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:
Юрьев Илья Евгеньевич
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 язык программирования Python:

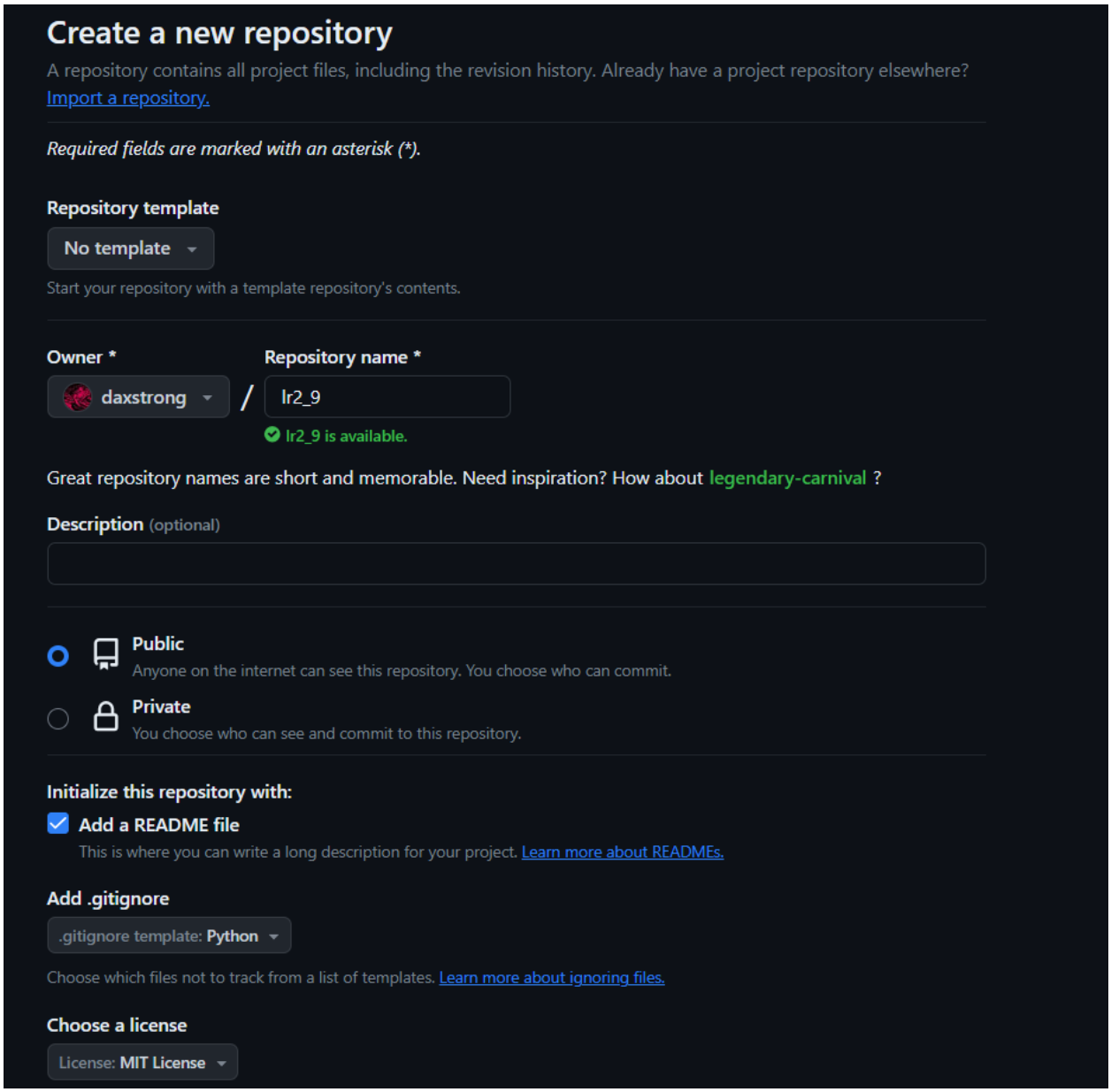


Рисунок 1 – Создание репозитория с заданными настройками

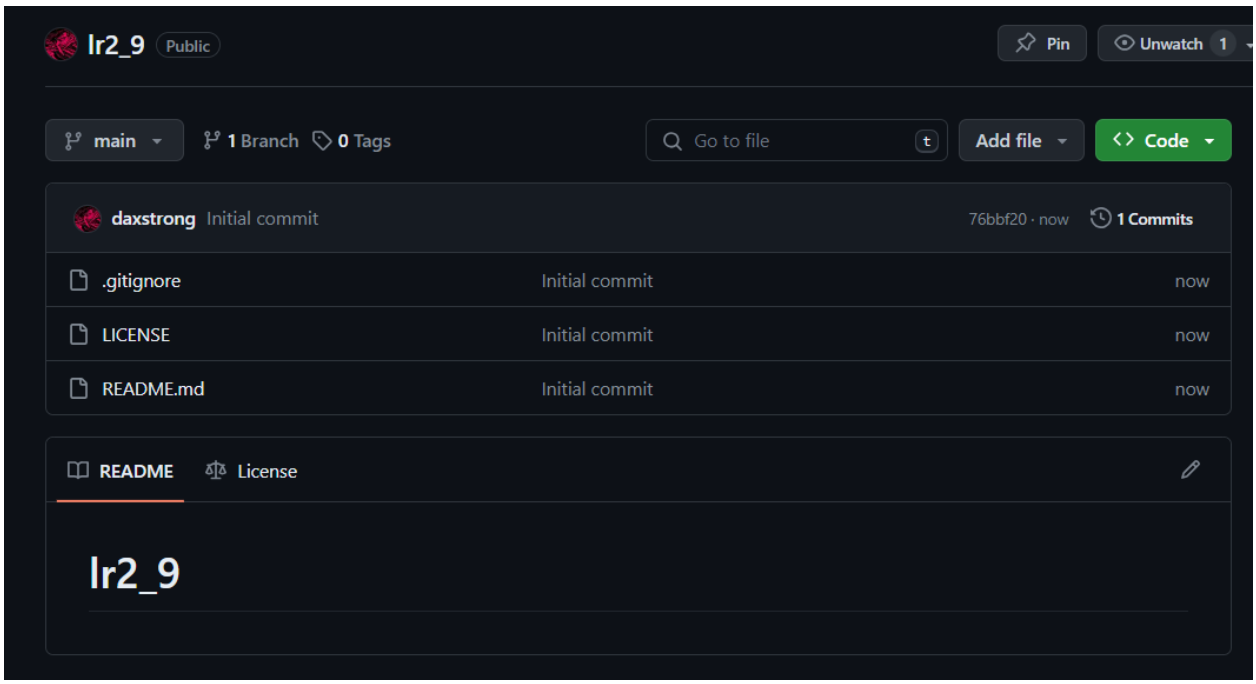


Рисунок 2 – Созданный репозиторий

```
ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии
$ git clone https://github.com/daxstrong/lr2_9.git
Cloning into 'lr2_9'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.
```

Рисунок 3 – Клонирование репозитория

```
ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии/lr2_9 (main)
$ git checkout -b develop
Switched to a new branch 'develop'
```

Рисунок 4 – Создание ветки develop

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 изменится скорость работы рекурсивных версий функций factorial и fib при использовании декоратора lru_cache? Приведите в отчет и обоснуйте полученные результаты:

Листинг программы:

```
import timeit
from functools import lru_cache


# 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)


# 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 <= 1:
        return n
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)


# Декоратор lru_cache для рекурсивных функций
@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)


@lru_cache(maxsize=None)
def fib_recursive_cached(n):
    if n <= 1:
        return n
    else:
        return fib_recursive_cached(n - 1) + fib_recursive_cached(n - 2)


# Оценка скорости работы итеративной и рекурсивной версий факториала
print("Factorial Iterative:", timeit.timeit("factorial_iterative(10)",
globals=globals(), number=100000))
print("Factorial Recursive:", timeit.timeit("factorial_recursive(10)",
globals=globals(), number=1000))
print("Factorial Recursive (Cached):",
timeit.timeit("factorial_recursive_cached(10)", globals=globals(),
number=1000))
```

```
# Оценка скорости работы итеративной и рекурсивной версий чисел Фибоначчи
print("Fibonacci Iterative:", timeit.timeit("fib_iterative(10)",
globals=globals(), number=100000))
print("Fibonacci Recursive:", timeit.timeit("fib_recursive(10)",
globals=globals(), number=1000))
print("Fibonacci Recursive (Cached):",
timeit.timeit("fib_recursive_cached(10)", globals=globals(), number=1000))
```

```
C:\Users\ilyay\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:/Users/ilyay/OneDrive
Factorial Iterative: 0.02976100001251325
Factorial Recursive: 0.0004618000239133835
Factorial Recursive (Cached): 4.0000013541430235e-05
Fibonacci Iterative: 0.030929300002753735
Fibonacci Recursive: 0.00613829999929294
Fibonacci Recursive (Cached): 3.6999990697950125e-05
```

Рисунок 5 – Вывод программы (Задание №1)

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.

Используя lru_cache для рекурсивных функций factorial и fibonacci, мы ожидаем увидеть улучшение в производительности, особенно при высоких значениях входных аргументов. Это происходит за счет кеширования результатов предыдущих вызовов и предотвращения повторных вычислений.

Однако, важно отметить, что эффект кеширования будет более заметен для функций с повторяющимися вызовами и большими значениями аргументов.

Проанализируем результаты:

1. Факториал:

- Итеративная версия: высокая эффективность, так как нет повторных вычислений.

- Рекурсивная версия: медленная из-за повторных вычислений.

- Рекурсивная версия с кешированием: заметное ускорение, так как результаты кешируются.

2. Числа Фибоначчи:

- Итеративная версия: эффективная итерация по последовательности.

- Рекурсивная версия: медленная из-за экспоненциального роста вызовов.

- Рекурсивная версия с кешированием: существенное ускорение, так как избегаются повторные вычисления.

В целом, использование lru_cache приводит к заметному улучшению производительности для рекурсивных функций, делая их более пригодными для использования с высокими значениями входных данных.

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:

```
import timeit


# Исключение для оптимизации хвостовой рекурсии
class TailRecurseException(BaseException):
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


# Декоратор для хвостовой рекурсии
def tail_recursive(func):
    def wrapper(*args, **kwargs):
        while True:
            try:
                return func(*args, **kwargs)
            except TailRecurseException as e:
                args = e.args
                kwargs = e.kwargs
                continue

    return wrapper
```

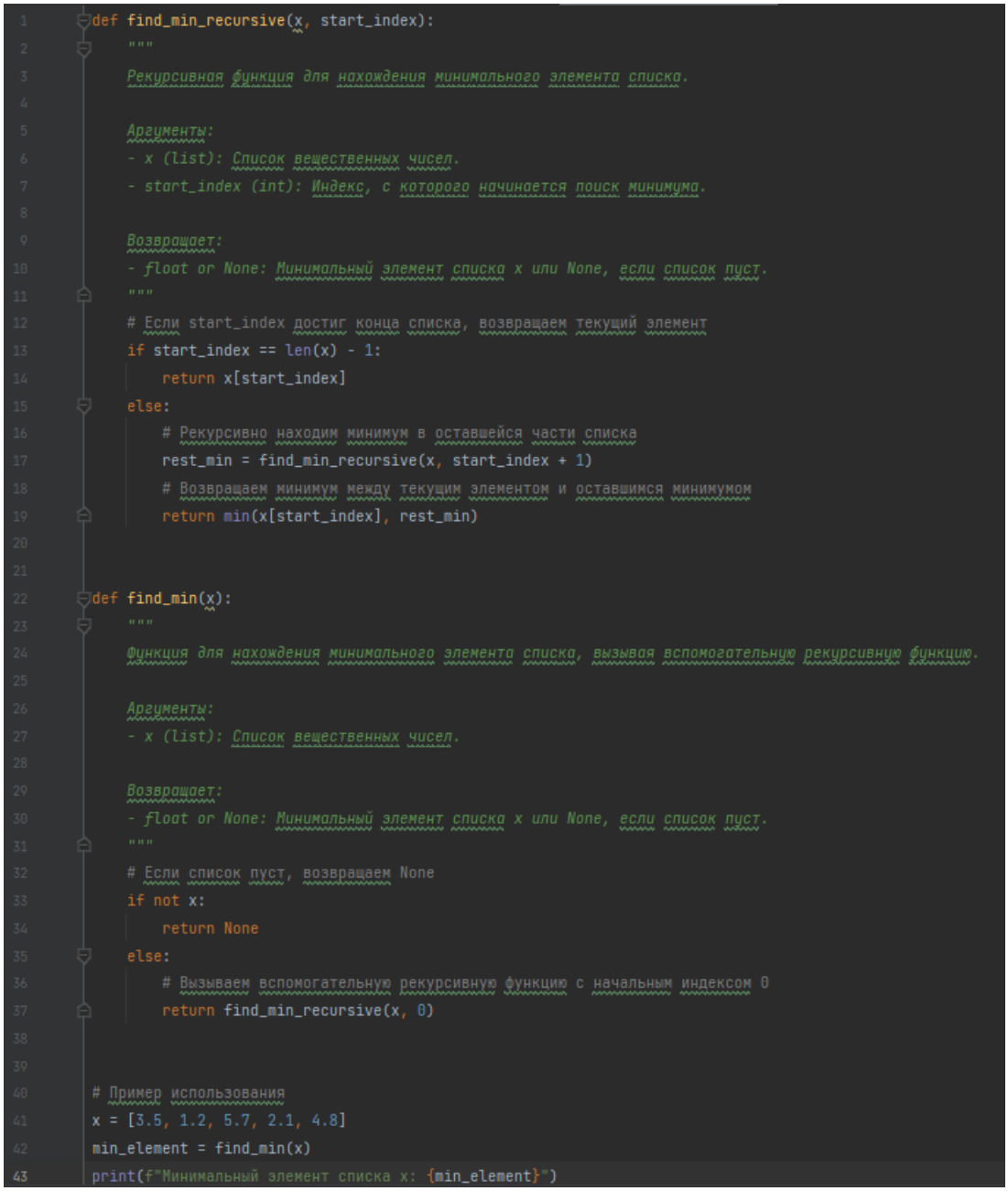
```python
# Рекурсивная функция для вычисления факториала с хвостовой рекурсией
@tail_recursive
def factorial(n, accumulator=1):
    """
    Рекурсивная функция для вычисления факториала.

    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), {})


# Рекурсивная функция для вычисления чисел Фибоначчи с хвостовой рекурсией
@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), {})


# Проверка, что скрипт запущен как основной
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:\Users\ilyay\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:/U
Время выполнения рекурсивной функции factorial: 0.08925630000885576
Время выполнения рекурсивной функции fib: 0.08434650002163835
```

Рисунок 6 – Вывод программы (Задание №2)

4. Выполним индивидуальное задание:

Дан список X из n вещественных чисел. Найти минимальный элемент списка,, используя вспомогательную рекурсивную функцию, находящую минимум среди последних элементов списка X, начиная с n-го.



```python
def find_min_recursive(x, start_index):
    """
    Рекурсивная функция для нахождения минимального элемента списка.

    Аргументы:
    - x (list): Список вещественных чисел.
    - start_index (int): Индекс, с которого начинается поиск минимума.

    Возвращает:
    - float or None: Минимальный элемент списка x или None, если список пуст.
    """
    # Если start_index достиг конца списка, возвращаем текущий элемент
    if start_index == len(x) - 1:
        return x[start_index]
    else:
        # Рекурсивно находим минимум в оставшейся части списка
        rest_min = find_min_recursive(x, start_index + 1)
        # Возвращаем минимум между текущим элементом и оставшимся минимумом
        return min(x[start_index], rest_min)


def find_min(x):
    """
    Функция для нахождения минимального элемента списка, вызывая вспомогательную рекурсивную функцию.

    Аргументы:
    - x (list): Список вещественных чисел.

    Возвращает:
    - float or None: Минимальный элемент списка x или None, если список пуст.
    """
    # Если список пуст, возвращаем None
    if not x:
        return None
    else:
        # Вызываем вспомогательную рекурсивную функцию с начальным индексом 0
        return find_min_recursive(x, 0)


# Пример использования
x = [3.5, 1.2, 5.7, 2.1, 4.8]
min_element = find_min(x)
print(f"Минимальный элемент списка x: {min_element}")
```

Рисунок 7 – Индивидуальное задание

```
C:\Users\ilyay\AppData\Local\Microsoft\WindowsApps\python3.11.exe
Минимальный элемент списка x: 1.2
```

Рисунок 8 – Вывод программы (Индивидуальное задание)

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

```
ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии/lr2_9 (develop)
$ git log --oneline
065aed4 (HEAD -> develop) Финальные изменения
76bbf20 (origin/main, origin/HEAD, main) Initial commit
```

Рисунок 9 – Коммиты ветки develop во время выполнения лабораторной работы

```
ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии/lr2_9 (develop)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии/lr2_9 (main)
$ git merge develop
Updating 76bbf20..065aed4
Fast-forward
 .idea/.gitignore                                |  8 +++
 .idea/inspectionProfiles/profiles_settings.xml |  6 +++
 .idea/lr2_9.iml                                |  8 +++
 .idea/misc.xml                                 |  4 ++
 .idea/modules.xml                              |  8 +++
 .idea/vcs.xml                                  |  6 +++
 individual.py                                  | 43 +++++++++++++++++
 task1.py                                       | 62 ++++++++++++++++++++++++
 task2.py                                       | 68 ++++++++++++++++++++++++++
 9 files changed, 213 insertions(+)
 create mode 100644 .idea/.gitignore
 create mode 100644 .idea/inspectionProfiles/profiles_settings.xml
 create mode 100644 .idea/lr2_9.iml
 create mode 100644 .idea/misc.xml
 create mode 100644 .idea/modules.xml
 create mode 100644 .idea/vcs.xml
 create mode 100644 individual.py
 create mode 100644 task1.py
 create mode 100644 task2.py
```

Рисунок 10 – Слияние веток main и develop

```
ilyay@DESKTOP-FF1JT6S MINGW64 ~/OneDrive/Рабочий стол/Основы программной инженер
ии/lr2_9 (main)
$ git push origin main
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 12 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 3.66 KiB | 3.66 MiB/s, done.
Total 13 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/daxstrong/lr2_9.git
   76bbf20..065aed4  main -> main
```

Рисунок 11 – Отправка изменений на удаленный репозиторий

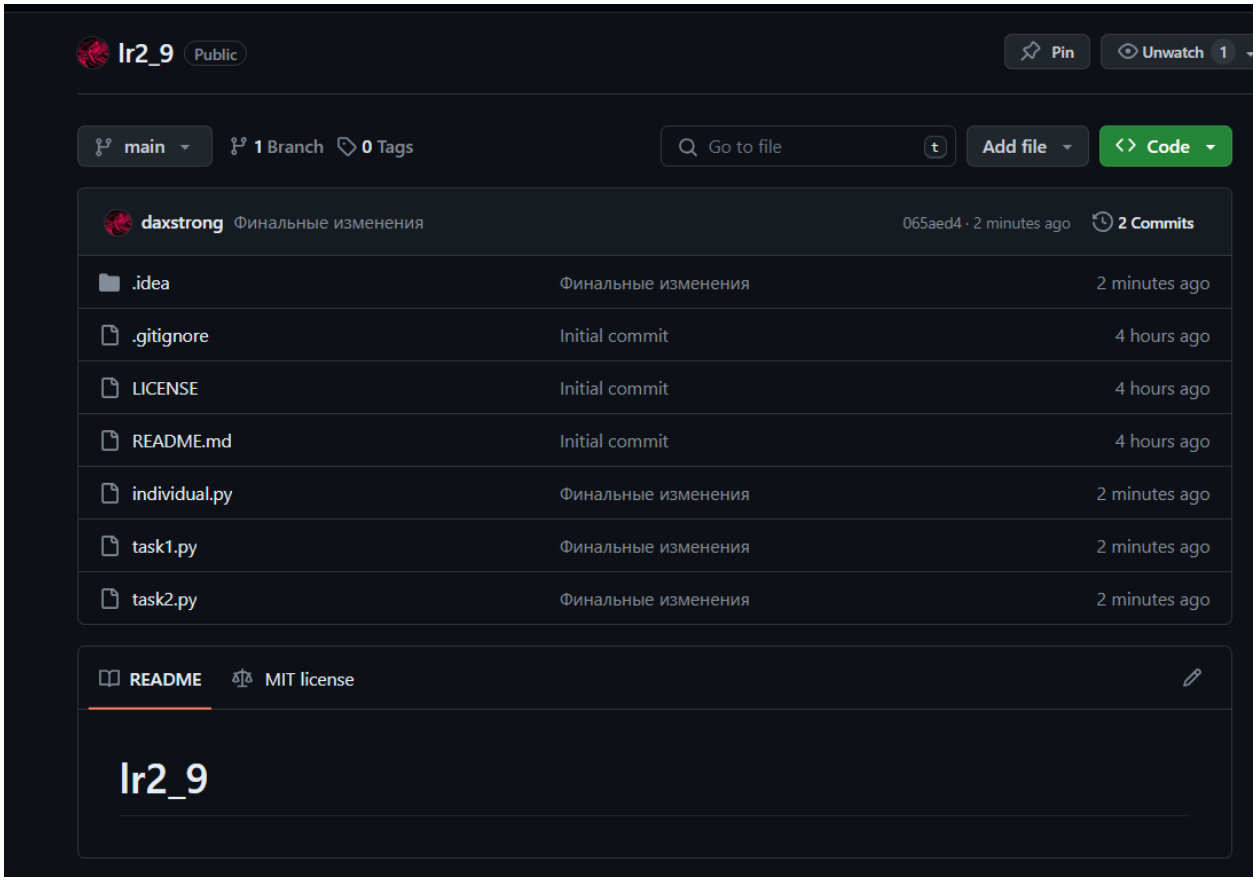


Рисунок 12 – Изменения удаленного репозитория

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.