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

**Кафедра**

**инфокоммуникаций**

**Институт цифрового**

**развития**

# ОТЧЁТ

**по лабораторной работе №2.9**

Дисциплина: «Основы программной инженерии» Тема: «Рекурсия в языке Python»

Выполнил:
студентка 2 курса
группы Пиж-б-о-21-1
Коныжев Максим
Викторович

Ставрополь 2022

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

1. Был создан репозиторий в Github в который были добавлены правила gitignore для работы IDE PyCharm, была выбрана лицензия MIT, сам репозиторий был клонирован на локальный сервер и был организован в соответствии с моделью ветвления git-flow.

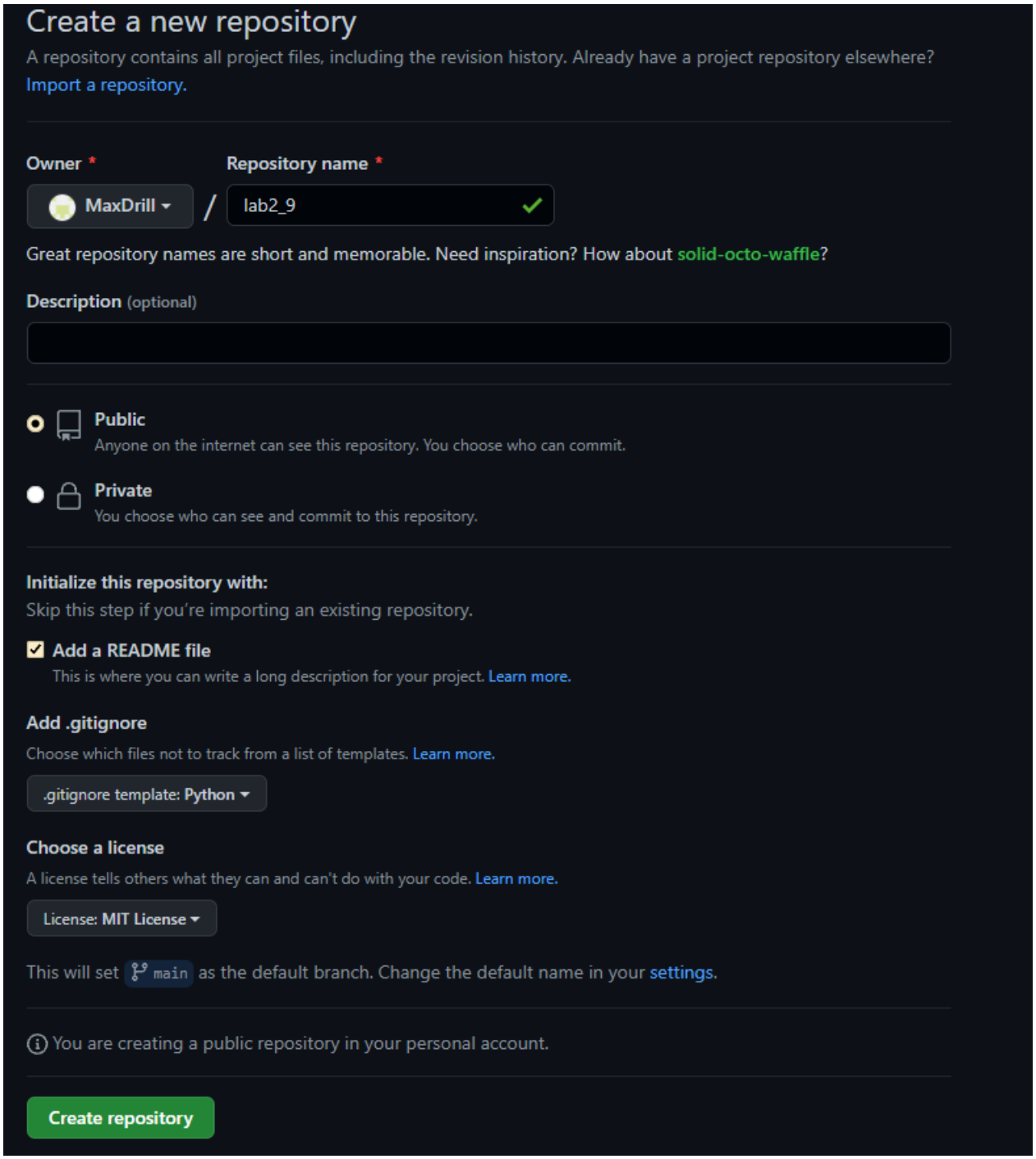


Рисунок 1 – Создание репозитория

```
C:\Users\UESR\gitproj>git clone https://github.com/MaxDrill/lab2_9.git
Cloning into 'lab2_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.

C:\Users\UESR\gitproj>cd lab2_9
```

Рисунок 2 – Клонирование репозитори

Задание 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


time_fact_rec = '''
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
'''


time_fib_rec = '''
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
'''


time_fact_itr = '''
def factorial(n):
    product = 1
    while n > 1:
```

```
        product *= n
        n -= 1
    return product
'''


time_fib_itr = '''
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
'''


time_fact_lru = '''
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)
'''


time_fib_lru = '''
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)
'''


if __name__ == '__main__':
    print('Результат рекурсивного факториала:', timeit.timeit(setup=time_fact_rec,
number=1000))
    print('Результат рекурсивного числа Фибоначи:', timeit.timeit(setup=time_fib_rec,
number=1000))
    print('Результат итеративного факториала:', timeit.timeit(setup=time_fact_itr,
number=1000))
    print('Результат итеративного числа Фибоначи:', timeit.timeit(setup=time_fib_itr,
number=1000))
    print('Результат факториала с декоратором:', timeit.timeit(setup=time_fact_lru,
number=1000))
    print('Результат числа Фибоначи с декоратором:', timeit.timeit(setup=time_fib_lru,
number=1000))
```

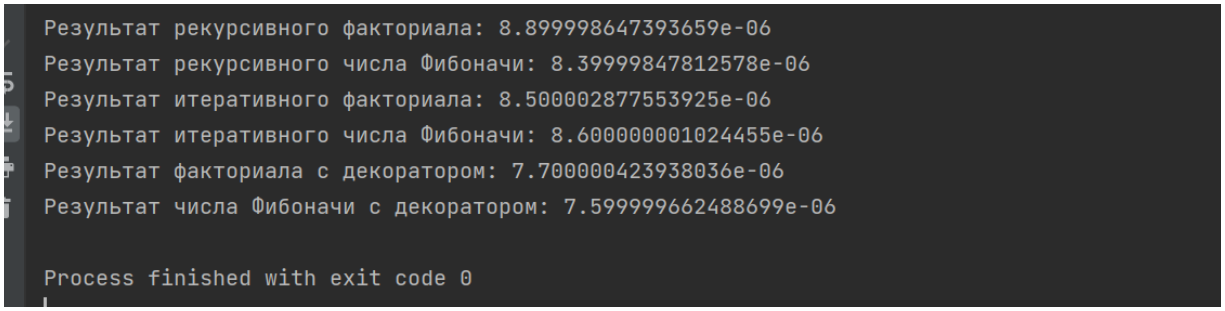


Рисунок 3 – Результат работы программы

Более быстрые вычисления происходят с использованием декоратора (это функция, которая принимает другую функцию в качестве аргумента, модифицирует или улучшает принятую функцию в последствии выдает измененную)

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.

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

import timeit

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


fib_without_intr = '''
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
'''


fact_with_intr = '''
class TailRecurseException:
    def __init__(self, args, kwargs):
```

```
        self.args = args
        self.kwargs = kwargs
def tail_call_optimized(g):
    def func(*args, **kwargs):
        f = sys._getframe()
        while f and f.f_code.co_filename == f:
            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
@tail_call_optimized
def factorial(n, acc=1):
    if n == 0:
        return acc
    return factorial(n-1, n*acc)
'''


fib_with_intr = '''
class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs
def tail_call_optimized(g):
    def func(*args, **kwargs):
        f = sys._getframe()
        while f and f.f_code.co_filename == f:
            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
@tail_call_optimized
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
'''


if __name__ == '__main__':
    print('Результат факториала:', timeit.timeit(setup=fact_without_intr, number=1000))
    print('Результат числа Фибоначи:', timeit.timeit(setup=fib_without_intr,
number=1000))
    print('Результат факториала с интроспекцией стека:',
timeit.timeit(setup=fact_with_intr, number=1000))
    print('Результат числа Фибоначи с интроспекцией стека:',
timeit.timeit(setup=fib_with_intr, number=1000))
```

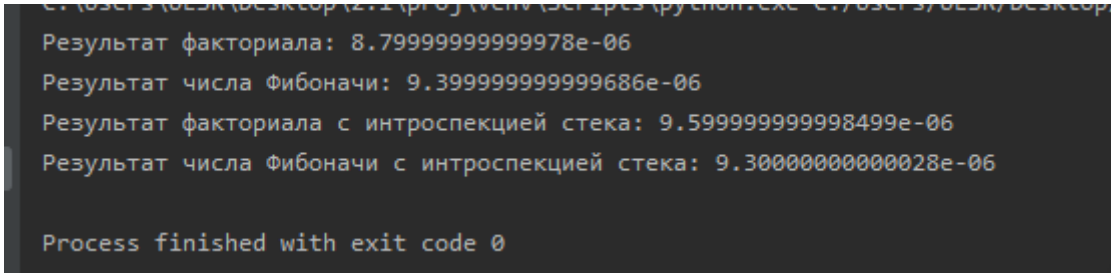


Рисунок 4 – Результат работы программы

Использование интроспекции стека сделала процесс вычисления более быстрым (за счет способность объекта во время выполнения получить информацию о его внутренней структуре.)

Индивидуальное задание

4. Создайте рекурсивную функцию, печатающую все возможные перестановки для целыхчисел от 1 до N.

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


def permutations(line):
    """creating permutations"""
    if len(line) == 1:
        return [line]  # базовая рекурсия
    else:
        # создаем массив для записи перестановок
        all = []
        # первый элемент списка помещаем в a
        a = line[0]
        # все перестановки, для последовательности без 1-го эл
        per = permutations(line[1:])
        # перестановки
        for p in per:
            # использовала enumerate, чтоб не брать range(len)
            for i, item in enumerate(p):
                # создаем комбинации с разным положением a (1-ый эл.)
                tmp = p[0:i] + [a] + p[i:]
                all.append(tmp)
            all.append(p + [a])
        return all


if __name__ == "__main__":
    n = int(input("Enter n: "))
    print(permutations([i for i in range(1, n + 1)]))
```

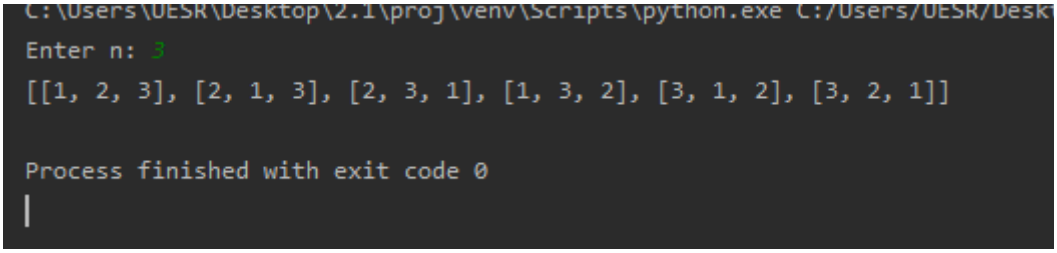


Рисунок 10 – Результат работы программы

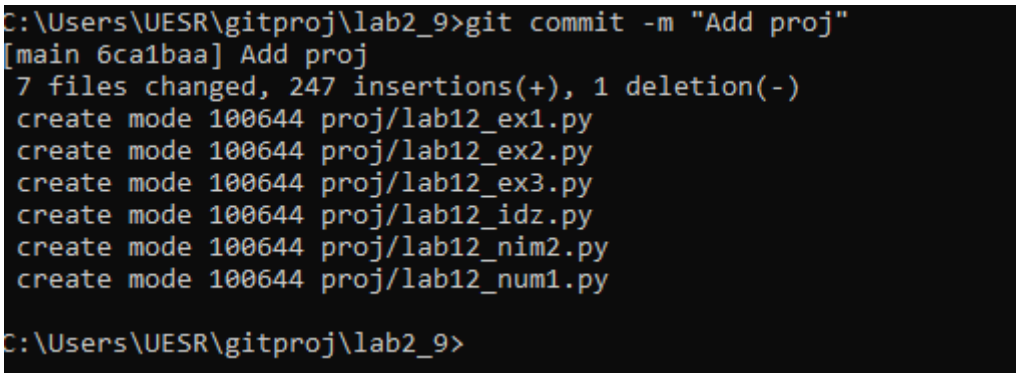


Рисунок 11 – Коммит и пуш изменений

Вывод: в результате выполнения лабораторной работы были приобретены навыки по работе с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.

Контрольные вопросы:

1. Для чего нужна рекурсия?

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия). Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

2. Что называется базой рекурсии?

База рекурсии – это такие аргументы функции, которые делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

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

Стек — это структура данных, в которой элементы хранятся в порядке поступления.

Стек хранит последовательность данных. Связаны данные так: каждый элемент указывает на тот, который нужно использовать следующим. Это линейная связь — данные идут друг за другом и нужно брать их по очереди. Из середины стека брать нельзя. Главный принцип работы стека — данные, которые попали в стек недавно, используются первыми. Чем раньше попал — тем позже используется. После использования элемент стека исчезает, и верхним становится следующий элемент.

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

Функция sys. getrecursionlimit() возвращает текущее значение предела рекурсии, максимальную глубину стека интерпретатора Python. Этот предел предотвращает бесконечную рекурсию от переполнения стека языка C и сбоя Python. Это значение может быть установлено с помощью sys.

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

Существует предел глубины возможной рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение RunTime.

6. Как изменить максимальную глубину рекурсии в языке Python? С помощью sys.setrecursionlimit(число).

7. Каково назначение декоратора lru_cache?

Функция lru_cache предназначается для мемоизации (предотвращения повторных вычислений), т. е. кэширует результат в памяти. Полезный инструмент, который уменьшает количество 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 путёмь преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах. В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии. Типовой механизм реализации вызова функции основан на сохранении адреса возврата, параметров и локальных переменных функции в стеке и выглядит следующим образом:

1. В точке вызова в стек помещаются параметры, передаваемые функции, и адрес возврата.

2. Вызываемая функция в ходе работы размещает в стеке собственные локальные переменные.

3. По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно — в один из регистров процессора).

4. Команда возврата из функции считывает из стека адрес возврата и выполняет переход поэтому адресу. Либо непосредственно перед, либо сразу после возврата из функции стек очищается от параметров