

Министерство 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 №2.9
дисциплины «Основы программной инженерии»

Выполнил:
Соколов Михаил Романович
2 курс, группа ПИЖ-б-о-22-1,
09.03.04 «Программная инженерия»,
направленность (профиль) «Разработка
и сопровождение программного
обеспечения», очная форма обучения

(подпись)

Руководитель практики:
Богданов С.С., ассистент кафедры
инфокоммуникаций

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2023 г.

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, в котором будет использована лицензия MIT и .gitignore файл для языка 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 ().*

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner * PoTtaTto ▾ / **Repository name *** fundamentals_of_software

✔ fundamentals_of_software_engineering_lab2_9 is available.

Great repository names are short and memorable. Need inspiration? How about [fictional-train](#) ?

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.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

① You are creating a public repository in your personal account.

Create repository

Рисунок 1 – Создание репозитория с заданными настройками

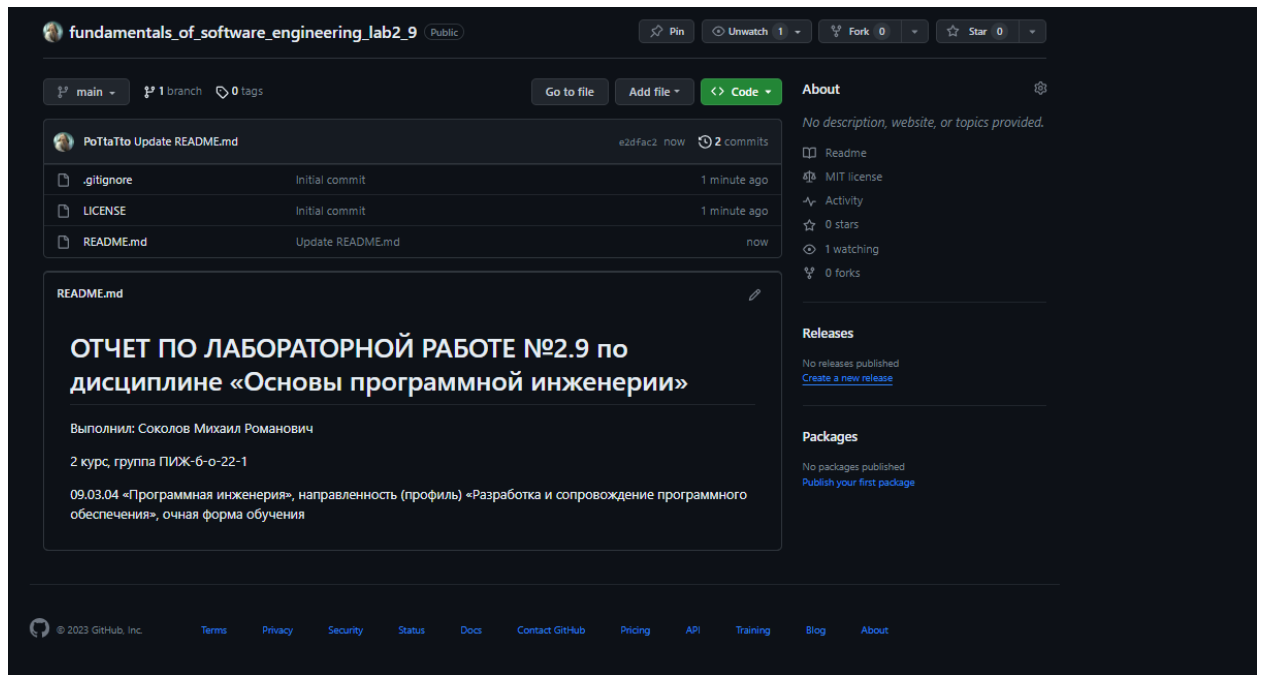


Рисунок 2 – Созданный репозиторий

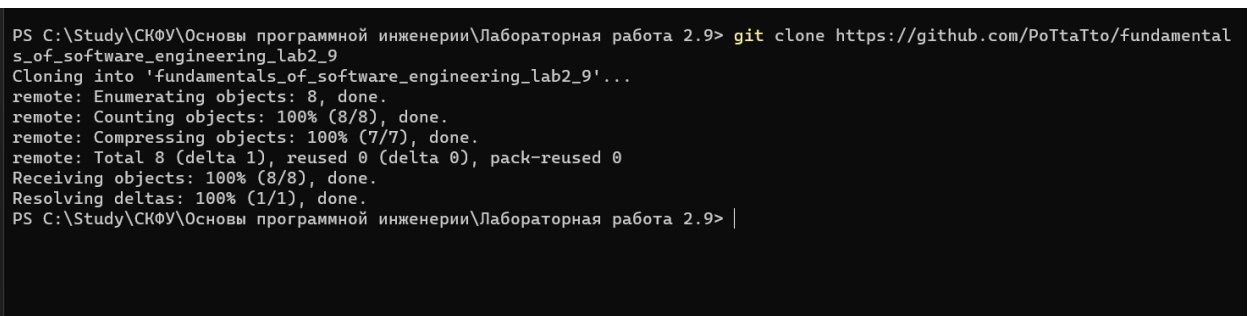


Рисунок 3 – Клонирование репозитория

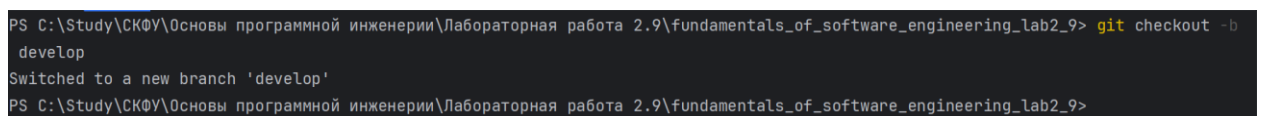


Рисунок 4 – Создание ветки develop, где будут происходить изменения проекта до его полного релиза

```
.gitignore x
1 # Byte-compiled / optimized / DLL files
2 __pycache__/
3 *.py[cod]
4 *$py.class
5
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 build/
12 develop-eggs/
13 dist/
14 downloads/
15 eggs/
16 .eggs/
17 lib/
18 lib64/
19 parts/
20 sdist/
21 var/
22 wheels/
23 share/python-wheels/
```

Рисунок 5 – Часть .gitignore файла, созданного GitHub

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:

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

import timeit
from functools import lru_cache

def factorial(n):
    """
    Рекурсивно вычисляет факториал числа n.

    Args:
        n (int): Целое число для вычисления факториала.
```

```

Returns:
- int: Факториал числа n.
"""
if n == 0:
    return 1
else:
    return n * factorial(n - 1)

def fib(n):
    """
    Рекурсивно вычисляет число Фибоначчи для n.

    Args:
    - n (int): Номер числа Фибоначчи.

    Returns:
    - int: Число Фибоначчи.
    """
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)

def factorial_iterative(n):
    """
    Итеративно вычисляет факториал числа n.

    Args:
    - n (int): Целое число для вычисления факториала.

    Returns:
    - int: Факториал числа n.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

def fib_iterative(n):
    """
    Итеративно вычисляет число Фибоначчи для n.

    Args:
    - n (int): Номер числа Фибоначчи.

    Returns:
    - int: Число Фибоначчи.
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for _ in range(n - 1):
            a, b = b, a + b
        return b

```

```

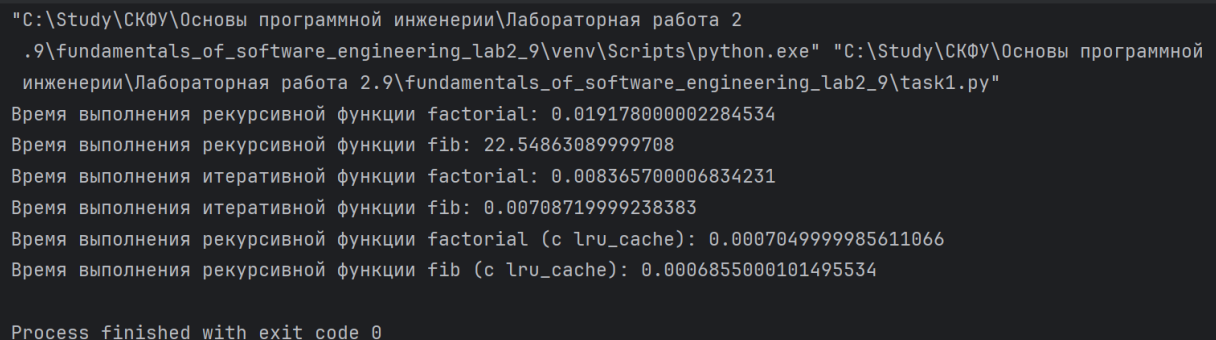
if __name__ == '__main__':
    # Оценка времени выполнения функций
    print("Время выполнения рекурсивной функции factorial:",
timeit.timeit(lambda: factorial(20), number=10000))
    print("Время выполнения рекурсивной функции fib:",
timeit.timeit(lambda: fib(20), number=10000))
    print("Время выполнения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

    # Применение декоратора lru_cache к рекурсивным функциям
    factorial = lru_cache(maxsize=None)(factorial)
    fib = lru_cache(maxsize=None)(fib)

    # Оценка времени выполнения функций с применением lru_cache
    print("Время выполнения рекурсивной функции factorial (с lru_cache):",
timeit.timeit(lambda: factorial(20), number=10000))
    print("Время выполнения рекурсивной функции fib (с lru_cache):",
timeit.timeit(lambda: fib(20), number=10000))

```

Листинг 1 – Код задания



```

"C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2
.9\fundamentals_of_software_engineering_lab2_9\venv\Scripts\python.exe" "C:\Study\СКФУ\Основы программной
инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9\task1.py"
Время выполнения рекурсивной функции factorial: 0.019178000002284534
Время выполнения рекурсивной функции fib: 22.54863089999708
Время выполнения итеративной функции factorial: 0.008365700006834231
Время выполнения итеративной функции fib: 0.00708719999238383
Время выполнения рекурсивной функции factorial (с lru_cache): 0.0007049999985611066
Время выполнения рекурсивной функции fib (с lru_cache): 0.0006855000101495534

Process finished with exit code 0

```

Рисунок 6 – Выполнение кода

Итеративные версии функций `factorial` и `fib` оказались значительно быстрее своих рекурсивных аналогов. Рекурсивные версии без кеширования сильно проигрывают по скорости, так как при каждом вызове рекурсивных функций происходит множество повторных вычислений.

Однако, после применения декоратора `lru_cache` к рекурсивным функциям, их скорость выполнения значительно улучшилась, приблизившись к скорости выполнения итеративных версий функций. Кэширование

(lru_cache) значительно уменьшило количество повторных вычислений, сохраняя результаты предыдущих вызовов функций.

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:

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

import timeit

# Decorator for tail recursion optimization
class TailRecurseException(BaseException):
    def __init__(self, args, kwargs):
        """
        Initializes the TailRecurseException instance.

        Args:
        - args (tuple): Arguments for the function.
        - kwargs (dict): Keyword arguments for the function.
        """
        self.args = args
        self.kwargs = kwargs

    def tail_recursive(func):
        def wrapper(*args, **kwargs):
            """
            Wraps a function to simulate tail recursion.

            Args:
            - args: Arguments for the function.
            - kwargs: Keyword arguments for the function.
            """
            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):
        """
        Recursive function to calculate factorial.

        Args:

```

```

- n (int): Number for factorial calculation.
- accumulator (int): Accumulator for intermediate results.

Returns:
- int: Factorial of n.
"""
if n == 0:
    return accumulator
else:
    raise TailRecurseException((n - 1, n * accumulator), {})

@tail_recursive
def fib(n, a=0, b=1):
    """
    Recursive function to calculate Fibonacci series.

    Args:
    - n (int): Number in the Fibonacci series.
    - a (int): First number in the series.
    - b (int): Second number in the series.

    Returns:
    - int: The n-th Fibonacci number.
    """
    if n == 0:
        return a
    else:
        raise TailRecurseException((n - 1, b, a + b), {})

if __name__ == '__main__':
    # Time evaluation of the functions
    print("Execution time for recursive function factorial:",
timeit.timeit(lambda: factorial(20), number=10000))
    print("Execution time for recursive function fib:",
timeit.timeit(lambda: fib(20), number=10000))

```

Листинг 2 – Код задания

```

"C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2
.9\fundamentals_of_software_engineering_lab2_9\venv\Scripts\python.exe" "C:\Study\СКФУ\Основы программной
инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9\task2.py"
Execution time for recursive function factorial: 0.10065790000953712
Execution time for recursive function fib: 0.09722920000785962

Process finished with exit code 0

```

Рисунок 7 – Выполнение кода

Этот код иллюстрирует использование оптимизации хвостовых вызовов (Tail Call Optimization). Она помогает избежать переполнения стека при

выполнении рекурсивных функций, выполняя их с минимальным использованием памяти.

1) TailRecurseException класс:

- Этот класс наследуется от BaseException и используется для сигнализации о рекурсивном вызове в хвостовой позиции.
- Инициализирует объект TailRecurseException с аргументами args и kwargs.

2) Декоратор tail_recursive:

- Этот декоратор изменяет функции для имитации оптимизации хвостового вызова.
- Возвращает обёртку wrapper, которая обрабатывает вызовы функции и перехватывает исключение TailRecurseException.

3) wrapper функция в tail_recursive:

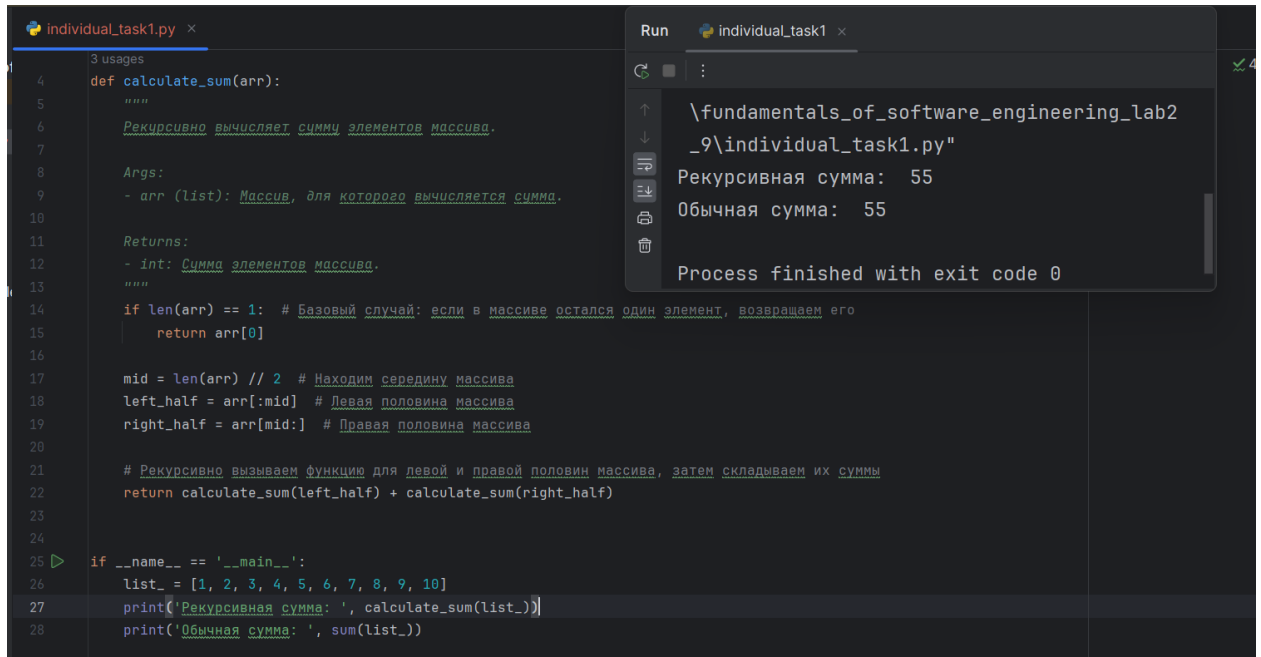
- Выполняет функцию func с предоставленными аргументами и возвращает результат.
- Перехватывает исключение TailRecurseException и использует его аргументы для продолжения рекурсии.

4) factorial и fib функции:

- Это рекурсивные функции для вычисления факториала и чисел Фибоначчи соответственно.
- Обе функции декорированы @tail_recursive, что позволяет использовать оптимизацию хвостового вызова.

4. Индивидуальное задание (вариант №7). Создайте функцию, подсчитывающую сумму элементов массива по следующему алгоритму: массив делится пополам, подсчитываются и складываются суммы элементов в каждой половине. Сумма элементов в половине массива подсчитывается по тому же алгоритму, то есть снова путем деления пополам. Деления

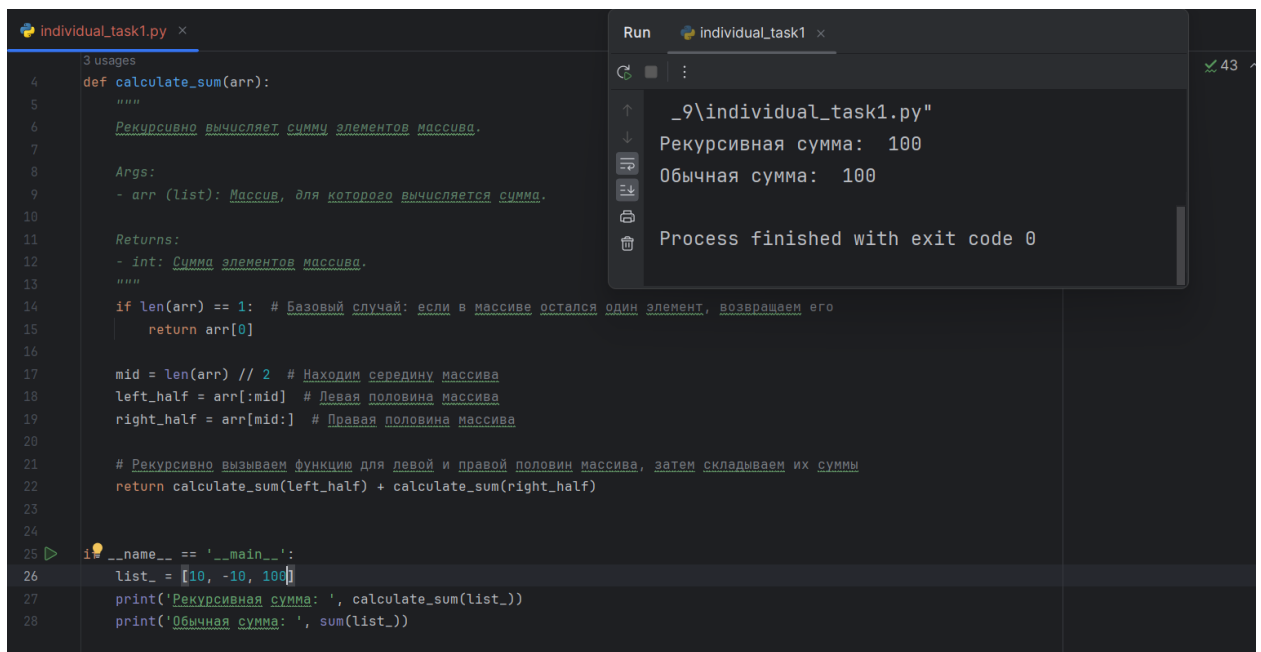
происходят, пока в получившихся кусках массива не окажется по одному элементу и вычисление суммы, соответственно, не станет тривиальным:



```
individual_task1.py x
3 usages
4 def calculate_sum(arr):
5     """
6     Рекурсивно вычисляет сумму элементов массива.
7
8     Args:
9     - arr (list): Массив, для которого вычисляется сумма.
10
11     Returns:
12     - int: Сумма элементов массива.
13     """
14     if len(arr) == 1: # Базовый случай: если в массиве остался один элемент, возвращаем его
15         return arr[0]
16
17     mid = len(arr) // 2 # Находим середину массива
18     left_half = arr[:mid] # Левая половина массива
19     right_half = arr[mid:] # Правая половина массива
20
21     # Рекурсивно вызываем функцию для левой и правой половин массива, затем складываем их суммы
22     return calculate_sum(left_half) + calculate_sum(right_half)
23
24
25 if __name__ == '__main__':
26     list_ = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
27     print('Рекурсивная сумма: ', calculate_sum(list_))
28     print('Обычная сумма: ', sum(list_))
```

```
Run individual_task1 x
43
\fundamentals_of_software_engineering_lab2
_9\individual_task1.py"
Рекурсивная сумма: 55
Обычная сумма: 55
Process finished with exit code 0
```

Рисунок 8 – Код и его выполнение (1)



```
individual_task1.py x
3 usages
4 def calculate_sum(arr):
5     """
6     Рекурсивно вычисляет сумму элементов массива.
7
8     Args:
9     - arr (list): Массив, для которого вычисляется сумма.
10
11     Returns:
12     - int: Сумма элементов массива.
13     """
14     if len(arr) == 1: # Базовый случай: если в массиве остался один элемент, возвращаем его
15         return arr[0]
16
17     mid = len(arr) // 2 # Находим середину массива
18     left_half = arr[:mid] # Левая половина массива
19     right_half = arr[mid:] # Правая половина массива
20
21     # Рекурсивно вызываем функцию для левой и правой половин массива, затем складываем их суммы
22     return calculate_sum(left_half) + calculate_sum(right_half)
23
24
25 if __name__ == '__main__':
26     list_ = [10, -10, 100]
27     print('Рекурсивная сумма: ', calculate_sum(list_))
28     print('Обычная сумма: ', sum(list_))
```

```
Run individual_task1 x
43
_9\individual_task1.py"
Рекурсивная сумма: 100
Обычная сумма: 100
Process finished with exit code 0
```

Рисунок 9 – Код и его выполнение (2)

5. Сделаем merge веток main/develop и отправим изменения на удаленный репозиторий:

```

b704a61 (HEAD -> develop) individual_task1.py is added
ae15f19 task2.py is added
c28a0d6 task1.py is added
e2dfac2 (origin/main, origin/HEAD, main) Update README.md
e8110ee Initial commit
PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9>

```

Рисунок 10 – Коммиты проекта

```

PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9> git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9> git merge develop
Updating e2dfac2..b704a61
Fast-forward
 .gitignore      | 2 +-
 individual_task1.py | 28 ++++++
 task1.py        | 91 ++++++
 task2.py        | 81 ++++++
 4 files changed, 201 insertions(+), 1 deletion(-)
 create mode 100644 individual_task1.py
 create mode 100644 task1.py
 create mode 100644 task2.py
PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9>

```

Рисунок 11 – Merge веток develop/main

```

PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9> git push origin main
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 12 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 2.68 KiB | 2.68 MiB/s, done.
Total 10 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
To https://github.com/PoTtaTto/fundamentals_of_software_engineering_lab2_9
 e2dfac2..b704a61 main -> main
PS C:\Study\СКФУ\Основы программной инженерии\Лабораторная работа 2.9\fundamentals_of_software_engineering_lab2_9>

```

Рисунок 12 – Отправка изменений на удаленный репозиторий

Ответы на контрольные вопросы:

1. Для чего нужна рекурсия?

Рекурсия используется для решения задач, разбивая их на более простые подзадачи того же типа и вызывая функцию из самой себя.

2. Что называется базой рекурсии?

База рекурсии – это условие выхода из рекурсии, обычно это проверка, при которой функция больше не вызывает саму себя.

3. Что такое стек программы и как он используется при вызове функций?

Стек программы – это область памяти, используемая для хранения данных и адресов возврата функций. При вызове функции происходит помещение информации о вызове на вершину стека, а при завершении функции - её удаление из стека.

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

Для получения текущего значения максимальной глубины рекурсии в Python можно воспользоваться функцией `sys.getrecursionlimit()`.

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

Если количество рекурсивных вызовов превысит максимальную глубину рекурсии в Python, возникнет исключение `RecursionError`.

6. Как изменить максимальную глубину рекурсии в языке Python?

Максимальную глубину рекурсии можно изменить с помощью функции `sys.setrecursionlimit()`.

7. Каково назначение декоратора `lru_cache`?

Декоратор `lru_cache` используется для кеширования результатов вызовов функции, сохраняя результаты предыдущих вызовов и возвращая сохраненное значение при повторных вызовах с теми же аргументами.

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

Хвостовая рекурсия – это форма рекурсии, когда рекурсивный вызов является последней операцией в функции. Оптимизация хвостовых вызовов включает в себя замену рекурсивных вызовов на итерацию, что позволяет уменьшить использование стека и памяти.