

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ ЦИФРОВОГО РАЗВИТИЯ**

**Отчет о лабораторной работе №12 по дисциплине основы программной
инженерии**

Выполнила:

Емельянова Яна

Александровна, 2 курс,

группа ПИЖ-б-о-20-1,

Проверил:

Доцент кафедры инфокоммуникаций,

Воронкин Р.А.

Ставрополь, 2021 г.

1. Рекурсия в языке Python

Примеры из методических указаний

Сущность рекурсии

```
ex1.py x
1  ▶  #!/usr/bin/env python3
2      # -*- coding: utf-8 -*-
3
4
5  def recursion(n):
6      if n == 1:
7          return 1
8      return n + recursion(n - 1)
9
10
11  ▶  if __name__ == '__main__':
12      n = int(input("Enter n: "))
13      summary = 0
14      for i in range(1, n + 1):
15          summary += i
16      print(f"Iterative sum: {summary}")
17      print(f"Recursion sum: {recursion(n)}")
18
```

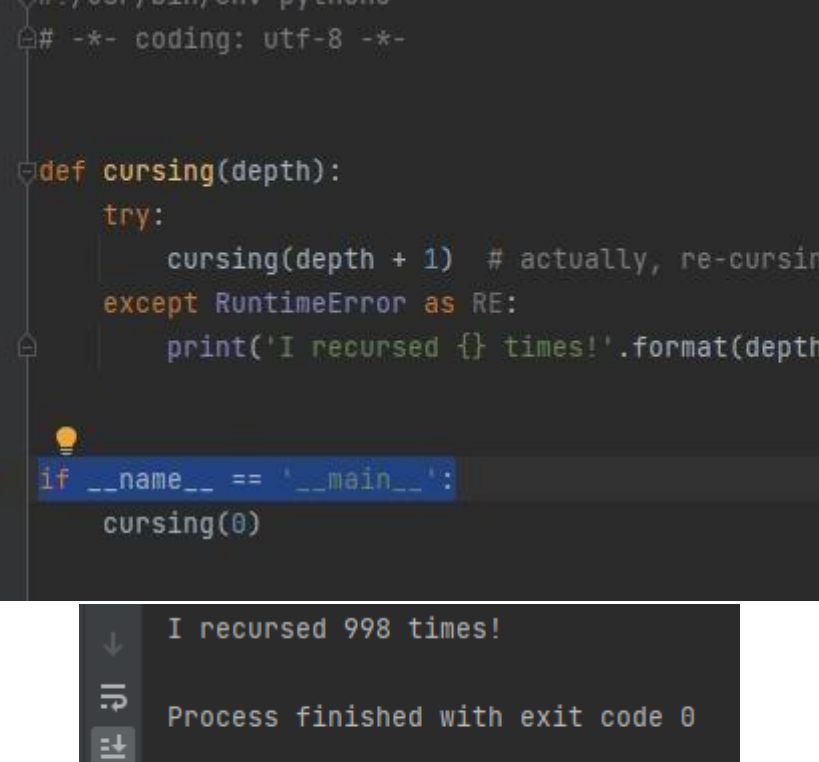
↓
Enter n: 12
Iterative sum: 78
Recursion sum: 78
Process finished with exit code 0

Как и когда происходит рекурсия

```
ex1.py x e2.py x
1 ▶ #!/usr/bin/env python3
2   # -*- coding: utf-8 -*-
3
4   from functools import lru_cache
5
6
7   @lru_cache
8   def fib(number):
9       if number == 0 or number == 1:
10           return number
11       else:
12           return fib(number - 2) + fib(number - 1)
13
14
15 ▶ if __name__ == '__main__':
16     n = int(input("Enter n: "))
17     print(f"Fibonacci's number of {n} is : {fib(n)}")
18
```

```
↓
Enter n: 12
Fibonacci's number of 12 is : 144
↻
Process finished with exit code 0
📄
🗑
```

Увеличение максимальной глубины рекурсии



```
e3.py
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  def cursing(depth):
6      try:
7          cursing(depth + 1) # actually, re-cursing
8      except RuntimeError as RE:
9          print('I recursed {} times!'.format(depth))
10
11
12  if __name__ == '__main__':
13      cursing(0)
14
```

I recursed 998 times!

Process finished with exit code 0

Хвостовая рекурсия

```
e4.py x
1 ▶ #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 def countdown(n):
5     if n == 0:
6         print("Blastoff!")
7     else:
8         print(n)
9         countdown(n-1)
10
11
12 ▶ if __name__ == '__main__':
13     ⚡ countdown(10)
14
```

```
↓ 10
↺ 9
↺ 8
↺ 7
↺ 6
↺ 5
↺ 4
↺ 3
↺ 2
↺ 1
Blastoff!

Process finished with exit code 0
```

```
e5.py x
1 ▶ #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 def find_max(seq, max_so_far):
5     if not seq:
6         return max_so_far
7     if max_so_far < seq[0]:
8         return find_max(seq[1:], seq[0])
9     else:
10        return find_max(seq[1:], max_so_far)
11
12
13 ▶ if __name__ == '__main__':
14     a = [i for i in range(10) if i % 2 != 0]
15     print(a)
16     print(find_max(a, a[0]))
17
```

```
↓ [1, 3, 5, 7, 9]
≡ 9
⇓ Process finished with exit code 0
🖨
🗑
```

Интроспекция стека

```

ex6.py x
1  ▶  #!/usr/bin/env python3
2      # -*- coding: utf-8 -*-
3      # Эта программа показывает работу декоратора, который производит оптимизацию
4      # хвостового вызова. Он делает это, вызывая исключение, если оно является его
5      # прародителем, и перехватывает исключения, чтобы вызвать стек.
6      import sys
7
8
9      class TailRecurseException(Exception):
10         def __init__(self, args, kwargs):
11             self.args = args
12             self.kwargs = kwargs
13
14
15     def tail_call_optimized(g):
16         def func(*args, **kwargs):
17             f = sys._getframe()
18             if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
19                 raise TailRecurseException(args, kwargs)
20             else:
21                 while True:
22                     try:
23                         return g(*args, **kwargs)
24                     except TailRecurseException as e:
25                         args = e.args
26                         kwargs = e.kwargs
27
28             func.__doc__ = g.__doc__
29             return func
30
31
32     @tail_call_optimized
33     def factorial(n, acc=1):
34         """calculate a factorial"""
35         if n == 0:
36             return acc
37         return factorial(n - 1, n * acc)
38
39
tail_call_optimized() > func()

```

```
37     return factorial(n - 1, n * acc)
38
39
40     @tail_call_optimized
41     def fib(i, current=0, next=1):
42         if i == 0:
43             return current
44         else:
45             return fib(i - 1, next, current + next)
46
47
48     if __name__ == '__main__':
49         print(factorial(10000))
50         print(fib(10000))
51
```

2846259680917054518906413212119868890148051401702799230794179
3364476487643178326662161200510754331030214846068006390656476

Process finished with exit code 0

1.1 Задача 1 (рис 1-4)


```
pr1.py x
1  ▶  #!/usr/bin/env python3
2      # -*- coding: utf-8 -*-
3
4      from functools import lru_cache
5      import timeit
6
7
8      @lru_cache
9      def factorial_rec(n, acc=1):
10         if n == 0:
11             return acc
12         return factorial_rec(n - 1, n * acc)
13
14
15     @lru_cache
16     def fib_rec(i, current=0, next=1):
17         if i == 0:
18             return current
19         else:
20             return fib_rec(i - 1, next, current + next)
21
22
23     def factorial_iter(n):
24         if n == 0 or n == 1:
25             return 1
26         fact = 1
27         for i in range(1, n + 1):
28             fact *= i
29         return fact
30
31
32     def fib_iter(n):
33         a = 0
34         b = 1
35         for i in range(n):
36             c = a + b
37             a = b
38             b = c
39         return a
```

Рисунок 1 – Код программы

```

38     b = c
39     return a
40
41
42 ▶ if __name__ == '__main__':
43     number = int(input("Enter the number to calculate: "))
44     start_time = timeit.default_timer()
45     factorial_rec(number)
46     print("Recursive factorial time is: ",
47           timeit.default_timer() - start_time
48         )
49
50     start_time = timeit.default_timer()
51     factorial_iter(number)
52     print("Iterative factorial time is :",
53           timeit.default_timer() - start_time
54         )
55
56     start_time = timeit.default_timer()
57     fib_rec(number)
58     print("Recursive Fibonacci time is :",
59           timeit.default_timer() - start_time
60         )
61
62     start_time = timeit.default_timer()
63     fib_iter(number)
64     print("Iterative Fibonacci time is :",
65           timeit.default_timer() - start_time
66         )
67

```

Рисунок 2 – Код программы, продолжение

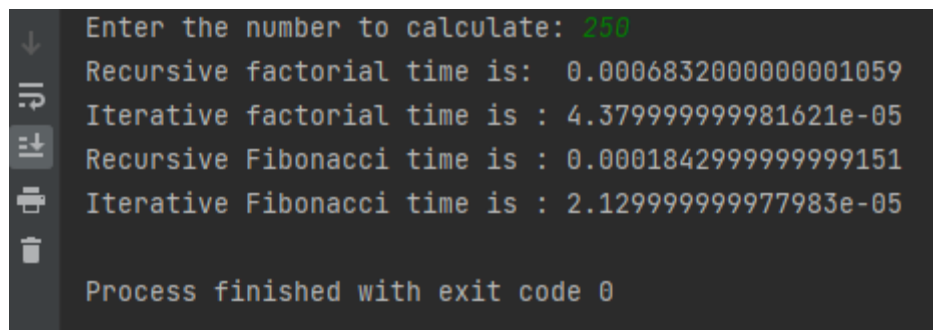
```

↓
Enter the number to calculate: 250
Recursive factorial time is: 0.00057780000000002947
Iterative factorial time is : 4.229999999960654e-05
Recursive Fibonacci time is : 8.639999999981995e-05
Iterative Fibonacci time is : 2.1699999999569286e-05

Process finished with exit code 0

```

Рисунок 3 – Результаты без lru_cache

A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for back, forward, search, and other navigation functions. The main area of the terminal displays the following text:

```
Enter the number to calculate: 250
Recursive factorial time is: 0.0006832000000001059
Iterative factorial time is : 4.379999999981621e-05
Recursive Fibonacci time is : 0.0001842999999999151
Iterative Fibonacci time is : 2.129999999977983e-05

Process finished with exit code 0
```

Рисунок 4 – Результаты с использованием lru_cache

Такие результаты связаны с тем, что декоратор кэша LRU проверяет наличие некоторых базовых случаев, а затем обортывает пользовательскую функцию с помощью оболочки `_lru_cache_wrapper`. Внутри оболочки происходит логика добавления элемента в кэш, логика LRU, т. е. Добавление нового элемента в круговую очередь, удаление элемента из круговой очереди. Значение в кэше хранится в виде списка из четырех элементов. Первый элемент-это ссылка на предыдущий элемент, второй элемент-ссылка на следующий элемент, третий элемент-ключ для конкретного вызова функции, четвертый элемент-результат. Вот фактическое значение для аргумента функции Фибоначчи 1 `[[[...], [...], 1, 1], [...], [...], 1, 1], None, None]`. [...] означает ссылку на себя(список). Когда элемент уже находится в кэше, нет необходимости проверять, заполнена ли циклическая очередь, или извлекать элемент из кэша. Скорее измените положение элементов в круговой очереди. Поскольку недавно использованный элемент всегда находится в верхней части, код перемещается в последнее значение в верхнюю часть очереди, и предыдущий верхний элемент становится следующим за текущим элементом `last[NEXT] = root[PREV] = link` и `link[PREV] = last` и `link[NEXT] = root`.

1.2 Задача 2 (рис 5-9)

```
pr2.py x
1  ▶  #!/usr/bin/env python3
2      # -*- coding: utf-8 -*-
3
4      import timeit
5      import sys
6
7
8      class TailRecurseException(Exception):
9          def __init__(self, args, kwargs):
10             self.args = args
11             self.kwargs = kwargs
12
13
14      def tail_call_optimized(g):
15          def func(*args, **kwargs):
16              f = sys._getframe()
17              if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
18                  raise TailRecurseException(args, kwargs)
19              else:
20                  while True:
21                      try:
22                          return g(*args, **kwargs)
23                      except TailRecurseException as e:
24                          args = e.args
25                          kwargs = e.kwargs
26
27              func.__doc__ = g.__doc__
28              return func
29
30
31      # @tail_call_optimized
32      def factorial_rec(n, acc=1):
33          if n == 0:
34              return acc
35          return factorial_rec(n - 1, n * acc)
36
37
38      # @tail_call_optimized
39      def fib_rec(i, current=0, next=1):
```

Рисунок 5 – Код программы

```

38 # @tail_call_optimized
39 def fib_rec(i, current=0, next=1):
40     if i == 0:
41         return current
42     else:
43         return fib_rec(i - 1, next, current + next)
44
45
46 def factorial_iter(n):
47     if n == 0 or n == 1:
48         return 1
49     fact = 1
50     for i in range(1, n + 1):
51         fact *= i
52     return fact
53
54
55 def fib_iter(n):
56     a = 0
57     b = 1
58     for i in range(n):
59         c = a + b
60         a = b
61         b = c
62     return a
63
64
65 if __name__ == '__main__':
66     number = int(input("Enter the number: "))
67
68     start_time = timeit.default_timer()
69     factorial_rec(number)
70     print("Recursive factorial time is: "
71         f"{timeit.default_timer() - start_time} microseconds"
72     )
73
74     start_time = timeit.default_timer()
75     factorial_iter(number)

```

Рисунок 6 – Код программы, продолжение

```

73
74     start_time = timeit.default_timer()
75     factorial_iter(number)
76     print("Iterative factorial time is :",
77           f"{timeit.default_timer() - start_time} microseconds"
78         )
79
80     start_time = timeit.default_timer()
81     fib_rec(number)
82     print("Recursive Fibonacci time is :",
83           f"{timeit.default_timer() - start_time} microseconds"
84         )
85
86     start_time = timeit.default_timer()
87     fib_iter(number)
88     print("Iterative Fibonacci time is :",
89           f"{timeit.default_timer() - start_time} microseconds"
90         )
91

```

Рисунок 7 – Код программы, продолжение

```

Enter the number: 250
Recursive factorial time is: 0.00046789999999985454
Iterative factorial time is : 4.959999999987197e-05
Recursive Fibonacci time is : 0.000105099999999985808
Iterative Fibonacci time is : 2.1400000000006026e-05

Process finished with exit code 0

```

Рисунок 8 – Результаты без хвостовой оптимизации

```

Enter the number: 250
Recursive factorial time is: 0.0009824999999996642
Iterative factorial time is : 7.789999999996411e-05
Recursive Fibonacci time is : 0.00068890000000001033
Iterative Fibonacci time is : 1.8999999999991246e-05

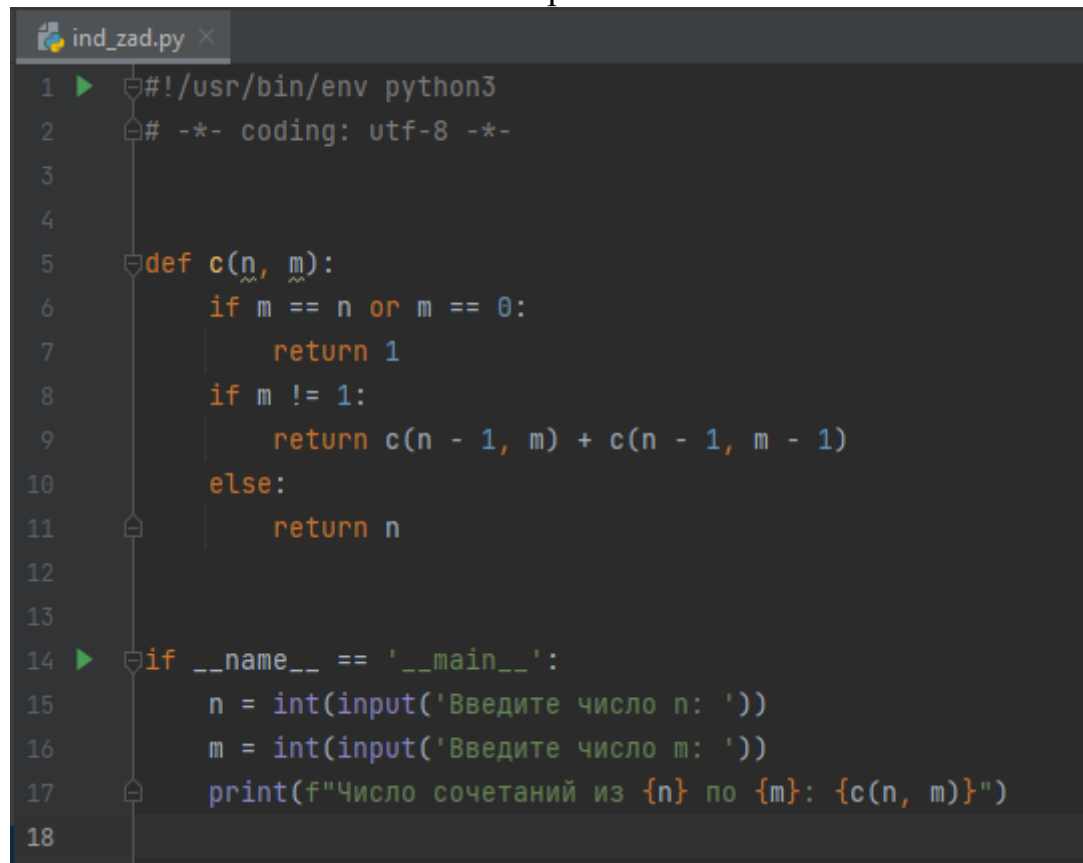
Process finished with exit code 0

```

Рисунок 9 – Результаты с применением хвостовой оптимизации

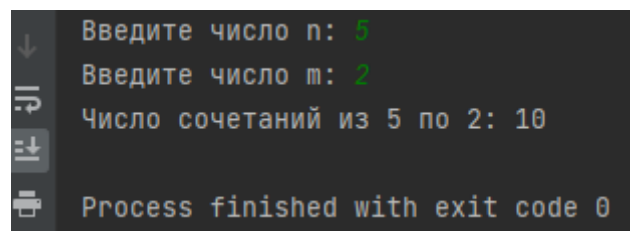
1.3 Индивидуальное задание (рис 10, 11)

Вариант 9



```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4
5 def c(n, m):
6     if m == n or m == 0:
7         return 1
8     if m != 1:
9         return c(n - 1, m) + c(n - 1, m - 1)
10    else:
11        return n
12
13
14 if __name__ == '__main__':
15     n = int(input('Введите число n: '))
16     m = int(input('Введите число m: '))
17     print(f"Число сочетаний из {n} по {m}: {c(n, m)}")
18
```

Рисунок 10 – Код программы



```
Введите число n: 5
Введите число m: 2
Число сочетаний из 5 по 2: 10
Process finished with exit code 0
```

Рисунок 11 – Вывод программы

2. Ответы на контрольные вопросы

1. Для чего нужна рекурсия?

Рекурсия может работать в обратную сторону, иногда рекурсивное решение проще, чем итеративное решение. Это очевидно при реализации обращения связанного списка.

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 функции – в теории вычислительных систем, LIFO-стек, хранящий информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания (в том числе при переключении задач в многозадачной среде).

Когда функция производит вложенный вызов, происходит следующее:

- 1) Выполнение текущей функции приостанавливается.
- 2) Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.
- 3) Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
- 4) После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

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

Выполнить команду `sys.getrecursionlimit()`

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

Когда предел достигнут, возникает исключение `RuntimeError` :
`RuntimeError: Maximum Recursion Depth Exceeded`

6. Как изменить максимальную глубину рекурсии в языке Python?

Нужно выполнить команду: `sys.setrecursionlimit(limit)`

7. Каково назначение декоратора `lru_cache` ?

Декоратор `@lru_cache()` модуля `functools` оборачивает функцию с переданными в нее аргументами и запоминает возвращаемый результат соответствующий этим аргументам. Такое поведение может сэкономить время и ресурсы, когда дорогая или связанная с вводом/выводом функция периодически вызывается с одинаковыми аргументами.

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. Оптимизация хвостового вызова (ТСО) — это способ автоматического сокращения рекурсии в рекурсивных функциях. Для оптимизации, к примеру, можно постараться каждый раз при вызове функции сохранять только кадр стека внутренней вызываемой функции, что может значительно сэкономить память.