

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**Институт цифрового развития**

**Кафедра информационных систем и технологий**

Отчет по лабораторной работе №12.

Дисциплина: «Основы программной инженерии»

**Выполнил:**

Студент группы ПИЖ-б-о-22-1,

направление подготовки: 09.03.04

«Программная инженерия»

ФИО: Джараян Арег Александрович

**Проверил:**

Воронкин Р. А.

Ставрополь 2022

Тема: Лабораторная работа 2.9. Рекурсия в языке Python.

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

Выполнение работы:

1. Изучил теоретический материал работы.
2. Создал репозиторий на git.hub.

**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 \*** aregdz / **Repository name \*** 12lab  
✔ 12lab is available.

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

**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](#).

*i* You are creating a public repository in your personal account.

**Create repository**

Рисунок 1 – создание репозитория

3. Клонировал репозиторий.

```
aregd@DESKTOP-5KV9QA9 MINGW64 ~  
$ cd "C:\Users\aregd\OneDrive\Рабочий стол\git12"  
  
aregd@DESKTOP-5KV9QA9 MINGW64 ~/OneDrive/Рабочий стол/git12  
$ git clone https://github.com/aregdz/12lab.git  
Cloning into '12lab'...  
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.  
  
aregd@DESKTOP-5KV9QA9 MINGW64 ~/OneDrive/Рабочий стол/git12  
$ |
```

Рисунок 2 – клонирование репозитория 4.

4. Дополнить файл gitignore необходимыми правилами.

```
.gitignore – Блокнот  
Файл Правка Формат Вид Справка  
# Created by .ignore support plugin (hsz.mobi)  
### Python template  
# Byte-compiled / optimized / DLL files  
__pycache__/  
*.py[cod]  
*$py.class  
  
# C extensions  
*.so  
  
# Distribution / packaging  
.Python  
env/  
build/  
develop-eggs/  
dist/  
downloads/
```

Рисунок 3 – .gitignore для IDE PyCharm

5. Организовать свой репозиторий в соответствии с моделью ветвления git-flow.

```
aregd@DESKTOP-5KV9QA9 MINGW64 ~/OneDrive/Рабочий стол/git12  
$ cd 12lab  
  
aregd@DESKTOP-5KV9QA9 MINGW64 ~/OneDrive/Рабочий стол/git12/12lab (main)  
$ git checkout -b develop  
Switched to a new branch 'develop'  
  
aregd@DESKTOP-5KV9QA9 MINGW64 ~/OneDrive/Рабочий стол/git12/12lab (develop)  
$ |
```

Рисунок 4 – создание ветки develop

6. 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  from timeit import timeit
5  from functools import lru_cache
6
7  import sys
8
9  usage
10 @lru_cache
11 def factorial_recursion(n):
12     if n == 0:
13         return 1
14     elif n == 1:
15         return 1
16     else:
17         return n * factorial_recursion(n - 1)
18
19 2 usages
20 @lru_cache
21 def fib_recursion(n):
22     if n == 0 or n == 1:
23         return n
24     else:
25         return fib_recursion(n - 2) + fib_recursion(n - 1)
26
27 def factorial_iterable(n):
28     product = 1
29     while n > 1:
30         product *= n
31         n -= 1
32     return product
33
34 def fib_iterable(n):
35     a, b = 0, 1
36     while n > 0:
37         a, b = b, a + b
38         n -= 1
39     return a
40
41
42 if __name__ == "__main__":
43     sys.setrecursionlimit(5000)
44     n = 35
45     setup1 = """from __main__ import fib_recursion"""
46     setup2 = """from __main__ import factorial_recursion"""
47     timer = timeit(stmt=f'fib_recursion({n})', number=10, setup=setup1)
48     print('Время выполнения рекурсивной функции с @lru_cache: ', {timer})
49     timer = timeit(stmt=f'factorial_recursion({n})', number=10, setup=setup2)
50     print('Время выполнения рекурсивной функции с @lru_cache: ', {timer})
```

Рисунок 5 – пример 1

```
C:\Users\aregd\AppData\Local\Programs\Python\Python311\python.exe
Время выполнения рекурсивной функции: {24.40867439995054}
Время выполнения итеративной функции: {3.100000321865082e-05}
```

Рисунок 6 – пример выполнения 1(fib\_iterable)

```
C:\Users\aregd\AppData\Local\Programs\Python\Python311\python.exe "C:\Users\aregd\AppData\Local\Programs\Python\Python311\python.exe"
Время выполнения рекурсивной функции: {5.019991658627987e-05}
Время выполнения итеративной функции: {3.630004357546568e-05}
```

Рисунок 7 – пример выполнения (factorial\_iterable)

```
Время выполнения рекурсивной функции с @lru_cache: {3.5400036722421646e-05}
Время выполнения рекурсивной функции с @lru_cache: {3.870006185024977e-05}
```

Рисунок 8 – 3 пример выполнения

7. 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
5  import sys
6  from timeit import timeit
7
8
9  class TailRecurseException(Exception):
10     def __init__(self, args, kwargs):
11         self.args = args
12         self.kwargs = kwargs
13
14     2 usages
15
16     def tail_call_optimized(g):
17         """
18         Эта программа показывает работу декоратора, который производит оптимизацию
19         хвостового вызова. Он делает это, вызывая исключение, если оно является его
20         прародителем, и перехватывает исключение, чтобы подделать оптимизацию хвоста.
21         Эта функция не работает, если функция декоратора не использует хвостовой вызов.
22         """
23
24         def func(*args, **kwargs):
25             f = sys._getframe()
26             if (f.f_back and f.f_back.f_back and
```

```

25         if (f.f_back and f.f_back.f_back and
26             f.f_back.f_back.f_code == f.f_code):
27             raise TailRecurseException(args, kwargs)
28         else:
29             while True:
30                 try:
31                     return g(*args, **kwargs)
32                 except TailRecurseException as e:
33                     args = e.args
34                     kwargs = e.kwargs
35
36     func.__doc__ = g.__doc__
37     return func
38
39
40 1 usage
41 @tail_call_optimized
42 def factorial(n, acc=1):
43     """calculate a factorial"""
44     if n == 0:
45         return acc
46     return factorial(*args: n - 1, n * acc)
47
48 1 usage
49 @tail_call_optimized
50 def fib(i, current=0, nxt=1):
51     if i == 0:
52         return current
53     else:
54         return fib(*args: i - 1, nxt, current + nxt)
55
56 if __name__ == '__main__':
57     n = 30
58     setup1 = """from __main__ import factorial"""
59     setup2 = """from __main__ import fib"""
60     timer = timeit(stmt=f'factorial({n})', number=10,
61                     print(f"Время выполнения функции factorial(): {timer}")
62     timer = timeit(stmt=f'fib({n})', number=10, setup=
63                     print(f"Время выполнения функции fib(): {timer}")
64

```

Рисунок 9 – код программы

```

0: (base) C:\Users\Artem\Documents\Python\Python011>python
Время выполнения функции factorial(): 0.000993399997241795
Время выполнения функции fib(): 0.0006835999665781856

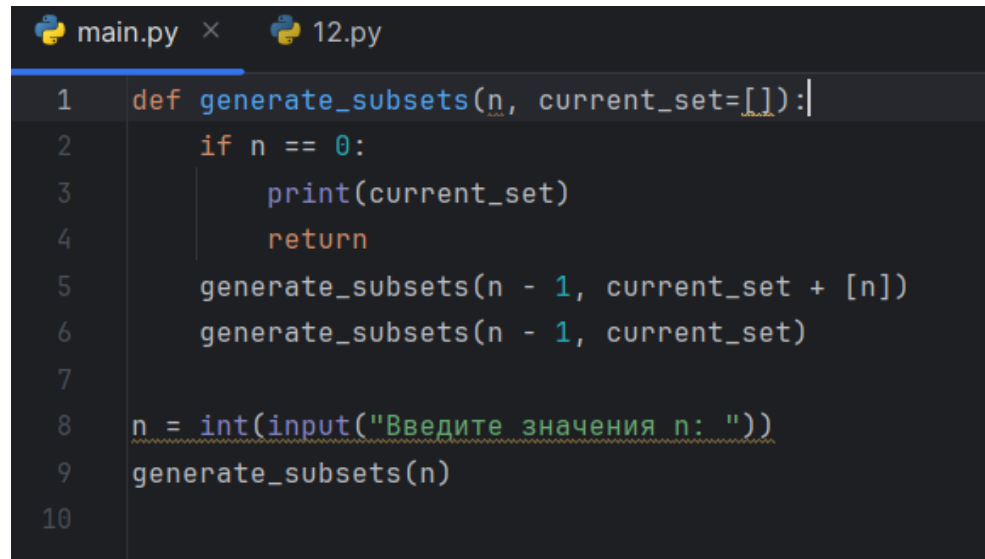
Process finished with exit code 0

```

Рисунок 10 – выполнение программы

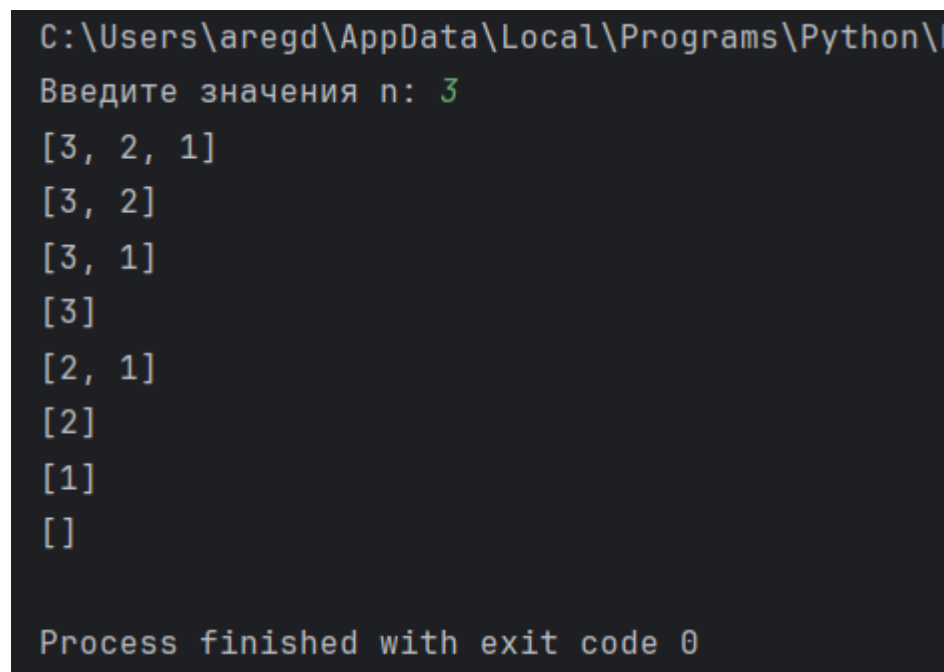
## Индивидуальное задание

8. Создайте рекурсивную функцию, печатающую все подмножества множества  $\{1, 2, \dots, N\}$



```
main.py x 12.py
1 def generate_subsets(n, current_set=[]):
2     if n == 0:
3         print(current_set)
4         return
5     generate_subsets(n - 1, current_set + [n])
6     generate_subsets(n - 1, current_set)
7
8 n = int(input("Введите значения n: "))
9 generate_subsets(n)
10
```

Рисунок 11 – выполнение индивидуального задания



```
C:\Users\aregd\AppData\Local\Programs\Python\
Введите значения n: 3
[3, 2, 1]
[3, 2]
[3, 1]
[3]
[2, 1]
[2]
[1]
[]

Process finished with exit code 0
```

Рисунок 12 – результат выполнения индивидуального задания

9. Зафиксировал все изменения в github в ветке develop.

```
aregd@DESKTOP-5KV9QA9 MINGW64 ~
$ cd "D:\Рабочий стол\git12\12lab"

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git12/12lab (develop)
$ git add .

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git12/12lab (develop)
$ git commit -m"1"
[develop 703b40c] 1
3 files changed, 122 insertions(+)
create mode 100644 PyCharm/7.py
create mode 100644 PyCharm/8.py
create mode 100644 PyCharm/individ.py
```

Рисунок 13 – фиксация изменений в ветку develop

10. Слил ветки.

```
aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git12/12lab (develop)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git12/12lab (main)
$ git merge develop
Updating 2ef9205..703b40c
Fast-forward
 PyCharm/7.py      | 48 +++++
 PyCharm/8.py      | 61 +++++
 PyCharm/individ.py | 13 +++++
3 files changed, 122 insertions(+)
create mode 100644 PyCharm/7.py
create mode 100644 PyCharm/8.py
create mode 100644 PyCharm/individ.py

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git12/12lab (main)
$ |
```

Рисунок 14 – сливание ветки develop в ветку main

Вывод: приобрел навыки по работе с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.

### Контрольные вопросы:

1. Для чего нужна рекурсия?



Рекурсия — это прием в программировании, когда функция вызывает сама себя. Этот подход может быть использован для решения задач, которые могут быть разбиты на более простые подзадачи. Рекурсия часто приводит к более чистому и понятному коду, особенно в случаях, когда задача имеет структуру, поддерживающую деление на подзадачи.

## **2. Что называется базой рекурсии?**

Утверждение, if  $n == 0$ : return 1

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

Стек программы (или стек вызовов) - это структура данных, используемая для управления вызовами функций в программе. Каждый раз, когда функция вызывается, информация о текущем состоянии функции (локальные переменные, адрес возврата и т. д.) сохраняется в стеке. Когда функция завершает выполнение, эта информация удаляется из стека, и управление возвращается к вызывающей функции. Основные операции со стеком - это "положить" (push) и "взять" (pop). Это относится к добавлению информации в стек при вызове функции и удалению ее при завершении функции. Вот как происходит использование стека программы при вызове функций:

- Когда функция вызывается, текущее состояние функции (локальные переменные, адрес возврата и т. д.) помещается в вершину стека.
- Текущая функция становится активной функцией, и управление передается в вызываемую функцию.
- Локальные переменные функции хранятся в том же фрейме стека, что и другая информация о состоянии функции.
- Эти переменные доступны только в пределах текущей функции.
- При завершении функции информация из вершины стека удаляется (pop), возвращая управление к вызывающей функции.
- Адрес возврата используется для определения, куда вернуть управление.
- Если функция вызывает саму себя (рекурсия), каждый вызов функции создает новый фрейм стека.
- Каждый уровень рекурсии имеет свои локальные переменные и адреса возврата.

Стек вызовов предоставляет эффективный механизм управления выполнением программы, обеспечивая правильный порядок вызова и возврата функций. Он также играет важную роль в обработке исключений и управлении памятью.

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

Чтобы проверить текущие параметры лимита, нужно запустить:  
`sys.getrecursionlimit()`

## **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 `RuntimeError` : `RuntimeError: Maximum Recursion Depth Exceeded`

## **6. Как изменить максимальную глубину рекурсии в языке Python?**

Можно изменить предел глубины рекурсии с помощью вызова:  
`sys.setrecursionlimit(limit)`

## **7. Каково назначение декоратора `lru_cache`?**

Декоратор `lru_cache` (Least Recently Used Cache) предназначен для кэширования результатов выполнения функций с использованием стратегии "Наименее недавно использованный" (LRU). Он сохраняет результаты выполнения функции для предотвращения повторных вычислений, когда те же самые входные значения встречаются повторно.

Когда функция вызывается с определенными аргументами, `lru_cache` сохраняет результат выполнения функции в кэше. Если функция вызывается с теми же аргументами позже, она возвращает сохраненный результат, минуя фактическое выполнение функции. Это может существенно ускорить выполнение функций, требующих вычислений, которые могут быть дорогими по времени.

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

Хвостовая рекурсия - это форма рекурсии, при которой рекурсивный вызов является последней операцией в функции. Такой вызов расположен в "хвосте" функции, то есть не сопровождается последующими операциями возврата или обработки результатов. Важным свойством хвостовой рекурсии является то, что она может быть оптимизирована компилятором или интерпретатором, сокращая использование стека вызовов и уменьшая вероятность переполнения стека.

Оптимизация хвостовых вызовов, называемая также "оптимизацией хвостовой рекурсии" или "хвостовой рекурсивной оптимизацией", заключается в том, что компилятор или интерпретатор понимают, что после хвостового рекурсивного вызова нет необходимости сохранять текущий контекст выполнения (значения переменных, адрес возврата и т. д.) на стеке вызовов.