

**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

Гребенкин Е. А. « » 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.

**Выполнение 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.

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` в качестве аргументу `setup`

```

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

import timeit
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)
    def factorial_s(n):
        if n == 0:
            return 1
        elif n == 1:
            return 1
        else:
            return n * factorial_s(n - 1)
    def factorial_i(n):
        product = 1
        while n > 1:
            product *= n
            n -= 1
        return product

code_1 = """
from __main__
import factorial
n = 12
"""
code_2 = """
from __main__
import factorial_s
n = 12
"""
code_3 = """
from __main__
import factorial_i
n = 12
"""
if __name__ == "__main__":
    print("Рекурсивная функция с lru_cache: ")
    print(timeit.timeit(setup=code_1, stmt="factorial(n)", number=10000))
    print("Рекурсивная функция: ")
    print(timeit.timeit(setup=code_2, stmt="factorial_s(n)", number=10000))
    print("Итеративная функция: ")
    print(timeit.timeit(setup=code_3, stmt="factorial_i(n)", number=10000))

```

```
Рекурсивная функция с lru_cache:  
0.0003013749956153333  
Рекурсивная функция:  
0.005159958032891154  
Итеративная функция:  
0.0032879170030355453
```

Рисунок 5 – Результат работы программы

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

```

import timeit
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)
def fib_s(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_s(n - 2) + fib_s(n - 1)
def fib_i(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a

code_1 = """ from __main__
import fib n
= 12 """
code_2 = """
from __main__ import fib_s n
= 12 """
code_3 = """
from __main__ import fib_i n
= 12 """
if __name__ == "__main__":
    print("Рекурсивная функция с lru_cache: ")
    print(timeit.timeit(setup=code_1, stmt="fib(n)", number=10000))
    print("Рекурсивная функция: ")
    print(timeit.timeit(setup=code_2, stmt="fib_s(n)", number=10000))
    print("Итеративная функция: ")
    print(timeit.timeit(setup=code_3, stmt="fib_i(n)", number=10000))

```

Рекурсивная функция с lru\_cache:

0.0003802919527515769

Рекурсивная функция:

0.16820741596166044

Итеративная функция:

0.003111166995950043

### Рисунок 6 – 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.

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.

```

#!/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 def
factorial(n, acc=1):

```



```

    if n == 0:
        return acc
    return factorial(n - 1, n*acc)

@tail_call_optimized def
factorial_o(n, acc=1):
    if n == 0:
        return acc
    return factorial(n - 1, n * acc)
def fib(i, current=0, next=1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)

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

code_1 = """    from __main__
import factorial n
= 100 """
code_2 = """    from __main__ import factorial_o n
= 100 """
code_3 = """    from __main__ import fib n
= 10 """
code_4 = """    from __main__ import fib_o n =
10
"""
if __name__ == '__main__':
    print("Рекурсивная функция (factorial):")
    print(timeit.timeit(setup=code_1, stmt="factorial(n)", number=10000))
    print("Рекурсивная функция с @tail_call_optimized(factorial):")
    print(timeit.timeit(setup=code_2, stmt="factorial_o(n)", number=10000))
    print("Рекурсивная функция (fib):")
    print(timeit.timeit(setup=code_3, stmt="fib(n)", number=10000))
    print("Рекурсивная функция с @tail_call_optimized (fib):")
    print(timeit.timeit(setup=code_4, stmt="fib_o(n)", number=10000))

```

```

Рекурсивная функция (factorial):
0.06858458300121129
Рекурсивная функция с @tail_call_optimized(factorial):
0.056594041001517326
Рекурсивная функция (fib):
0.004009167023468763
Рекурсивная функция с @tail_call_optimized (fib):
0.11283154203556478

```

Рисунок 7 – Результат работы программы

Сокращения времени выполнения после оптимизации вовсе нет.

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

Даны целые числа  $m$  и  $n$ , где  $0 \leq m \leq n$ , вычислить, используя рекурсию, число сочетаний  $C_n^m$  по формуле:  $C_n^0 = C_n^n = 1$ ,  $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$  при  $0 \leq m \leq n$ .  
Воспользовавшись формулой

$$C_n^m = \frac{n!}{m!(n-m)!} \quad (1)$$

```

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

def C(m, n):
    if m == n or m == 0:
        return 1
    elif 0 <= m <= n:
        return C(m, n - 1) + C(m - 1, n - 1)

if __name__ == '__main__':
    print(C(int(input()),
int(input())))

```

```

C:\Users\dimu7\AppData
enter x: 3.4
enter n: 5
454.35423999999999

```

## Рисунок 8 – 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.