

Отчет по лабораторной работе 3

Рекурсия

Дата: 2025-10-10 **Семестр:** 3 курс 5 семестр **Группа:** ПИЖ-6-о-23-2(1) **Дисциплина:** Анализ сложности алгоритмов **Студент:** Черников Дмитрий Дмитриевич

Цель работы

Освоить принцип рекурсии, научиться анализировать рекурсивные алгоритмы и понимать механизм работы стека вызовов. Изучить типичные задачи, решаемые рекурсивно, и освоить технику мемоизации для оптимизации рекурсивных алгоритмов. Получить практические навыки реализации и отладки рекурсивных функций.

Практическая часть

Выполненные задачи

- ☐ Задача 1: Реализовать классические рекурсивные алгоритмы.
- ☐ Задача 2: Проанализировать их временную сложность и глубину рекурсии.
- ☐ Задача 3: Реализовать оптимизацию рекурсивных алгоритмов с помощью мемоизации.
- ☐ Задача 4: Сравнить производительность наивной рекурсии и рекурсии с мемоизацией.
- ☐ Задача 5: Решить практические задачи с применением рекурсии.

Ключевые фрагменты кода

```
# recursion.py
def factorial(n):
    """
    Вычисляет факториал положительного целого числа n рекурсивно.
    Аргументы:
        n: положительное целое число, для которого вычисляется факториал.
    Возвращает:
        n!, если n >= 1; -1 для некорректных (неположительных) аргументов.
    """
    if n == 1:
        return 1
    elif n > 1:
        return n * factorial(n-1)
    else:
        return -1

# Временная сложность: O(n)
# Глубина рекурсии: n

def fibbonachi(n):
```

```

"""
Вычисляет n-ое число Фибоначчи рекурсивно.
Аргументы:
    n: целое число (индекс) – позиция числа Фибоначчи в последовательности.
Возвращает:
    n-ое число Фибоначчи
    -1 для некорректных (неположительных) аргументов.
"""
if n > 0 and n < 3:
    return 1
elif n > 2:
    return fibonacci(n-1) + fibonacci(n-2)
else:
    return -1

# Временная сложность:  $O(2^n)$ 
# Глубина рекурсии: n

def quick_power(val, p):
    """
    Быстрое возведение в степень (возведение val в целую степень p) рекурсивно.
    Аргументы:
        val: число – основание степени.
        p: целое число – показатель степени.
    Возвращает:
        val**p как результат возведения в степень.
    """
    if p < 0:
        return quick_power(1/val, -p)
    if p == 0:
        return 1
    if p % 2 == 0:
        return quick_power(val*val, p/2)
    else:
        return val*quick_power(val*val, (p-1)/2)

# Временная сложность:  $O(\log p)$ 
# Глубина рекурсии:  $\log_2 |p|$  (по основанию 2)

```

```

# memoization.py
import timeit
import matplotlib.pyplot as plt

# Глобальная переменная для подсчёта числа рекурсивных вызовов
fib_memo_calls = 0
fib_calls = 0

def naive_fibonacci(n):

```

```

"""
Вычисляет n-ое число Фибоначчи рекурсивно.
Аргументы:
    n: целое число (индекс) – позиция числа Фибоначчи в последовательности.
Возвращает:
    n-ое число Фибоначчи
    -1 для некорректных (неположительных) аргументов.
"""

```

```

global fib_calls
fib_calls += 1
if n > 0 and n < 3:
    return 1
elif n > 2:
    return naive_fibonacci(n-1) + naive_fibonacci(n-2)
else:
    return -1

```

```

def fibonacci_memoized(n, memo={}):

```

```

"""
Вычисляет n-ое число Фибоначчи с мемоизацией.
Аргументы:
    n: целое число – позиция числа Фибоначчи в последовательности.
    мемо: словарь для хранения уже вычисленных значений Фибоначчи.
Возвращает:
    n-ое число Фибоначчи
"""

```

```

global fib_memo_calls
fib_memo_calls += 1
if n in memo:
    return memo[n]
if n > 0 and n < 3:
    return 1
elif n > 2:
    memo[n] = (fibonacci_memoized(n-1, memo) +
               fibonacci_memoized(n-2, memo))
    return memo[n]
else:
    return -1

```

```

def compare_fibonacci(n):

```

```

"""
Сравнивает результаты вычисления n-го числа Фибоначчи с мемоизацией и без.
Аргументы:
    n: целое число – позиция числа Фибоначчи в последовательности.
Возвращает:
    Словарь содержащий два ключа time, calls значениями
    которых являются кортежи из двух значений:
    (результат без мемоизации, результат с мемоизацией).
"""

```

```

global fib_memo_calls
global fib_calls

```

```

start1 = timeit.default_timer()
naive_fibonacci(n)
end1 = timeit.default_timer()
fib_time = (end1 - start1) * 1000

fib_count = fib_calls
fib_calls = 0

start2 = timeit.default_timer()
fibonacci_memoized(n)
end2 = timeit.default_timer()
fib_memo_time = (end2 - start2) * 1000

fib_memo_count = fib_memo_calls
fib_memo_calls = 0

result = {"time": (fib_time, fib_memo_time),
          "calls": (fib_count, fib_memo_count)}

return result

```

```

def Visualization(sizes):
    """
    Визуализация результатов замеров времени вычисления числа Фибоначчи
    с мемоизацией и без.
    """
    naive_times = []
    memoized_times = []
    print(sizes)
    for size in sizes:
        result = compare_fibonacci(size)
        naive_times.append(result["time"][0])
        memoized_times.append(result["time"][1])

    plt.plot(sizes, naive_times, marker="o", color="red", label="Naive")
    plt.plot(sizes, memoized_times, marker="o", color="blue", label="Memoized")
    plt.xlabel("n (индекс числа Фибоначчи)")
    plt.ylabel("Время выполнения (ms)")
    plt.title("Сравнение времени вычисления "
              "числа Фибоначчи с мемоизацией и без")
    plt.legend(loc="upper left", title="Метод")
    plt.savefig("ОТЧЁТ/fibonacci_comparison.png", dpi=300, bbox_inches='tight')
    plt.show()

# Характеристики вычислительной машины
pc_info = """
Характеристики ПК для тестирования:
- Процессор: Intel Core i5-12500H @ 2.50GHz
- Оперативная память: 32 GB DDR4
- ОС: Windows 11
- Python: 3.12
"""
print(pc_info)

```

```
print(f"{naive_times} - naive \n {memoized_times} - memoized")
```

```
# recursion_tasks.py

import os

def binary_search_recursive(arr, target, left, right):
    """
    Рекурсивно ищет элемент target в отсортированном списке arr.
    Аргументы:
        arr: отсортированный список элементов.
        target: искомый элемент.
        left: левая граница поиска.
        right: правая граница поиска.
    Возвращает:
        Индекс target в arr, если найден, иначе -1.
    """
    if left > right:
        return -1
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search_recursive(arr, target, left, mid - 1)
    else:
        return binary_search_recursive(arr, target, mid + 1, right)

def print_directory_tree(path, indent=""):
    """
    Рекурсивно выводит дерево каталогов и файлов, начиная с заданного пути.
    Аргументы:
        path: строка – путь к директории, с которой начинается обход.
        indent: строка – отступ для текущего уровня (служебный параметр).
    """
    print(f"{indent}{os.path.basename(path)}/")
    try:
        for entry in os.listdir(path):
            full_path = os.path.join(path, entry)
            if os.path.isdir(full_path):
                print_directory_tree(full_path, indent + "    ")
            else:
                print(f"{indent}    {entry}")
    except PermissionError:
        print(f"{indent}    [Permission Denied]")

def hanoi(n, source, target, auxiliary):
    """
```

Рекурсивно решает задачу Ханойские башни для n дисков и 3 стержней.

Аргументы:

n: количество дисков.

source: имя исходного стержня (строка).

target: имя целевого стержня (строка).

auxiliary: имя вспомогательного стержня (строка).

"""

if n == 1:

print(f"Переместить диск 1 с {source} на {target}")

return

hanoi(n-1, source, auxiliary, target)

print(f"Переместить диск {n} с {source} на {target}")

hanoi(n-1, auxiliary, target, source)

main.py

```
from modules.recursion import factorial, fibonacci, quick_power
from modules.memoization import compare_fibonacci, Visualization
from modules.recursion_tasks import binary_search_recursive, print_directory_tree,
hanoi
```

Демонстрация работы функций из модуля recursion

print(factorial(1), factorial(5), factorial(7), factorial(9), factorial(-1))

print(fibonacci(1), fibonacci(5), fibonacci(

7), fibonacci(15), fibonacci(-1))

print(quick_power(5, 7), quick_power(2, 21), quick_power(2, 65))

Демонстрация работы функций из модуля memoization

print(compare_fibonacci(35))

Демонстрация работы функций из модуля recursion_tasks

arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

print(binary_search_recursive(arr, 7, 0, len(arr)-1))

print_directory_tree("./src")

hanoi(3, 'A', 'C', 'B')

Демонстрация работы функции визуализации из модуля memoization

Visualization([5, 10, 15, 20, 25, 30, 35])

1 120 5040 362880 -1

1 5 13 610 -1

78125 2097152 36893488147419103232

{'time': (1755.4174000106286, 0.020399995264597237), 'calls': (18454929, 67)}

3

src/

main.py

modules/

```
memoization.py
recursion.py
recursion_tasks.py
__pycache__/
    memoization.cpython-313.pyc
    recursion.cpython-313.pyc
    recursion_tasks.cpython-313.pyc
```

Переместить диск 1 с A на C
Переместить диск 2 с A на B
Переместить диск 1 с C на B
Переместить диск 3 с A на C
Переместить диск 1 с B на A
Переместить диск 2 с B на C
Переместить диск 1 с A на C
[5, 10, 15, 20, 25, 30, 35]

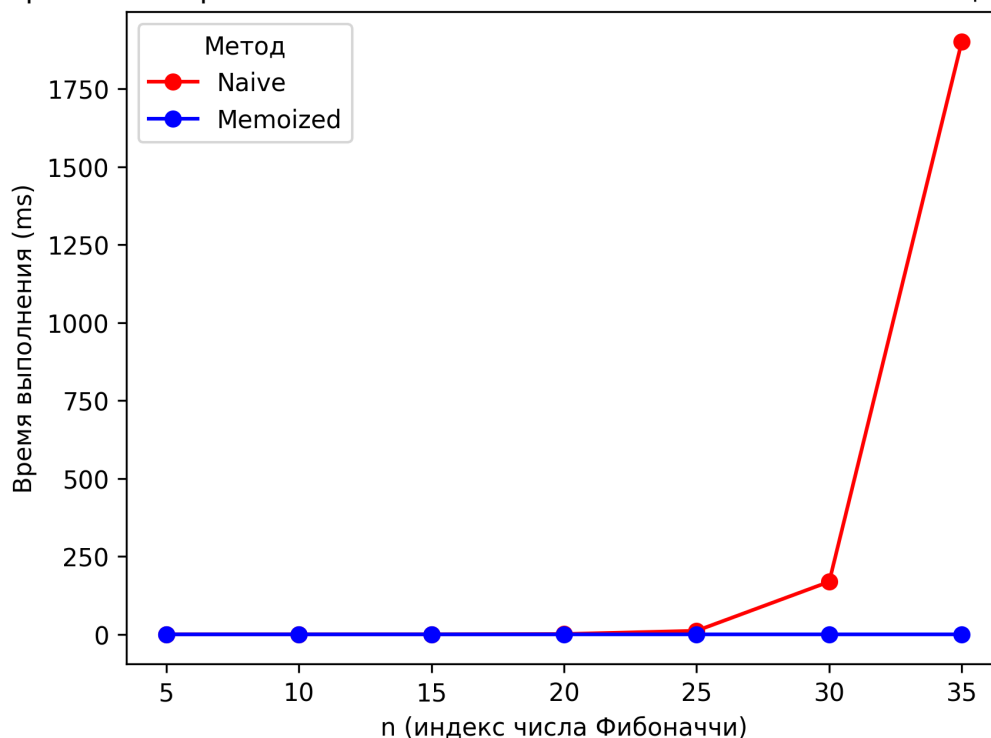
Характеристики ПК для тестирования:

- Процессор: Intel Core i5-12500H @ 2.50GHz
- Оперативная память: 32 GB DDR4
- ОС: Windows 11
- Python: 3.12

[0.0021999876480549574, 0.008300004992634058, 0.09539999882690609,
1.0861999908229336, 11.520400003064424, 168.91429999668617, 1902.3602999950526] -
naive

[0.0009999930625781417, 0.000200001522898674, 0.000300002284348011,
0.000200001522898674, 0.0018999999156221747, 0.0016999983927235007,
0.0017999991541728377] - memoized

Сравнение времени вычисления числа Фибоначчи с мемоизацией и без



Ответы на контрольные вопросы

Ответы на вопросы по теме "Рекурсия"

1. Базовый случай и рекурсивный шаг. Почему отсутствие базового случая приводит к ошибке

- **Базовый случай (условие выхода)** — Обязательное условие, которое прекращает рекурсивные вызовы и предотвращает за цикливание.
- **Рекурсивный шаг** — Шаг, на котором задача разбивается на более простую подзадачу того же типа и производится рекурсивный вызов.

Если **базового случая нет**, функция будет вызывать саму себя бесконечно, что приведёт к **переполнению стека вызовов (RecursionError)** — программа не сможет завершить вычисления.

2. Как работает мемоизация и как она влияет на вычисление чисел Фибоначчи

Мемоизация — Техника оптимизации, позволяющая избежать повторных вычислений результатов функций для одних и тех же входных данных путем сохранения ранее вычисленных результатов в кеше (например, в словаре).

Пример влияния на сложность:

- **Наивная рекурсия** для чисел Фибоначчи: $O(2^n)$ — из-за повторных пересчётов одних и тех же значений.
 - **С мемоизацией**: $O(n)$ — каждое значение вычисляется один раз и сохраняется.
-

3. Проблема глубокой рекурсии и её связь со стеком вызовов

Каждый рекурсивный вызов занимает место в **стеке вызовов**, где хранятся локальные переменные и адрес возврата.

При слишком большой глубине рекурсии стек переполняется, и программа завершает работу с ошибкой **RecursionError**.

Это особенно актуально для языков с ограниченным размером стека (например, Python по умолчанию ограничивает глубину до ~ 1000 вызовов).

4. Алгоритм решения задачи о Ханойских башнях для 3 дисков

Задача: нужно переместить 3 диска с **стержня А** на **стержень С**, используя **стержень В** как вспомогательный.

Алгоритм:

1. Переместить 2 верхних диска с А → В (используя С как вспомогательный).
2. Переместить нижний (третий) диск с А → С.
3. Переместить 2 диска с В → С (используя А как вспомогательный).

Последовательность шагов:

- 1. A → C
- 2. A → B
- 3. C → B
- 4. A → C
- 5. B → A
- 6. B → C
- 7. A → C

Всего шагов: $7 = 2^3 - 1$.
Общая сложность: $O(2^n)$.

5. Рекурсивные и итеративные алгоритмы: преимущества и недостатки

| Подход | Преимущества | Недостатки |
|-------------|---|---|
| Рекурсивный | Простой и наглядный код, легко описывает задачи, основанные на самоподобии (деревья, графы, Ханойские башни). | Использует стек вызовов, может вызвать переполнение при большой глубине; иногда медленнее из-за накладных расходов на вызовы функций. |
| Итеративный | Эффективен по памяти, не зависит от глубины рекурсии, быстрее при больших объёмах данных. | Код может быть сложнее и менее интуитивен для задач с рекурсивной природой. |