

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ
ФЕДЕРАЦИИ**

**Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**Кафедра инфокоммуникаций
«Рекурсия в языке Python»**

**Отчет по лабораторной работе № 2.9
по дисциплине «Основы программной инженерии»**

Выполнил студент группы ПИЖ-б-о-21-1
Коновалова В.Н. « » 2022г.

Подпись студента _____

Работа защищена « » _____ 20__ г.

Проверил Воронкин Р.А. _____
(подпись)

Ставрополь 2022

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

Ход работы:

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия IT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Самостоятельно изучите работу со стандартным пакетом Python timeit. Оцените с помощью этого модуля скорость работы итеративной и рекурсивной версий функций factorial и fib. Во сколько раз измениться скорость работы рекурсивных версий функций factorial и fib при использовании декоратора lru_cache? Приведите в отчет и обоснуйте полученные результаты.

Этот модуль предоставляет простой способ определения времени выполнения небольших фрагментов кода на Python. Он имеет как интерфейс командной строки, так и вызываемый. Это позволяет избежать ряда распространенных ловушек для измерения времени выполнения.

Модуль определяет три удобные функции и открытый класс.

Синтаксис:

timeit.timeit(stmt, setup, timer, number), где

- **stmt**: это код, для которого вы хотите измерить время выполнения. Значение по умолчанию - "pass".
- **setup**: здесь будут детали настройки, которые необходимо выполнить перед stmt. Значение по умолчанию - "pass".

- **timer:** это будет иметь значение таймера, `timeit()` уже имеет значение по умолчанию, и мы можем его игнорировать.
- **number:** `stmt` будет выполняться в соответствии с номером, указанным здесь. Значение по умолчанию - 1000000.

Для работы с `timeit()` нам нужно импортировать соответствующий модуль.

Важно, модулем `timeit` ваш код выполняется в другом пространстве имен. Таким образом, он не распознает функции, которые вы определили в своем глобальном пространстве имен. Для того, чтобы `timeit` распознавал ваши функции, вам необходимо импортировать его в то же пространство имен. Вы можете добиться этого, передав `from __main__ import func_name` аргументу `setup`

Код Рекурсия:

```
# -*- coding: utf-8 -*-
# Рекурсия

import timeit

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

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

if __name__ == '__main__':
    r_fib = fib(15)
    r_factorial = factorial(15)

    print("Рекурсивная функция factorial:")
    print(timeit.timeit("r_factorial", setup="from __main__ import r_factorial"))
    print("Рекурсивная функция fib:")
    print(timeit.timeit("r_fib", setup="from __main__ import r_fib"))
```

```
C:\Users\vika1\AppData\Local\Microsoft\WindowsAp
Рекурсивная функция factorial:
0.019896200000000003
Рекурсивная функция fib:
0.025868099999999999

Process finished with exit code 0
```

Рисунок 1 – Результат работы программы

Код Рекурсия с @lru_cache :

```
# -*- coding: utf-8 -*-
# Рекурсия с @lru_cache

from functools import lru_cache
import timeit

@lru_cache(maxsize=10)
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)

@lru_cache(maxsize=10)
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)

if __name__ == '__main__':
    r_fib = fib(15)
    r_factorial = factorial(15)

    print("Рекурсивная функция factorial с lru_cache:")
    print(timeit.timeit("r_factorial", setup="from __main__ import
r_factorial"))
    print("Рекурсивная функция fib с lru_cache:")
    print(timeit.timeit("r_fib", setup="from __main__ import r_fib"))
```

```
C:\Users\vika1\AppData\Local\Microsoft\WindowsApps\p
Рекурсивная функция factorial с lru_cache:
0.025658399999999998
Рекурсивная функция fib с lru_cache:
0.0296198
```

Рисунок 2 – Результат работы программы

Исходя из результатов мы видим, что рекурсивная функция выполняется медленнее итеративной, при этом использование декоратора `lru_cache` позволяет сократить время работы рекурсивной функции в 10-11 раз.

Код Итеративная версия:

```
# -*- coding: utf-8 -*-
# Итеративная версия

import timeit

def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a

if __name__ == '__main__':
    r_fib = fib(15)
    r_factorial = factorial(15)

    print("Итеративная функция factorial: ")
    print(timeit.timeit("r_factorial", setup="from __main__ import r_factorial"))
    print("Итеративная функция fib: ")
    print(timeit.timeit("r_fib", setup="from __main__ import r_fib"))
```

```

C:\Users\vika1\AppData\Local\Microsoft\WindowsApps\python.exe "C
Итеративная функция factorial:
0.019706099999999997
Итеративная функция fib:
0.0206543

Process finished with exit code 0

```

Рисунок 3 – Результат работы программы

8. Самостоятельно проработайте пример с оптимизацией хвостовых вызовов в Python. С помощью пакета `timeit` оцените скорость работы функций `factorial` и `fib` с использованием интроспекции стека и без использования интроспекции стека. Приведите полученные результаты в отчет.

Код:

```

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

# Эта программа показывает работу декоратора, который производит оптимизацию
# хвостового вызова. Он делает это, вызывая исключение, если оно является его
# прародителем, и перехватывает исключения, чтобы вызвать стек.

import sys
import timeit

class TailRecurseException(Exception):
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
    """
    Эта программа показывает работу декоратора, который производит оптимизацию
    хвостового вызова. Он делает это, вызывая исключение, если оно является
    его
    прародителем, и перехватывает исключения, чтобы подделать оптимизацию
    хвоста.
    Эта функция не работает, если функция декоратора не использует хвостовой
    вызов.
    """
    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code ==
f.f_code:

```

```

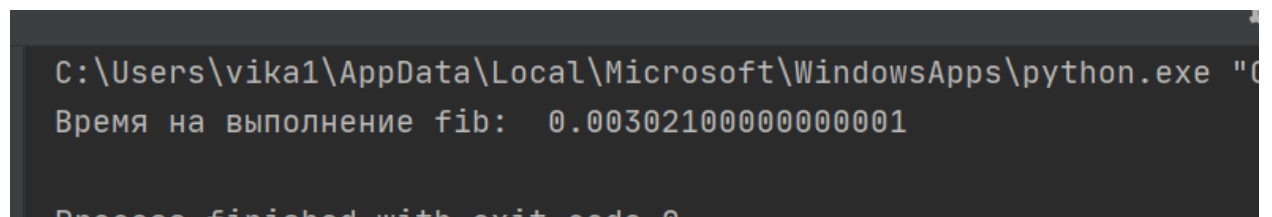
        raise TailRecurseException(args, kwargs)
    else:
        while True:
            try:
                return g(*args, **kwargs)
            except TailRecurseException as e:
                args = e.args
                kwargs = e.kwargs

func.__doc__ = g.__doc__
return func

@tail_call_optimized
def fib(i, current=0, nextt=1):
    if i == 0:
        return current
    else:
        return fib(i - 1, nextt, current + nextt)

if __name__ == '__main__':
    start_time = timeit.default_timer()
    fib(100)
    print("Время на выполнение fib: ", timeit.default_timer() - start_time)
    start_time = timeit.default_timer()

```



```

C:\Users\vika1\AppData\Local\Microsoft\WindowsApps\python.exe "C:\Users\vika1\AppData\Local\Microsoft\WindowsApps\python.exe"
Время на выполнение fib: 0.003021000000000001
Process finished with exit code 0

```

Рисунок 3 – Результат работы программы

Как мы можем увидеть, посмотрев на результат работы, сокращение времени выполнения после оптимизации вовсе нет.

9. Выполните индивидуальные задания. Приведите в отчете скриншоты работы программ решения индивидуального задания.

Создайте рекурсивную функцию, печатающую все возможные перестановки для целых чисел от 1 до N.

```

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

# Создайте рекурсивную функцию, печатающую все возможные перестановки для
целых
# чисел от 1 до N.

def permutation(s):

```

```

"""
    Перестановка чисел
"""
if len(s) == 1:
    return [s]
else:
    a = s[0] # первый элемент списка
    p = permutation(s[1:]) # все перестановки, для последовательности
    без 1-го эл
    r = [] # создаем массив для записи перестановок
    for pp in p: # вставляем a в каждую возможную позицию каждой
перестановки хвоста
        for i, ithem in enumerate(pp):
            tmp = pp[0:i] + [a] + pp[i:]
            r.append(tmp)
        r.append(pp + [a])
    return r

def main():
    """
    Главная функция программы
    """
    n = int(input("n="))
    print(permutation([i for i in range(1, n + 1)]))

if __name__ == '__main__':
    main()

```

```

C:\Users\vika1\AppData\Local\Microsoft\WindowsApps\python.exe "C:\Use
n=4
[[1, 2, 3, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1], [1, 3, 2, 4]
Process finished with exit code 0

```

Рисунок 4 – Результат работы программы

10. Зафиксируйте сделанные изменения в репозитории.

Вопросы для защиты работы

1. Для чего нужна рекурсия?

Рекурсия подразумевает более компактный вид записи выражения. Обычно это зависимость процедур (функций, членов прогресс и т.д.) соседних порядковых номеров. Некоторые зависимости очень сложно

выразить какойлибо формулой, кроме как рекурсивной. Рекурсия незаменима в ряде случаев при программировании замкнутых циклов.

2. Что называется базой рекурсии?

Если ветвь же приводит к очевидному результату и решение не требует дальнейших вложенных вызовов, эта ветвь называется базой рекурсии.

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

Стек хранит информацию для возврата управления из подпрограмм в программу и для возврата в программу из обработчика прерывания. При вызове подпрограммы или возникновении прерываний, в стек заносится адрес возврата – адрес в памяти следующей инструкции приостановленной программы и управление передаётся подпрограмме или подпрограмме обработчику.

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?

Программа выдаст ошибку: `RuntimeError: Maximum Recursion Depth Exceeded`

6. Как изменить максимальную глубину рекурсии в языке Python?

Изменить максимальную глубины рекурсии можно с помощью `sys.setrecursionlimit(limit)`.

7. Каково назначение декоратора `lru_cache` ?

Декоратор можно использовать для уменьшения количества лишних вычислений.

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.