

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

Институт цифрового развития  
Кафедра инфокоммуникаций

**«Рекурсия в языке Python»**

**ОТЧЕТ**  
**по лабораторной работе №12**  
**дисциплины**  
**«Основы программной инженерии»**

Выполнил:

Сотников Андрей Александрович  
2 курс, группа ПИЖ-б-о-21-1,  
09.03.04 «Программная  
инженерия», направленность  
(профиль) «Разработка и  
сопровождение программного  
обеспечения», очная форма  
обучения

---

(подпись)

Проверил:

---

(подпись)

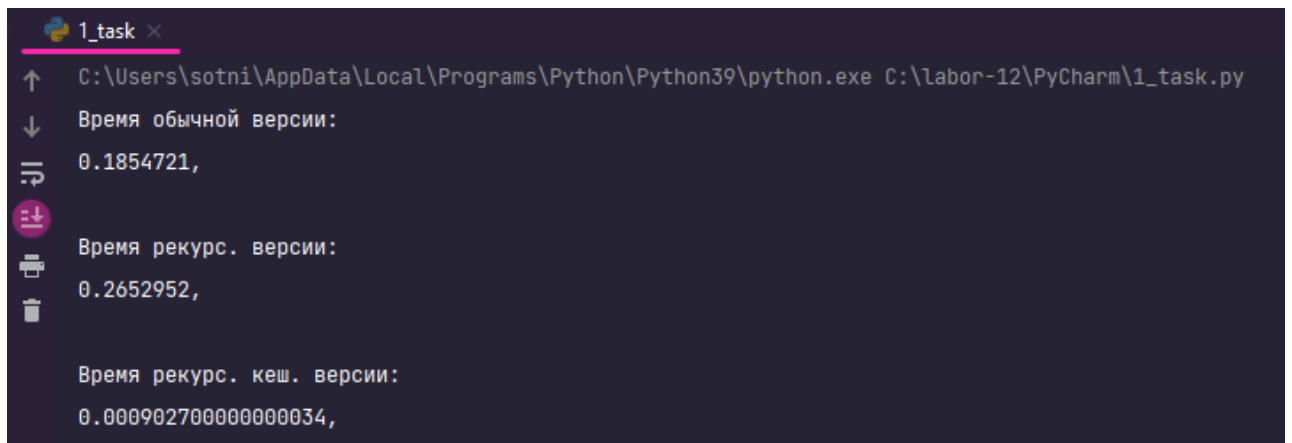
Отчет защищен с оценкой \_\_\_\_\_ Дата защиты \_\_\_\_\_

Ставрополь, 2022 г.

**Задание №1:** 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.

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import ...
5
6
7  def factorial_iteration(n):
8      product = 1
9      while n > 1:
10         product *= n
11         n -= 1
12     return product
13
14
15  def factorial(n):
16      if n == 0 or n == 1:
17         return 1
18      else:
19         return n * factorial(n - 1)
20
21
22  @lru_cache
23  def factorial_cache(n):
24      if n == 0 or n == 1:
25         return 1
26      else:
27         return n * factorial_cache(n - 1)
28
29  if __name__ == "__main__":
30      print("Время обычной версии:")
31      print(f'{timeit.timeit(lambda: factorial_iteration(200), number=10000)}, \n')
32      print("Время рекурс. версии:")
33      print(f'{timeit.timeit(lambda: factorial(200), number=10000)}, \n')
34      print("Время рекурс. кеш. версии:")
35      print(f'{timeit.timeit(lambda: factorial_cache(200), number=10000)}, \n')
```

Рисунок 1 – Код задания №1 (поиск факториала)



The image shows a terminal window titled "1\_task" with a dark background. On the left side, there is a vertical toolbar with icons for running, stepping through code, and other debugging actions. The terminal output is as follows:

```
C:\Users\sotni\AppData\Local\Programs\Python\Python39\python.exe C:\labor-12\PyCharm\1_task.py
Время обычной версии:
0.1854721,
Время рекурс. версии:
0.2652952,
Время рекурс. кеш. версии:
0.000902700000000034,
```

Рисунок 2 – Результат работы кода задания №1 (поиск факториала)

```

1  ▶  #!/usr/bin/env python3
2      # -*- coding: utf-8 -*-
3
4      import timeit
5      from functools import lru_cache
6
7
8      new *
9      def fib_iter(n):
10         a, b = 0, 1
11         while n > 0:
12             a, b = b, a + b
13             n -= 1
14         return a
15
16      AndrejMirrox *
17      def fib_rec(n):
18         if n == 0 or n == 1:
19             return n
20         else:
21             return fib_rec(n - 2) + fib_rec(n - 1)
22
23      new *
24      @lru_cache
25      def fib_rec_lru(n):
26         if n == 0 or n == 1:
27             return n
28         else:
29             return fib_rec_lru(n - 2) + fib_rec_lru(n - 1)
30
31  ▶  if __name__ == "__main__":
32         print("Первый вариант:")
33         print(f'{timeit.timeit(lambda: fib_iter(15), number=10000)}')
34         print("Второй вариант:")
35         print(f'{timeit.timeit(lambda: fib_rec(15), number=10000)}')
36         print("Третий вариант:")
37         print(f'{timeit.timeit(lambda: fib_rec_lru(15), number=10000)}')

```

Рисунок 3 – Код задания №1 (числа Фибоначчи)

```
C:\Users\sotni\AppData\Local\Programs\Python\Python39\python.exe C:\labor-12\PyCharm\2_task.py
Первый вариант:
0.0094273
Второй вариант:
1.4693385
Третий вариант:
0.0007743999999998419
```

Рисунок 4 – Результат работы кода задания №1 (числа Фибоначчи)

**Задание №2:** 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.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import timeit
5
6
7  new *
8  class rec(object):
9      new *
10     def __init__(self, func):
11         self.func = func
12
13     new *
14     def __call__(self, *args, **kwargs):
15         result = self.func(*args, **kwargs)
16         while callable(result): result = result()
17         return result
18
19     new *
20     def call(self, *args, **kwargs):
21         return lambda: self.func(*args, **kwargs)
22
23
24     new *
25     @rec
26     def factorial_opt(n, acc = 1):
27         if n == 0:
28             return acc
29         return factorial(n - 1, n * acc)
30
31
32     new *
33     def factorial(n, acc = 1):
34         if n == 0:
35             return acc
36         return factorial(n - 1, n * acc)
37
38
39  if __name__ == "__main__":
40     print("Первый вариант:")
41     print(f'{timeit.timeit(lambda: factorial_opt(250), number=10000)}')
42     print("Второй вариант:")
43     print(f'{timeit.timeit(lambda: factorial(15), number=10000)}')

```

Рисунок 5 – Код задания №2

```
Run: 3_task x
C:\Users\sotni\AppData\Local\Programs\Python\Python39\python.exe C:\labor
Первый вариант:
0.4205796
Второй вариант:
0.016612699999999998
```

Рисунок 6 – Результат работы кода задания №2

### Индивидуальное задание:

Напишите программу вычисления функции Аккермана для всех неотрицательных целых аргументов  $m$  и  $n$ :

$$A(m, n) = \begin{cases} A(0, n) = n + 1 \\ A(m, 0) = A(m - 1, 1), & m \\ A(m, n) = A(m - 1, A(m, n - 1)), & m, n > 0. \end{cases}$$

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  import sys
6
7  sys.setrecursionlimit(30000)
8
9
10 new *
11 def A(m, n):
12     if m == 0:
13         return n + 1
14     elif m > 0 and n == 0:
15         return A(m - 1, 1)
16     elif m > 0 and n > 0:
17         return A(m - 1, A(m, n - 1))
18
19 if __name__ == "__main__":
20
21     in_m = int(input("Введите число m:"))
22     in_n = int(input("Введите число n:"))
23     for mi in range(in_m + 1):
24         for ni in range(in_n + 1):
25             result = A(mi, ni)
26             print(f"A({mi}, {ni}) = {result}")
27
```

Рисунок 7 – Код программы индивидуального задания

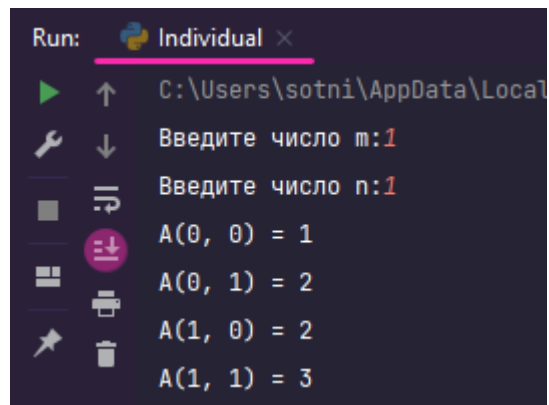


Рисунок 8 – Результат работы программы индивидуального задания



## Контрольные вопросы

1. Для чего нужна рекурсия?

Функция может содержать вызов других функций. В том числе процедура может вызвать саму себя.

2. Что называется базой рекурсии?

У рекурсии, как и у математической индукции, есть база — аргументы, для которых значения функции определены

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

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

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

Функция `sys.getrecursionlimit()` возвращает текущее значение предела рекурсии, максимальную глубину стека интерпретатора Python.

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

Ошибка `RuntimeError`

6. Как изменить максимальную глубину рекурсии в языке Python?

С помощью функции `sys.setrecursionlimit()` модуля `sys`

## 7. Каково назначение декоратора lru\_cache ?

Декоратор `@lru_cache()` модуля `functools` оборачивает функцию с переданными в нее аргументами и запоминает возвращаемый результат соответствующий этим аргументам. Такое поведение может сэкономить время и ресурсы, когда дорогая или связанная с вводом/выводом функция периодически вызывается с одинаковыми аргументами.

## 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. Оптимизация хвостовой рекурсии выглядит так:

```
class recursion(object):
    "Can call other methods inside..."
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        result = self.func(*args, **kwargs)
        while callable(result): result = result()
        return result

    def call(self, *args, **kwargs):
        return lambda: self.func(*args, **kwargs)

@recursion
def sum_natural(x, result=0):
    if x == 0:
        return result
    else:
        return sum_natural.call(x - 1, result + x)

# Даже такой вызов не заканчивается исключением
# RuntimeError: maximum recursion depth exceeded
print(sum_natural(1000000))
```