Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

Звіт
про виконання лабораторної роботи №3
на тему:

«Обчислення ряду Фібоначчі та факторіалу різними способами»

Виконала:
Студентка групи ФеС-32
Філь Дарина

Львів - 2025

Мета:

Ознайомлення з основами функціонального програмування та його застосуванням на практиці. Порівняння рекурсивного підходу та reduce

Теоретичні відомості:

1.1. Функціональне програмування

Функціональне програмування (Functional Programming, FP) — це парадигма програмування, яка базується на використанні чистих функцій, відсутності змінюваного стану та функцій вищого порядку.

Основні концепції FP:

✅ Чисті функції – завжди повертають один і той же результат для однакових вхідних даних.

✅Немає змінюваного стану – відсутні змінні, які змінюють своє значення під час виконання.

✅ Функції вищого порядку – функції можуть приймати інші функції як аргументи або повертати їх.

✅Рекурсія замість циклів – замість for і while використовується рекурсія.

1.2. Числа ФібоначчіРяд Фібоначчі — це послідовність чисел, де кожен наступний елемент є сумою двох попередніх:

$F(n)=F(n-1)+F(n-2)F(n)=F(n-1)+F(n-2)$

Початкові значення:

$F(0)=0,F(1)=1F(0)=0,F(1)=1$

Приклад ряду Фібоначчі:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

1.3. Факторіал

Факторіал числа n (позначається n!) – це добуток всіх натуральних чисел від 1 до n:

$n!=n\times(n-1)\times(n-2)\times...\times1n!=n\times(n-1)\times(n-2)\times...\times1$

Приклади:

5!=5×4×3×2×1=1205!=5×4×3×2×1=120

Для обчислення даних прикладів, можна використовувати наступні підходи, які легко реалізувати на різних мовах програмування.

1. Рекурсія:Переваги:

○ Простота та елегантність: Рекурсія дозволяє вирішувати деякі проблеми у дуже компактний та інтуїтивно зрозумілий спосіб (наприклад, для обчислення факторіала чи чисел Фібоначчі).

○ Розв'язання складних задач: Рекурсія корисна для вирішення задач, де проблема розділяється на підзадачі того ж типу (наприклад, розв'язування задач з деревами, графами, комбінаторними задачами).

○ Покращена читабельність: Для деяких типів задач рекурсивне рішення виглядає набагато чистішим і зрозумілішим за ітераційне.

Недоліки:

○ Зростання пам'яті: Кожен рекурсивний виклик створює новий запис у стеку викликів. При великій глибині рекурсії це може призвести до переповнення стеку.

○ Низька ефективність: У разі без мемоізації, рекурсія часто призводить до дублювання обчислень, що може збільшити час виконання (особливо для задач, де одна й та сама підзадача обчислюється багато разів).

○ Витрати на підтримку: Деякі рекурсивні рішення складніші для налагодження та підтримки, особливо коли виникають проблеми з переповненням стеку.

2. Цикли:

Переваги○ Ефективність: Цикли зазвичай мають більшу ефективність щодо пам'яті, оскільки вони не створюють нові записи в стеки, а працюють в межах єдиної ітераційної змінної.

○ Швидкість: Ітеративні рішення часто швидші за рекурсивні, оскільки немає додаткових накладних витрат на виклики функцій.

○ Простота: Для багатьох задач ітерація може бути простішим і менш ресурсозатратним підходом.

Недоліки:

○ Менш інтуїтивно: Для деяких задач, зокрема тих, що мають природну рекурсивну структуру, використання циклів може бути менш інтуїтивним і важким для розуміння.

○ Менша гнучкість: Ітерація може не підходити для всіх типів задач, де логіка рішення передбачає розподіл на підзадачі (наприклад, обробка дерев чи графів).

3. Функція reduce:

Переваги:

○ Функціональний стиль програмування: reduce є частиною функціонального стилю програмування та дозволяє зберігати чистоту функцій, оскільки не змінює зовнішні стани.

○ Лаконічність: Функція reduce може бути дуже компактною для виконання таких операцій, як зведення списку до одного значення (наприклад, сума елементів або обчислення добутку).

○ Читабельність у деяких випадках: Для простих операцій reduce може виглядати зрозуміло і зручно.

Недоліки:

○ Читабельність у складних випадках: Коли логіка стає складнішою, функція reduce може бути менш зрозумілою, ніж звичайний цикл.

○ Низька продуктивність для великих колекцій: Іноді виконання з використанням reduce може бути менш ефективним, ніж просто використання циклів, оскільки зазвичай це створює додаткові функціональні виклики.

Мемоізація — це техніка оптимізації, яка полягає в збереженні результатів функцій для тих аргументів, які вже обчислювались, щоб уникнути дублювання обчислень. Цей підхід особливо корисний для рекурсивних функцій, які можуть повторно обчислювати однакові значення.

Мемоізація створює кеш (зазвичай словник або хеш-таблицю), де зберігаються результати функцій для певних параметрів. Якщо функція викликається з тими самими параметрами, результат одразу забирається з кешу, замість того щоб обчислювати його знову.

Переваги мемоізації:

○ Швидкість: Мемоізація дозволяє значно зменшити кількість обчислень, зберігаючи вже отримані результати.

○ Простота в застосуванні: Додавання мемоізації до рекурсивних функцій може бути простим, не змінюючи загальну логіку функції.

○ Особливо корисно для проблем з дублюванням обчислень:

Наприклад, у задачах, де одна й та сама підзадача виконується кілька разів (наприклад, при обчисленні чисел Фібоначчі).

Недоліки мемоізації:

○ Витрати пам'яті: Якщо функція має багато унікальних вхідних значень, кеш може зайняти велику кількість пам'яті.

○ Невигідно для простих функцій: Для функцій з простими або лінійними обчисленнями мемоізація може не дати істотного приросту в продуктивності.

Підсумки:

● Рекурсія підходить для природно рекурсивних задач, але має обмеження щодо продуктивності і пам'яті.

● Цикли є більш ефективними за часом і пам'яттю, особливо для задач, де кожен етап обчислень залежить лише від попереднього.

● reduce є функціональним методом для зведення колекцій до одного значення, але може бути менш ефективним та інтуїтивним у складних випадках.

● Мемоізація допомагає уникнути дублювання обчислень у рекурсивних функціях, підвищуючи продуктивність, але має свої обмеження з точки зору пам'яті.Приклад обчислення рекурсивно n- ого члена послідовності

Завдання до роботи:

1. Ознайомитись з теоретичним матеріалом та обчислити факторіал та ряд n-ий член послідовності Фібоначчі рекурсивно. Використовуючи пайтон або будь яку іншу мову (наприклад python, go, ruby, elixir, javascript і т. д.)

2. Обчислити факторіал та n-ий член послідовності Фібоначчі за допомогою циклу.

3. Обчислити факторіал та n-ий член послідовності Фібоначчі за допомогою reduce.

4. Використовуючи бібліотеку бенчмаркінгу порівняти час виконання для n= 10 та n = 10000.

Хід роботи

1. Ознайомитись з теоретичним матеріалом та обчислити факторіал та ряд n-ий член послідовності Фібоначчі рекурсивно. Використовуючи пайтон або будь яку іншу мову (наприклад python, go, ruby, elixir, javascript і т. д.)

```python
@lru_cache(maxsize=None)  3 usages
def fib_rec(n):
    if n <= 1:
        return 1
    return fib_rec(n - 1) + fib_rec(n - 2)

@lru_cache(maxsize=None)  2 usages
def fact_rec(n):
    if n == 0:
        return 1
    return n * fact_rec(n - 1)
```

```
n = 10
func = fib_rec
Execution time 5.500000042957254e-06
func = fact_rec
Execution time 4.099998477613553e-06
n = 3330
func = fib_rec
Execution time = ❌RecursionError
func = fact_rec
Execution time = ❌RecursionError
```

2. Обчислити факторіал та n-ий член послідовності Фібоначчі за допомогою циклу.

```python
def fib_loop(n):  1 usage
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

def fact_loop(n):  1 usage
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

```
n = 10
func = fib_loop
Execution time 3.200000719516538e-06
func = fact_loop
Execution time 2.9000002541579306e-06
```

3. Обчислити факторіал та n-ий член послідовності Фібоначчі за допомогою reduce.

```python
def fact_reduce(n):  1 usage
    return reduce(lambda a, b: a * b, range(1, n + 1))

def fib_reduce(n):  1 usage
    if n == 0:
        return 0
    seq = reduce(lambda acc, _: acc + [acc[-2] + acc[-1]], range(n - 2), [0, 1])
    return sum(seq) + 1
```

```
n = 10
func = fib_loop
Execution time 3.200000719516538e-06
func = fact_loop
Execution time 2.9000002541579306e-06
func = fact_reduce
Execution time 4.200001058052294e-06
func = fib_reduce
Execution time 7.3999999585794285e-06
```

4. Використовуючи бібліотеку бенчмаркінгу порівняти час виконання для n= 10 та n = 10000.

```python
def benchmark(n, funcs):  4 usages
    print(f"n = {n}")
    for name, func in funcs:
        try:
            time = timeit(lambda: func(n), number=1)
            print(f"func = {name}")
            print(f"Execution time {time}")
        except RecursionError:
            print(f"func = {name}")
            print("Execution time = ❌ RecursionError")
```

```
n = 10
func = fib_rec
Execution time 6.299998858594336e-06
func = fact_rec
Execution time 5.100000635138713e-06
n = 3330
func = fib_rec
Execution time = ❌RecursionError
func = fact_rec
Execution time = ❌RecursionError
n = 10
func = fib_loop
Execution time 3.2999996619764715e-06
func = fact_loop
Execution time 2.2999993234407157e-06
func = fact_reduce
Execution time 3.899998773704283e-06
func = fib_reduce
Execution time 7.3999999585794285e-06
n = 10000
func = fib_loop
Execution time 0.0015612000006512972
func = fact_loop
Execution time 0.020409799999470124
func = fact_reduce
Execution time 0.021584899999652407
func = fib_reduce
Execution time 0.1190948000003118
```

Висновок:

У цій лабораторній роботі я ознайомилася з основами функціонального програмування та його застосуванням на практиці. Порівняла рекурсивний підхід та reduce.

Додаток

Код програми:

```
from functools import reduce, lru_cache
from timeit import timeit

@lru_cache(maxsize=None)
def fib_rec(n):
    if n <= 1:
        return 1
    return fib_rec(n - 1) + fib_rec(n - 2)

@lru_cache(maxsize=None)
def fact_rec(n):
    if n == 0:
        return 1
    return n * fact_rec(n - 1)

def fib_loop(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

def fact_loop(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def fact_reduce(n):
    return reduce(lambda a, b: a * b, range(1, n + 1))

def fib_reduce(n):
```

```
    if n == 0:
        return 0
    seq = reduce(lambda acc, _: acc + [acc[-2] + acc[-1]], range(n - 2), [0, 1])
    return sum(seq) + 1

def benchmark(n, funcs):
    print(f"n = {n}")
    for name, func in funcs:
        try:
            time = timeit(lambda: func(n), number=1)
            print(f"func = {name}")
            print(f"Execution time {time}")
        except RecursionError:
            print(f"func = {name}")
            print("Execution time = ✕ RecursionError")

if __name__ == '__main__':
    funcs_rec = [("fib_rec", fib_rec), ("fact_rec", fact_rec)]
    funcs = [
        ("fib_loop", fib_loop),
        ("fact_loop", fact_loop),
        ("fact_reduce", fact_reduce),
        ("fib_reduce", fib_reduce),
    ]

    import sys
    sys.setrecursionlimit(1000000)
    sys.set_int_max_str_digits(430000)

    benchmark(10, funcs_rec)
    benchmark(3330, funcs_rec)
    benchmark(10, funcs)
    benchmark(10000, funcs)
```