МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

Отчет по лабораторной работе №4
Работа с функциями в языке Python.

по дисциплине «Технологии программирования и алгоритмизации»

Выполнила студентка группы ИВТ-б-о-20-1
Новикова В.С. « »_____20__г.
Подпись студента _____
Работа защищена « »_____20__г.
Проверил Воронкин Р.А. _____
(подпись)

Ставрополь 2021

Цель работы: приобретение навыков по работе с функциями при написании программ с помощью языка программирования Python версии 3x.

Ход работы:

Ссылка на репозиторий: https://github.com/Valentina1502/LABA_4

Пример 1. (рис. 1).

Оценить с помощью модуля timeit скорость работы итеративной и рекурсивной версий функций factorial и fib. Во сколько раз изменится скорость работы рекурсивных версий функций factorial и fib при использовании декоратора lru_cache ?

Код:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import timeit

code1 = '''
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
'''

code2 = '''
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
'''

code3 = '''
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
        return product
'''
code4 = '''
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
'''

code5 = '''
from functools import lru_cache
```

```
@lru_cache
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
'''
code6 = '''
from functools import lru_cache
@lru_cache
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
'''
if __name__ == '__main__':
    print('\nРезультат рекурсивго факториала: ',
min(timeit.Timer(setup=code1).repeat(4, 1000)))
    print('Результат рекурсивного числа Фибоначчи:',
min(timeit.Timer(setup=code2).repeat(4, 1000)))
    print('*    ' * 15)
    print('Результат итеративного факториала:',
min(timeit.Timer(setup=code3).repeat(4, 1000)))
    print('Результат итеративного числа Фибоначчи:',
min(timeit.Timer(setup=code4).repeat(4, 1000)))
    print('*    ' * 15)
    print('Результат факториала с декоратором:',
min(timeit.Timer(setup=code5).repeat(4, 1000)))
    print('Результат числа Фибоначчи с декоратором:',
min(timeit.Timer(setup=code6).repeat(4, 1000)))
```

```
C:\Users\Valentina\AppData\Local\Programs\Python\Python38\python.exe C:/Users/

Результат рекурсивго факториала:  1.2828999999998925e-05
Результат рекурсивного числа Фибоначчи: 1.2828999999998925e-05
*    *    *    *    *    *    *    *    *    *    *    *    *    *    *
Результат итеративного факториала: 1.2829999999998398e-05
Результат итеративного числа Фибоначчи: 1.3255999999996215e-05
*    *    *    *    *    *    *    *    *    *    *    *    *    *    *
Результат факториала с декоратором: 1.3256999999995689e-05
Результат числа Фибоначчи с декоратором: 1.2828999999998925e-05

Process finished with exit code 0
```

Рисунок 1 – Пример 1

По полученным результатам можно сделать вывод, что рекурсивная версия вычисления факториала числа выполняется быстрее в сравнении с итеративной версией или с использованием декоратора.

В случае вычисления числа Фибоначчи наибольшее время показал Итеративный метод вычисления.

Пример 2 (рис. 2):

Проработать пример с оптимизацией хвостовых вызовов в Python. С помощью пакета timeit оценить скорость работы функций factorial и fib с использованием интроспекции стека и без использования интроспекции стека.

Код:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


from timeit import timeit


code1 = """
def factorial(n, acc=1):
    if n == 0:
        return acc

    return factorial(n-1, n*acc)
"""
code2 = """
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
"""
code3 = """
class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


def tail_call_optimized(g):
    def func(*args, **kwargs):
        f = sys._getframe()
        while f and f.f_code.co_filename == f:
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func


@tail_call_optimized
def factorial(n, acc=1):
    if n == 0:
        return acc

    return factorial(n-1, n*acc)
"""
code4 = """
class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
```

```
        self.kwargs = kwargs


def tail_call_optimized(g):
    def func(*args, **kwargs):
        f = sys. getframe()
        while f and f.f_code.co_filename == f:
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func


@tail_call_optimized
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
"""
if __name__ == '__main__':
    print("Время выполнения функции factorial(): ",
          timeit(setup=code1, number=10000))
    print("Время выполнения функции factorial() с"
          " использованием интроспекции стека: ",
          timeit(setup=code3, number=10000))
    print("Время выполнения функции fib(): ",
          timeit(setup=code2, number=10000))
    print("Время выполнения функции fib() с"
          " использованием интроспекции стека: ",
          timeit(setup=code4, number=10000))
```

```
C:\Users\Valentina\AppData\Local\Programs\Python\Python38\python.exe C:/Users/Valentina/Documents/
Время выполнения функции factorial():  0.00013042900000000135
Время выполнения функции factorial() с использованием интроспекции стека:  0.0001300009999999907
Время выполнения функции fib():  0.0001300009999999907
Время выполнения функции fib() с использованием интроспекции стека:  0.00015865200000000912

Process finished with exit code 0
```

Рисунок 2 – Пример 2

Выполнение функции факториала происходит быстрее с интроспекцией стека. Выполнение функции для чисел Фибоначчи происходит быстрее без интроспекции стека.

Задание 1 (рис. 3):

Задан список положительных чисел, признаком конца которых служит отрицательное число. Используя рекурсию, подсчитать количество чисел и их сумму.

Код:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
def list_sum(list2):
    """
    возвращает сумму элементов входящего списка
    Вытаскивается первый элемент из списка с помощью
    среза и передается в следующий рекурсивный вызов функции
    """
    if list2 == []:
        return 0
    else:
        summ = list_sum(list2[1:])
        summ = summ + list2[0]
        return summ


def list1():
    """
    Поэлементный ввод значений в список
    возвращает список элементов
    """
    numbers = []
    print("Введите элементы (до 7) списка через Enter\n"
          "Отрицательное число - признак конца ввода")
    for i in range(7):
        element = int(input())
        if element > 0:
            numbers.append(element)
        else:
            break
    return numbers


if __name__ == '__main__':

    p = list1()
    print('Список: ', p)
    print('Длина списка: ', len(p))
    print('Сумма элементов списка: ', list_sum(p))
```

```
C:\Users\Valentina\AppData\Local\Programs\Python\
Введите элементы (до 7) списка через Enter
Отрицательное число - признак конца ввода
43
67
34
-5
Список:  [43, 67, 34]
Длина списка:  3
Сумма элементов списка:  144

Process finished with exit code 0
```

Рисунок 3 – Задание 1

Контрольные вопросы:

1. Для чего нужна рекурсия?

Функция может содержать вызов других функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту функцию. Без разницы, какая функция дала команду это делать.

2. Что называется базой рекурсии?

База рекурсии – это такие аргументы функции, которые делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

3. Самостоятельно изучите, что является стеком программы. Как используется стек программы при вызове функций?

Стек в Python – это линейная структура данных, в которой данные расположены объектами друг над другом. Он хранит данные в режиме LIFO (Last in First Out). Данные хранятся в том же порядке, в каком на кухне тарелки располагаются одна над другой. Мы всегда выбираем последнюю тарелку из стопки тарелок. В стеке новый элемент вставляется с одного конца, и элемент может быть удален только с этого конца.

4. Как получить текущее значение максимальной глубины рекурсии в языке Python?

Чтобы проверить текущие параметры лимита, нужно запустить: sys.getrecursionlimit().

5. Что произойдет если число рекурсивных вызовов превысит максимальную глубину рекурсии в языке Python?

Существует предел глубины возможной рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение RuntimeError.

6. Как изменить максимальную глубину рекурсии в языке Python?

Изменить максимальную глубину рекурсии можно с помощью sys.setrecursionlimit().

7. Каково назначение декоратора lru_cache ?

Декоратор lru_cache является полезным инструментом, который можно использовать для уменьшения количества лишних вычислений. Декоратор оборачивает функцию с переданными в нее аргументами и запоминает возвращаемый результат, соответствующий этим аргументам.

8. Что такое хвостовая рекурсия? Как проводится оптимизация хвостовых вызовов?

Хвостовая рекурсия — частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Подобный вид рекурсии примечателен тем, что может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции. Оптимизация хвостовой рекурсии путём преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах. В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии.

Вывод: при выполнении заданий были приобретены навыки по работе с функциями при написании программ с помощью языка программирования Python.