

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

Кафедра инфокоммуникаций

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

Выполнил студент группы ПИЖ-б-о-22-1

Душин Александр Владимирович.

Подпись студента _____

Работа защищена « » _____ 20__ г.

Проверил Воронкин Р.А. _____
(подпись)

Ставрополь 2023

Тема: Рекурсия в языке Python.

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

Ход выполнения работы:


1. Создать общедоступный репозиторий на GitHub с использованием лицензии MIT и язык программирования Python:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner *

 MrPlatynum ▾

Repository name *

ProgrammEngineering12

✓ ProgrammEngineering12 is available.

Great repository names are short and memorable. Need inspiration? How about [glowing-computing-machine](#) ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: Python ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: MIT License ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)



You are creating a public repository in your personal account.

Create repository

Рисунок 1 – Создание общедоступного репозитория на GitHub с заданными настройками

```
Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents
$ git clone https://github.com/MrPlatynum/ProgrammEngineering12.git
Cloning into 'ProgrammEngineering12'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

Рисунок 2 – Клонирование созданного репозитория на локальный компьютер

```
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/

# pytype static type analyzer
.pytype/

# Cython debug symbols
cython_debug/

# PyCharm
# JetBrains specific template is maintained in a separate JetBrains.gitignore that can
# be found at https://github.com/github/gitignore/blob/main/Global/JetBrains.gitignore
# and can be added to the global gitignore or merged into this file. For a more nuclear
# option (not recommended) you can uncomment the following to ignore the entire idea folder.
.idea/
```

Рисунок 3 – файл .gitignore

```
Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (main)
$ git checkout -b develop
Switched to a new branch 'develop'

Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (develop)
$
```

Рисунок 4 – организация репозитория в соответствии с моделью ветвления git flow

2. 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:

Листинг программы:

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

```

import timeit
from functools import lru_cache

# Итеративная версия чисел Фибоначчи
def fib_iterative(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

# Рекурсивная версия чисел Фибоначчи
def fib_recursive(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_recursive(n - 2) + fib_recursive(n - 1)

# Итеративная версия факториала
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Рекурсивная версия факториала
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

# Измерение времени выполнения для итеративных функций
print("Итеративный факториал:", timeit.timeit('factorial_iterative(10)',
globals=globals()))
print("Итеративные числа Фибоначчи:", timeit.timeit('fib_iterative(10)',
globals=globals()))

# Измерение времени выполнения для рекурсивных функций
print("Рекурсивный факториал:", timeit.timeit('factorial_recursive(10)',
globals=globals()))
print("Рекурсивные числа Фибоначчи:", timeit.timeit('fib_recursive(10)',
globals=globals()))

# Декорируем рекурсивные функции для использования кэша
@lru_cache(maxsize=None)
def fib_recursive_cached(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_recursive_cached(n - 2) + fib_recursive_cached(n - 1)

@lru_cache(maxsize=None)

```

```
def factorial_recursive_cached(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive_cached(n - 1)

# Измерение времени выполнения для кэшированных рекурсивных функций
print("Кэшированный рекурсивный факториал:",
timeit.timeit('factorial_recursive_cached(10)', globals=globals()))
print("Кэшированные рекурсивные числа Фибоначчи:",
timeit.timeit('fib_recursive_cached(10)', globals=globals()))
```

```
"C:\Program Files\Python312\python.exe" "C:/Users/Alexander/Desktop/Универ/3 семестр/Основы программной инженерии/ЛР12_ДушинAB/task1.py"
Итеративный факториал: 0.37613940000301227
Итеративные числа Фибоначчи: 0.3421911000041291
Рекурсивный факториал: 0.6624605000251904
Рекурсивные числа Фибоначчи: 9.639830499974778
Кэшированный рекурсивный факториал: 0.047434200008865446
Кэшированные рекурсивные числа Фибоначчи: 0.045101000025169924
```

Рисунок 5 – Вывод программы

Результаты измерений показывают, что итеративные версии функций (factorial_iterative и fib_iterative) выполняются значительно быстрее по сравнению с их рекурсивными аналогами (factorial_recursive и fib_recursive). Время выполнения итеративных функций значительно меньше, что обусловлено отсутствием избыточных вызовов функций и более прямым выполнением алгоритма без переключений контекста вызовов.

Рекурсивные версии функций без кэширования (factorial_recursive и fib_recursive) имеют высокую сложность из-за множественных вызовов функций с одинаковыми аргументами. Это приводит к повторным вычислениям и долгому времени выполнения, особенно для больших значений n .

Однако, когда используется декоратор lru_cache для кэширования рекурсивных вызовов (factorial_recursive_cached и fib_recursive_cached), время выполнения резко сокращается. Это связано с тем, что результаты вычислений сохраняются в кэше, и повторные вызовы функций с теми же аргументами возвращают сохраненные результаты без дополнительных вычислений. Поэтому кэширование уменьшает время выполнения рекурсивных функций на несколько порядков, делая их выполнение близким к итеративным версиям.

Итак, результаты подтверждают, что кэширование с помощью lru_cache

значительно повышает производительность рекурсивных функций за счет устранения избыточных вычислений и сохранения результатов для повторного использования.

3. 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.

Листинг программы:

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

import timeit

# Декоратор для оптимизации хвостовой рекурсии
class TailRecurseException(BaseException):
    def __init__(self, args, kwargs):
        """
        Инициализирует экземпляр TailRecurseException.

        Аргументы:
        - args (tuple): Аргументы для функции.
        - kwargs (dict): Именованные аргументы для функции.
        """
        self.args = args
        self.kwargs = kwargs

def tail_recursive(func):
    def wrapper(*args, **kwargs):
        """
        Обёртка для функции для имитации хвостовой рекурсии.

        Аргументы:
        - args: Аргументы для функции.
        - kwargs: Именованные аргументы для функции.
        """
        while True:
            try:
                return func(*args, **kwargs)
            except TailRecurseException as e:
                args = e.args
                kwargs = e.kwargs
                continue

    return wrapper

@tail_recursive
def factorial(n, accumulator=1):
    """
    Рекурсивная функция для вычисления факториала.
```

```

Аргументы:
- n (int): Число для вычисления факториала.
- accumulator (int): Аккумулятор для промежуточных результатов.

Возвращает:
- int: Факториал числа n.
"""
if n == 0:
    return accumulator
else:
    raise TailRecurseException((n - 1, n * accumulator), {})

@tail_recursive
def fib(n, a=0, b=1):
    """
    Рекурсивная функция для вычисления чисел Фибоначчи.

    Аргументы:
    - n (int): Число в последовательности Фибоначчи.
    - a (int): Первое число в последовательности.
    - b (int): Второе число в последовательности.

    Возвращает:
    - int: n-ное число в последовательности Фибоначчи.
    """
    if n == 0:
        return a
    else:
        raise TailRecurseException((n - 1, b, a + b), {})

if __name__ == '__main__':
    # Оценка времени выполнения функций
    print("Время выполнения рекурсивной функции factorial:",
timeit.timeit(lambda: factorial(20), number=10000))
    print("Время выполнения рекурсивной функции fib:",
timeit.timeit(lambda: fib(20), number=10000))

```

```

"C:\Program Files\Python312\python.exe" "C:/Users/Alexander/Desktop/Универ/3 семестр/Основы программной инженерии/ЛР12_ДушинAB/task2.py"
Время выполнения рекурсивной функции factorial: 0.1422547000038321
Время выполнения рекурсивной функции fib: 0.130904500001634

```

Рисунок 6 – Вывод программы

4. Выполним индивидуальные задания:

```
individual1.py
1 def print_number_sum_representations(n, i=1, output=""):
2     """
3     Печатает все возможные представления числа n в виде суммы других натуральных чисел.
4
5     Аргументы:
6     - n (int): Натуральное число, для которого ищутся представления в виде суммы.
7     - i (int): Текущее число для добавления к сумме.
8     - output (str): Строка, представляющая текущее представление суммы.
9     """
10    if n == 0:
11        print(output)
12        return
13    if n < 0:
14        return
15    while i <= n:
16        print_number_sum_representations(n - i, i, f"{output} + {i}" if output else str(i))
17        i += 1
18
19
20 if __name__ == '__main__':
21     n = int(input("Введите число, которое хотите представить в виде сумм: "))
22     print_number_sum_representations(n)
23
```

Рисунок 7 – Решение индивидуального задания

```
"C:\Program Files\Python312\python.exe" "C:/Users/Alexander/Desktop/Универ/3 семестр/Основы программной инженерии/ЛР12_ДушинАВ/individual1.py"
Введите число, которое хотите представить в виде сумм: 5
1 + 1 + 1 + 1 + 1
1 + 1 + 1 + 2
1 + 1 + 3
1 + 2 + 2
1 + 4
2 + 3
5
```

Рисунок 8 – Вывод программы

5. Зафиксируем сделанные изменения, сольем ветки и отправим на удаленный репозиторий:

```
Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (develop)
$ git log --oneline
8654c64 (HEAD -> develop) Финальные изменения
28c93ac (origin/main, origin/HEAD, main) Initial commit
```

Рисунок 9 – Коммиты ветки develop во время выполнения лабораторной работы


```

Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (develop)
$ git checkout main
Switched to branch 'main'
M       .gitignore
Your branch is up to date with 'origin/main'.

Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (main)
$ git merge develop
Updating 28c93ac..8654c64
Fast-forward
 individual1.py | 22 ++++++
 task1.py       | 68 ++++++
 task2.py       | 81 ++++++
 3 files changed, 171 insertions(+)
 create mode 100644 individual1.py
 create mode 100644 task1.py
 create mode 100644 task2.py

```

Рисунок 10 – Слияние ветки develop в ветку main

```

Alexander@DESKTOP-IUJLQQ3 MINGW64 ~/Documents/ProgrammEngineering12 (main)
$ git push origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 12 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 2.34 KiB | 2.34 MiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/MrPlatynum/ProgrammEngineering12.git
 28c93ac..8654c64  main -> main

```

Рисунок 11 – Отправка на удаленный репозиторий

Ответы на контрольные вопросы:

1. Для чего нужна рекурсия?

Рекурсия позволяет функции вызывать саму себя. Это полезно для решения задач, которые могут быть выражены через более простые случаи этой же задачи.

2. Что называется базой рекурсии?

База рекурсии – это условие, при котором рекурсивные вызовы завершаются, обычно это самый простой случай задачи, который не требует дальнейшего разбиения.

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

Стек программы – это структура данных, которая используется для хранения временных данных вызовов функций. При вызове функции данные помещаются в стек, а при завершении функции они удаляются из стека. Это позволяет программе отслеживать, откуда вернуться после завершения каждого вызова функции.

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

Можно получить текущее значение максимальной глубины рекурсии с помощью sys модуля:

```
import sys  
print(sys.getrecursionlimit())
```

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

Превышение максимальной глубины рекурсии вызовет ошибку RecursionError.

6. Как изменить максимальную глубину рекурсии в Python?

Максимальную глубину рекурсии можно изменить с помощью sys.setrecursionlimit(new_limit). Однако, изменение этого значения может повлиять на работу программы, поскольку слишком большая глубина рекурсии

может привести к переполнению стека и ошибкам.

7. Каково назначение декоратора `lru_cache`?

`lru_cache` – это декоратор, который кэширует результаты вызова функции в памяти. Это позволяет избежать повторных вычислений при повторных вызовах функции с теми же аргументами, что улучшает производительность.

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

Хвостовая рекурсия – это вид рекурсии, где рекурсивный вызов функции является последней операцией перед возвратом из функции. Оптимизация хвостовых вызовов (`tail call optimization`, TCO) позволяет некоторым интерпретаторам, таким как определенные реализации Python, оптимизировать использование памяти при хвостовой рекурсии, избегая увеличения стека вызовов.