Отчет по лабораторной работе 3

Рекурсия

Дата: 2025-10-10
Семестр: 3 курс 5 семестр
Группа: ПИЖ-б-о-23-2(1)
Дисциплина: Анализ сложности алгоритмов
Студент: Торубаров Максим Евгеньевич

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

```python
# recursion.py
def factorial(n):
    # Функция вычисляет факториал числа n
    if n == 1:
        return 1  # O(1)
    else:
        return n * factorial(n - 1)  # O(n)
    # Общая сложность: O(n)
    # Глубина: O(n)


def fibonacci(n):
    # Функция вычисляет n-е число Фибоначчи
    if n == 0:  # O(1)
        return 0  # O(1)
    elif n == 1:  # O(1)
        return 1  # O(1)
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # O(2^n)
    # Общая сложность: O(2^n)
    # Глубина: O(2^n)


def fast_power(a, n):
    # Возведение a в n через степени двойки
    if n == 0:
        return 1  # O(1)
    if n == 1:
        return a  # O(1)

    half = fast_power(a, n // 2)  # O(log n)

    if n % 2 == 0:  # O(1)
        return half * half  # O(1)
    else:
        return half * half * a  # O(1)
    # Общая сложность: O(log n)
    # Глубина: O(log n)
```

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


def count_calls(func):
    # Декоратор для подсчета вызовов функции
    def wrapper(*args, **kwargs):
        wrapper.call_count += 1
        return func(*args, **kwargs)

    wrapper.call_count = 0
    return wrapper
```

```python
@count_calls
def fibonacci_memo(n, memo={}):
    # Функция вычисления n числа Фибоначи с мемоизацией
    if n in memo:  # O(1)
        return memo[n]
    if n == 1:  # O(1)
        return 1
    if n == 0:
        return 0  #

    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)  # O(1)
    return memo[n]  # O(1)
    # Общая сложность: O(n)


@count_calls
def fibonacci(n):
    # Функция вычисляет n-е число Фибоначчи
    if n == 0:  # O(1)
        return 0  # O(1)
    elif n == 1:  # O(1)
        return 1  # O(1)
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # O(2^n)
    # Общая сложность: O(2^n)


# Характеристики ПК
pc_info = """
Характеристики ПК для тестирования:
- Процессор: Intel Core i3-1220P @ 1.5GHz
- Оперативная память: 8 GB DDR4
- ОС: Windows 11
- Python: 3.12.10
"""
print(pc_info)

# Замер времени выполнения и глубины рекурсии при n = 35
print("Замер выремени выполнения и глубины рекурсии при n = 35:")
print("{:>10} {:>25} {:>25}".format(
    "Тип функции", "Время", "Глубина рекурсии"))

time_non_memo = timeit.timeit(lambda: fibonacci(35), number=1)
count_non_memo = fibonacci.call_count
time_memo = timeit.timeit(lambda: fibonacci_memo(35), number=1)
count_memo = fibonacci_memo.call_count

print("{:>10} {:>25.4f} {:>25}".format(
    "Без мемоизации", time_non_memo, count_non_memo))
print("{:>10} {:>25.4f} {:>25}".format("С мемоизацией", time_memo, count_memo))
```

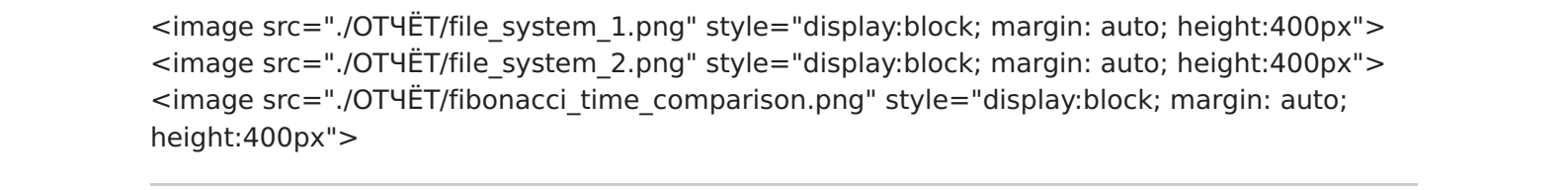
```
# Сравнение времени выполнения при разных n

# Диапазон значений n
n_values = list(range(1, 30))

# Время выполнения наивной и мемоизированной функции
print("""Замер выремени выполнения наивной
       и мемоизированной функций при разных N:""")
print("{:>10} {:>30} {:>30}".format(
    "N", "Время (мкс) - naive", "Время (мкс) - memo"))
times_non_memo = []
times_memo = []

for n in n_values:
    time_non_memo = timeit.timeit(lambda: fibonacci(n), number=10) * 1000 / 10
    times_non_memo.append(time_non_memo)
    time_memo = timeit.timeit(lambda: fibonacci_memo(n), number=10) * 1000 / 10
    times_memo.append(time_memo)
    print("{:>10} {:>30.4f} {:>30.4f}".format(n, time_non_memo, time_memo))

# Построение графика
plt.figure(figsize=(10, 6))
plt.plot(n_values, times_non_memo, label="Без мемоизации", marker="o")
plt.plot(n_values, times_memo, label="С мемоизацией", marker="x")
plt.xlabel("N (порядковый номер числа Фибоначчи)")
plt.ylabel("Время выполнения (мкс)")
plt.title("Сравнение времени выполнения: Фибоначчи с мемоизацией и без")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("fibonacci_time_comparison.png", dpi=300, bbox_inches="tight")
plt.show()
```

```
# recursion_tasks.py

import os
import sys

sys.setrecursionlimit(10000)  # Установка максимального размера стека рекурсии


# Задание 1
def task_1():
    def binary_search_recursive(arr, target, left=0, right=None):
        # Рекурсивный алгоритм бинарного поиска.

        if right is None:
            right = len(arr) - 1  # O(1)

        if left > right:
```

```
            return -1  # Элемент не найден O(1)

        mid = (left + right) // 2  # Средний индекс O(1)

        if arr[mid] == target:
            return mid  # Элемент найден O(1)
        elif arr[mid] < target:
            return binary_search_recursive(
                arr, target, mid + 1, right
            )  # Искать справа O(log n)
        else:
            return binary_search_recursive(
                arr, target, left, mid - 1
            )  # Искать слева O(log n)

    sorted_list = [1, 3, 5, 7, 9, 11, 13]
    print(binary_search_recursive(sorted_list, 7))
    print(binary_search_recursive(sorted_list, 6))


# Задание 2
def task_2():
    def tree(path, prefix="", is_last=True, depth=0):
        # Рекурсивный обход файловой системы.
        global max_depth
        if not os.path.exists(path):
            print(f"Путь '{path}' не существует.")  # O(1)
            return

        items = os.listdir(path)  # O(n)
        for i, item in enumerate(items):  # O(n)
            full_path = os.path.join(path, item)  # O(1)

            # Символы для визуального оформления дерева
            connector = "└── " if is_last and i == len(items) - 1 else "├── "
            new_prefix = prefix + ("    " if is_last else "│   ")  # O(1)

            print(prefix + connector + item)  # O(1)

            if os.path.isdir(full_path):
                if depth > max_depth:
                    max_depth = depth
                tree(full_path, new_prefix, i == len(items) - 1, depth + 1)
        # Общая сложность O(n)

    # Пример использования
    tree("D://University/3 курс")
    print(f"\nМаксимальная глубина рекурсии: {max_depth}")


# Задание 3
def task_3():
```

```
    def hanoi(n, source, target, auxiliary):
        # Рекурсивное решение задачи Ханойские башни.
        if n == 1:  # O(1)
            print(f"Переместить диск 1 с {source} на {target}")
            return

        # Шаг 1: Переместить n-1 дисков из source на auxiliary,
        # используя target как вспомогательный
        hanoi(n - 1, source, auxiliary, target)  # O(n - 1)

        # Шаг 2: Переместить n-й диск из source на target
        print(f"Переместить диск {n} с {source} на {target}")

        # Шаг 3: Переместить n-1 дисков из auxiliary на target,
        # используя source как вспомогательный
        hanoi(n - 1, auxiliary, target, source)  # O(n - 1)
        # Общая сложность O(2^n)

    hanoi(4, "A", "C", "B")


# Инициализация глобальной переменной для обхода файловой системыЦ
max_depth = 0
task_2()
```

```
# main.py

from modules.recursion import factorial, fibbonachi, quick_power
from modules.memoization import compare_fibbonachi, 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(fibbonachi(1), fibbonachi(5), fibbonachi(
    7), fibbonachi(15), fibbonachi(-1))
print(quick_power(5, 7), quick_power(2, 21), quick_power(2, 65))

# Демонстрация работы функций из модуля memoization
print(compare_fibbonachi(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])
```



Ответы на контрольные вопросы

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

Рекурсивный алгоритм состоит из двух обязательных компонентов: базового случая и рекурсивного шага. **Базовый случай** определяет условие терминации рекурсии, при котором функция возвращает конкретное значение без последующих самовызовов. **Рекурсивный шаг** декомпозирует исходную задачу на одну или несколько подзадач того же типа, но меньшей размерности, осуществляя вызов самой себя с изменёнными аргументами.

Отсутствие корректно определённого базового случая нарушает условие сходимости рекурсивного процесса. Это приводит к бесконечной последовательности рекурсивных вызовов, что вызывает исчерпание выделенной памяти под **стек вызовов** и последующее аварийное завершение программы с ошибкой **переполнения стека** (Stack Overflow или RecursionError).

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

Мемоизация — это методика оптимизации, основанная на кешировании результатов выполнения функций для предотвращения повторных вычислений при одинаковых входных данных. Реализуется, как правило, с использованием ассоциативной структуры данных (например, словаря), где ключом являются аргументы функции, а значением — вычисленный результат.

Применительно к вычислению чисел Фибоначчи, наивный рекурсивный алгоритм имеет экспоненциальную временную сложность **$O(2^n)$**, поскольку многократно пересчитывает значения для одних и тех же аргументов (например, `F(3)` вычисляется многократно при расчёте `F(5)`). Мемоизация устраняет избыточные вычисления, сохраняя каждое вычисленное значение `F(n)`. После этого любое обращение к `F(n)` выполняется за константное время **O(1)**, что снижает общую сложность алгоритма до **O(n)**.

3. Проблема глубокой рекурсии и её связь со стеком вызовов

Проблема глубокой рекурсии возникает из-за ограниченности размера стека вызовов — области памяти, используемой для хранения контекста выполнения (локальных переменных, адресов возврата) для каждого активного вызова функции. Каждый рекурсивный вызов помещает новый фрейм в стек. При превышении лимита глубины рекурсии, определяемого как аппаратными ресурсами, так и настройками runtime-окружения (например, ~1000 в Python), происходит **переполнение стека**, приводящее к исключению `RecursionError`.

4. Алгоритм решения задачи о Ханойских башнях для 3 дисков

Алгоритм перемещения n дисков со стержня A (источник) на стержень C (приёмник) с использованием B (вспомогательный) формулируется рекурсивно. Для $n = 3$ процесс выглядит следующим образом:

1. Рекурсивно переместить 2 верхних диска с A на B (используя C).
2. Переместить оставшийся самый большой диск с A на C.
3. Рекурсивно переместить 2 диска с B на C (используя A).

Последовательность операций для 3 дисков:

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

   Количество ходов составляет $2^3 - 1 = 7$, что соответствует общей формуле и временной сложности алгоритма **$O(2^n)$**.

5. Рекурсивные и итеративные алгоритмы: преимущества и недостатки

Критерий	Рекурсивный алгоритм	Итеративный алгоритм
Преимущества	Лаконичность и элегантность кода для задач с рекурсивной природой (обход деревьев, задачи типа "разделяй и властвуй"). Прямое отражение математической рекуррентной формулы.	Эффективность по памяти, так как не используется стек вызовов. Как правило, более высокая производительность из-за отсутствия накладных расходов на вызов функций.
Недостатки	Риск переполнения стека при большой глубине рекурсии. Потенциально более низкая производительность.	Код может быть менее читаемым и интуитивно понятным для задач, изначально рекурренных по своей сути. Требует явного управления состоянием (например, с помощью циклов и счетчиков).