_digit\_num)  
 frequencies\_naive[two\_digit\_num] = count\_naive  
  
 count\_kmp = kmp\_search(primes\_str, two\_digit\_num)  
 frequencies\_kmp[two\_digit\_num] = count\_kmp  
  
# Наиболее часто встречающиеся двузначные числа и их количество для каждого алгоритма  
most\_common\_naive = max(frequencies\_naive.values())  
result\_naive = [num for num, freq in frequencies\_naive.items() if freq == most\_common\_naive]  
  
most\_common\_kmp = max(frequencies\_kmp.values())  
result\_kmp = [num for num, freq in frequencies\_kmp.items() if freq == most\_common\_kmp]  
  
# Вывод результата  
def print\_result():  
  
 print("Наиболее часто встречающиеся двузначные числа и их количество для наивного алгоритма:")  
 for num in result\_naive:  
 print(f"Число: {num}, Количество: {most\_common\_naive}")  
  
 print("\nНаиболее часто встречающиеся двузначные числа и их количество для алгоритма Кнута-Морриса-Пратта:")  
 for num in result\_kmp:  
 print(f"Число: {num}, Количество: {most\_common\_kmp}")

***Вывод программы***:

******

**Задание №3 (Плагиат или сам?)**

***Условие***: С помощью алгоритма Бойера-Мура для заданного реферата (файл), определите количество плагиата (в % от общего количества символов в реферате), взяв за основу соответствующие статьи из Википедии (название файла = название статьи). За плагиат считать любые 3 совпавших слова, идущих подряд.

***Код программы***:

from docx import Document  
  
# Реализация алгоритма Бойера-Мура для поиска подстроки в тексте  
def boyer\_moore(text, pattern):  
 # Конвертирование текста и паттерна в нижний регистр  
 text = text.lower()  
 pattern = pattern.lower()  
  
 # Получение длин текста и паттерна  
 n = len(text)  
 m = len(pattern)  
  
 # Подсчет таблицы смещений для алгоритма Бойера-Мура  
 shifts = {}  
 for i in range(m - 1):  
 shifts[pattern[i]] = m - i - 1  
  
 # Начало поиска с позиции паттерна  
 i = m - 1  
 while i < n:  
 j = m - 1  
 k = i  
  
 # Проверка совпадения символов с конца паттерна  
 while j >= 0 and text[k] == pattern[j]:  
 k -= 1  
 j -= 1  
  
 # Если все символы паттерна совпали, возвращаем индекс начала совпадения  
 if j == -1:  
 return i - m + 1  
  
 # Смещение в соответствии с таблицей смещений  
 if text[i] in shifts:  
 i += shifts[text[i]]  
 else:  
 i += m  
  
 return -1 # Паттерн не найден  
  
  
# Функция для чтения текста из docx файла  
def read\_text\_from\_docx(file\_name):  
 doc = Document(file\_name)  
 text = ''  
 for paragraph in doc.paragraphs:  
 text += paragraph.text + '\n'  
 return text  
  
  
# Функция для определения плагиата  
def find\_plagiarism(reference\_text, wikipedia\_text):  
 reference\_words = reference\_text.split() # Разделение реферата на слова  
  
 # Инициализация переменной для подсчета количества совпадений  
 plagiarism\_count = 0  
  
 # Поиск плагиата, считаем три подряд идущих слова как плагиат  
 for i in range(len(reference\_words) - 2):  
 three\_words = ' '.join(reference\_words[i:i + 3]) # Получение трех слов  
 index = boyer\_moore(wikipedia\_text, three\_words) # Поиск трех слов в тексте из Википедии  
 if index != -1:  
 plagiarism\_count += 1  
  
 # Вычисление процента плагиата  
 total\_words = len(reference\_words)  
 plagiarism\_percentage = (plagiarism\_count / (total\_words - 2)) \* 100  
  
 return plagiarism\_percentage  
  
  
# Пример использования  
# Считываем тексты из docx файлов  
wikipedia\_article = read\_text\_from\_docx('/home/ru6ik/Desktop/Alina Study/Programming/lab6/exercise\_three/fileForSearchAndOriginal//Жизнь\_из\_Википедии.docx')  
reference\_article = read\_text\_from\_docx('/home/ru6ik/Desktop/Alina Study/Programming/lab6/exercise\_three/fileForSearchAndOriginal//Жизнь.docx')  
  
plagiarism\_percentage = find\_plagiarism(reference\_article, wikipedia\_article)  
def print\_answer():  
 print(f"Процент плагиата: {plagiarism\_percentage:.2f}%")

***Вывод программы:***