иМинистерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №12
дисциплины «Программирование на Python»
Вариант 31

Выполнил:
Репкин Александр Павлович
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Р.А., канд. техн. наук,
доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Тема: Рекурсия в языке Python.

Цель: приобрести навыки работы с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.

Порядок выполнения работы:

1. Выполнено задание №7. В нём требовалось проанализировать временные затраты на рекурсивный, итеративный, рекурсивный с использованием декоратора алгоритмы поиска числа Фибоначчи и Факториала. Из полученных данных ярко видно преимущество рекурсивного алгоритма с использованием lru_cache перед итеративным и просто рекурсивным способами. Хуже всего работал просто рекурсивный алгоритм.

```
Recursion Factorial 10: 0.000042
Iterative Factorial 10: 0.000025
Recursion Factorial with lru_cache 10: 0.000007
Recursion Fib 10: 0.001205
Iterative Fib 10: 0.000034
Recursion Fib with lru_cache 10: 0.000006
Recursion Factorial 11: 0.000048
Iterative Factorial 11: 0.000030
Recursion Factorial with lru_cache 11: 0.000003
Recursion Fib 11: 0.001091
Iterative Fib 11: 0.000032
Recursion Fib with lru_cache 11: 0.000003
Recursion Factorial 12: 0.000045
Iterative Factorial 12: 0.000033
Recursion Factorial with lru_cache 12: 0.000003
```

Рисунок 1. Полученный результат задания №7.

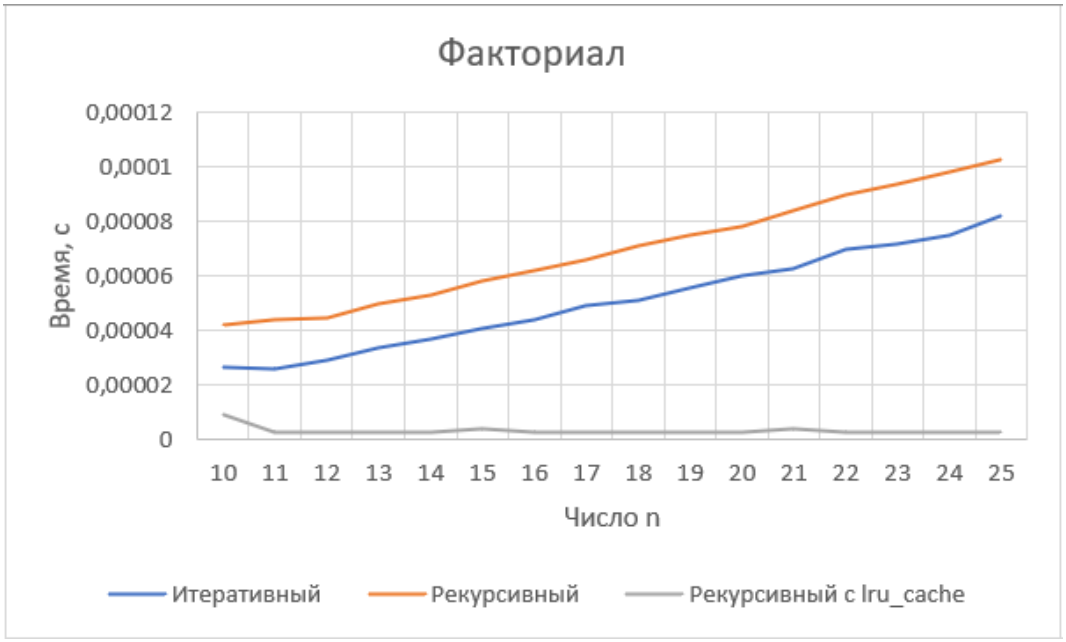

Рисунок 2. Полученная зависимость времени от требуемого факториала числа.

Рисунок 3. Полученная зависимость времени от требуемого числа Фибоначчи.

Рисунок 4. Зависимость Фибоначчи без рекурсивного алгоритма.

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

import timeit
from functools import lru_cache


def recursion_factorial(n):
    """Рекурсивное вычисление факториала."""

    if n == 0:
        return 1
    else:
        return n * recursion_factorial(n - 1)


def recursion_fib(n):
    """Рекурсивное вычисление числа Фибоначчи."""

    if 1 >= n >= 0:
        return n
    else:
        return recursion_fib(n - 2) + recursion_fib(n - 1)


def iterative_factorial(n):
    """Итеративное вычисление факториала."""

    value = 1
    while n > 1:
        value *= n
        n -= 1
    return value


def iterative_fib(n):
    """Итеративное вычисление числа Фибоначчи."""
```

Рисунок 5. Полученный код задания №7.

2. Выполнено индивидуальное задание. В нём требовалось создать программу, создающую все возможные комбинации закрывающихся и открывающихся скобочек относительно введённого пользователем значения пар (Например, если введено 3, то создаются комбинации из трёх пар: ((())), (()()), ()(()), (())(), ()()() – 5 комбинаций). Для выполнения использована рекурсивная функция, анализирующая количество оставшихся открывающих и закрывающих скобочек.

```
Good day! Please, enter amount of pairs - 3
All possible variants are: ((())) ; (()()) ; (())() ; ()(()) ; ()()()
```

Рисунок 6. Полученный результат индивидуального задания.

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

def create_brackets(combination, open_amount, close_amount, variants):
    """
    Данная рекурсивная функция генерирует все комбинации закрывающихся
    и открывающихся скобок. На вход она получает текущую комбинацию
    (это combination), оставшееся количество неиспользованных скобок
    (Открывающиеся - open_amount, Закрывающиеся - close_amount) и
    все полученные варианты (variants).
    """

    # Если не осталось доступных для исп. открывающихся и закрывающихся
    # скобок, то добавляем полученную комбинацию.
    if open_amount == 0 and close_amount == 0:
        variants.append(combination)
    # Если ещё есть открывающиеся, то добавляем их и запускаем с уменьшенным кол.
    if open_amount > 0:
        create_brackets(
            combination + "(", open_amount - 1, close_amount, variants)
    # Если ещё есть закрывающиеся и их меньше, чем открывающихся,
    # то добавляем их и запускаем с уменьшенным кол.
    if close_amount > 0 and open_amount < close_amount:
        create_brackets(combination + ")", open_amount,
                        close_amount - 1, variants)
    return variants


if __name__ == '__main__':
    """Получение количества пар от пользователя и вызов функции."""
    n = int(input("Good day! Please, enter amount of pairs - "))
    result = []
    result = create_brackets("", n, n, result)
    print("All possible variants are:", " ; ".join(result))

```

Рисунок 7. Полученный код индивидуального задания.

3. Ответы на вопросы.

1) Для чего нужна рекурсия?

Ответ: рекурсии используются, когда задача может быть разбита на более мелкие подзадачи того же типа. При их использовании функция вызывает саму себя, передавая новому варианту иные входные значения.

2) Что называется базой рекурсии?

Ответ: база рекурсии - условие, при котором рекурсивные вызовы функции прекращаются и начинается возврат из рекурсивных вызовов. Такие аргументы функции делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

3) Что является стеком программы? Как используется стек программы при вызове функции?

Ответ: стек программы является структурой данных, хранящей информацию о вызовах функций во время выполнения программы. Информация последовательно добавляется в “стопку” , каждый новый объект помещается поверх предыдущего, а извлекаются объекты в обратном порядке,

начиная с верхнего. При вызове функции, информация о текущем её состоянии (локальные переменные, адрес возврата), помещается в стек. Когда функция завершает выполнение, информация извлекается из стека, и управление передается обратно вызывающей функции. Это позволяет программе возвращаться к предыдущему состоянию после завершения выполнения вызванной функции.

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

Ответ: чтобы получить максимальную глубину рекурсии, достаточно использовать функцию sys.getrecursionlimit(), возвращающую максимальное количество рекурсивных вызовов, которое может быть выполнено до возникновения ошибки "RecursionError".

5) Что произойдёт, если число рекурсивных вызовов превысит максимальную глубину рекурсии в языке Python?

Ответ: при попытке превысить допустимую глубину рекурсий, будет выдана ошибка “RecursionError”.

6) Как изменить максимальную глубину рекурсии в Python?

Ответ: функция setrecursionlimit() изменяет максимальную допустимую глубину рекурсий.

7) Каково назначение декоратора lru_cache?

Ответ: декоратор lru_cache кэширует результаты функции в соответствии с последними аргументами вызова. Если функция вызывается с теми же аргументами, что и ранее, результат возвращается из кэша, вместо того чтобы вычислять его снова. Это существенно улучшает производительность рекурсивных функций.

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

Ответ: хвостовая рекурсия - частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Для оптимизации хвостовой рекурсии компилятор или

интерпретатор может заменить рекурсивный вызов на цикл, что позволяет избежать увеличения стека вызовов. Это позволяет снизить использование памяти и улучшить производительность программ.

Вывод: в ходе выполнения лабораторной работы были приобретены навыки работы с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.