Отчет по лабораторной работе 3

Рекурсия

Дата: 2025-10-03
Семестр: 3 курс 1 полугодие - 5 семестр
Группа: ПИЖ-б-о-23-2
Дисциплина: Анализ сложности алгоритмов
Студент: Мальцев Виталий Игоревич

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.

Задание:

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

```python
# Файл: memoization.py

import time
import matplotlib.pyplot as plt

# Мемоизированная версия для вычисления чисел Фибоначчи
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]

# Сравнение производительности наивной и мемоизированной версий для n=35
from recursion import fibonacci  # Импортируем функцию из recursion.py

n = 35

# Наивная версия
start_time = time.time()
result_naive = fibonacci(n)  # Из файла recursion.py
end_time = time.time()
print(f"Наивная рекурсия (n={n}): {end_time - start_time:.6f} сек")

# Мемоизированная версия
```

```
start_time = time.time()
result_memo = fibonacci_memo(n)
end_time = time.time()
print(f"Мемоизация (n={n}): {end_time - start_time:.6f} сек")

# Замеры времени выполнения для чисел Фибоначчи
times_naive = []
times_memo = []

for n in range(1, 36):
    start_time = time.time()
    fibonacci(n)  # Наивная рекурсия
    times_naive.append(time.time() - start_time)

    start_time = time.time()
    fibonacci_memo(n)  # Мемоизация
    times_memo.append(time.time() - start_time)

# Построение графика
plt.plot(range(1, 36), times_naive, label="Наивная рекурсия")
plt.plot(range(1, 36), times_memo, label="Мемоизация")
plt.xlabel("n")
plt.ylabel("Время выполнения (сек)")
plt.title("Сравнение времени выполнения")
plt.legend()

plt.savefig("fibonacci_comparison.png")

plt.show()
```

```python
# Файл: recursion.py

import sys

# 1. Вычисление факториала числа n
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

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

# 2. Вычисление n-го числа Фибоначчи
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

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

# 3. Быстрое возведение числа a в степень n
def fast_power(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        half = fast_power(a, n // 2)
        return half * half
    else:
        return a * fast_power(a, n - 1)

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

```python
# Файл: recursion_tasks.py

import os

# 1. Бинарный поиск с использованием рекурсии
def binary_search(arr, target, left, right):
    if left > right:
        return -1
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, left, mid - 1)
    else:
```

```
        return binary_search(arr, target, mid + 1, right)

# Пример использования:
arr = [1, 3, 5, 7, 9, 11, 13]
target = 7
index = binary_search(arr, target, 0, len(arr) - 1)
print(f"Элемент {target} найден по индексу {index}")

# 2. Рекурсивный обход файловой системы
def list_files(directory, indent=0):
    for item in os.listdir(directory):
        path = os.path.join(directory, item)
        print(" " * indent + item)
        if os.path.isdir(path):
            list_files(path, indent + 4)

# Пример использования:
# Замените 'path_to_directory' на путь к каталогу
# list_files('path_to_directory')

# 3. Ханойские башни
def hanoi_towers(n, source, target, auxiliary):
    if n == 1:
        print(f"Перемещаем диск 1 со стержня {source} на стержень {target}")
        return
    hanoi_towers(n - 1, source, auxiliary, target)
    print(f"Перемещаем диск {n} со стержня {source} на стержень {target}")
    hanoi_towers(n - 1, auxiliary, target, source)

# Пример использования:
hanoi_towers(3, 'A', 'C', 'B')

# Добавляем функцию для измерения максимальной глубины рекурсии при обходе
файловой системы
def list_files_with_depth(directory, indent=0, max_depth=0):
    current_depth = indent // 4
    max_depth = max(max_depth, current_depth)
    for item in os.listdir(directory):
        path = os.path.join(directory, item)
        if os.path.isdir(path):
            max_depth = list_files_with_depth(path, indent + 4, max_depth)
    return max_depth

# Пример использования:
# max_depth = list_files_with_depth('path_to_directory')
# print(f"Максимальная глубина рекурсии: {max_depth}")
```

<image src="./fibonacci_comparison.png">

```
# Контрольные вопросы

## 1. Что такое базовый случай и рекурсивный шаг в рекурсивной функции? Почему отсутствие базового случая приводит к ошибке?

- **Базовый случай (base case):** Условие, при котором рекурсия останавливается, и функция возвращает результат без дальнейших вызовов. Это "дно" рекурсии.
- **Рекурсивный шаг:** Вызов функции самой себя с изменёнными параметрами, приближаясь к базовому случаю.

**Пример:**
```python
def factorial(n):
    if n == 0:           # Базовый случай
        return 1
    else:
        return n * factorial(n - 1)  # Рекурсивный шаг
```

Почему отсутствие базового случая приводит к ошибке?
Без базового случая рекурсия становится бесконечной, что приводит к переполнению стека вызовов (`RecursionError`).

---

2. Объясните, как работает механизм мемоизации. Как он меняет временную сложность вычисления чисел Фибоначчи по сравнению с наивной рекурсией?

Мемоизация: Техника кэширования результатов для избежания повторных вычислений.

Пример:

```python
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

Сложность:

- **Наивная рекурсия:** $O(2^n)$ — экспоненциальная сложность.
- **С мемоизацией:** $O(n)$ — линейная сложность.

Вывод: Мемоизация уменьшает сложность, делая алгоритм практичным.

---

3. В чем заключается основная проблема глубокой рекурсии и как она связана со стеком вызовов?
```

Проблема: При каждом рекурсивном вызове создаётся новый фрейм в стеке. Если рекурсия слишком глубока, стек переполняется.

Последствия: Ошибка `StackOverflowError`.

Решения:

- Использовать итерацию.
- Применять хвостовую рекурсию (если поддерживается).
- Увеличивать лимит стека.

---

4. Задача о Ханойских башнях решается рекурсивно. Опишите алгоритм решения для 3 дисков.

Условие: Переместить диски с A на C, используя B как вспомогательный.

Алгоритм:

1. Переместить верхние (n−1) дисков с A на B.
2. Переместить самый нижний диск с A на C.
3. Переместить (n−1) дисков с B на C.

Для 3 дисков:

1. Переместить диск 1 с A на C.
2. Переместить диск 2 с A на B.
3. Переместить диск 1 с C на B.
4. Переместить диск 3 с A на C.
5. Переместить диск 1 с B на A.
6. Переместить диск 2 с B на C.
7. Переместить диск 1 с A на C.

---

5. Рекурсивный и итеративный алгоритмы могут решать одни и те же задачи. Назовите преимущества и недостатки каждого подхода.

Рекурсивный подход

Преимущества:

- Интуитивен для рекурсивных задач.
- Код короче и чище.

Недостатки:

- Может привести к переполнению стека.
- Больше расход памяти.

Итеративный подход

Преимущества:

- Эффективнее по памяти и времени.
- Не зависит от глубины рекурсии.

Недостатки:

- Сложнее в реализации для некоторых задач.
- Код может быть длиннее.

Выбор: Зависит от задачи и требований к производительности.
