МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет електроніки та комп'ютерних технологій

ЗВІТ

про виконання лабораторної роботи №4

з курсу "Функціональне програмування"

«Рекурсія та мемоізація в Python»

Виконав:

Студент 2 курсу

групи ФеП-23

Чепара Станіслав

Перевірив: Доцент Франів В.А.

Львів-2025

Мета роботи

1) Зрозуміти базові елементи рекурсії: база й крок.

2) Ознайомитися з обмеженням глибини стеку та відсутністю TCO.

3) Навчитись застосовувати мемоізацію через functools.lru_cache.

4) Реалізувати quicksort, обходи дерев, Fibonacci (memo).

Реалізація

• quicksort: випадковий pivot, чисте рекурсивне сортування, без мутацій.

• Node + обходи: preorder/inorder/postorder (рекурсивно), bfs/dfs (ітеративно).

• Fibonacci: рекурсія з @lru_cache та ітеративний варіант.

Команди перевірки

pytest -q

pytest -q --durations=5

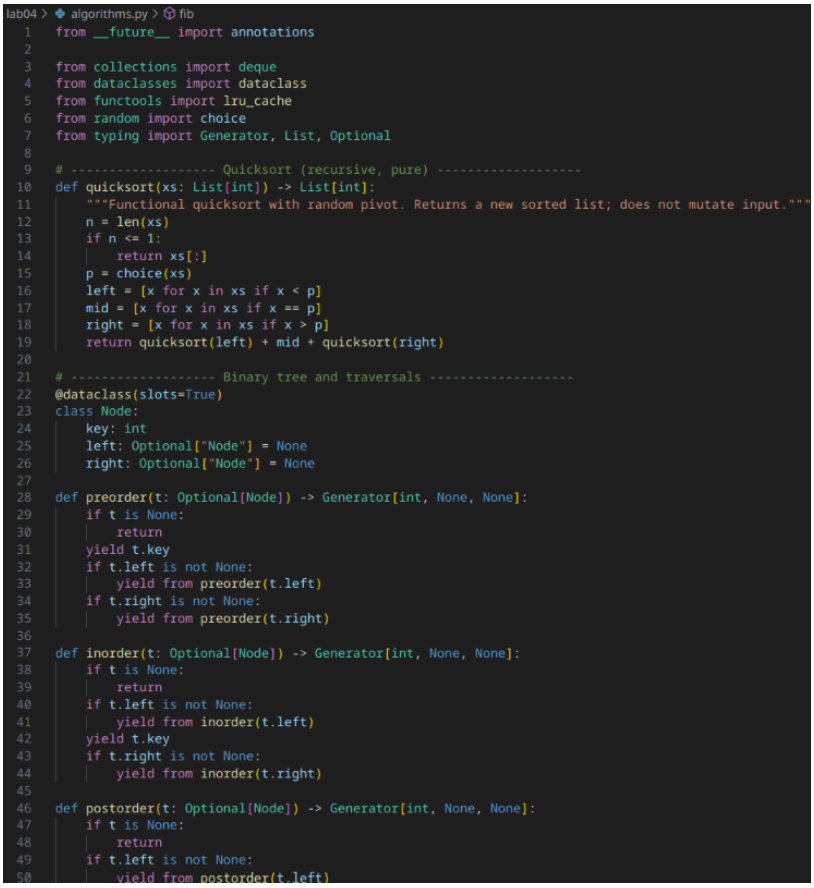
mypy lab04

black --check lab04

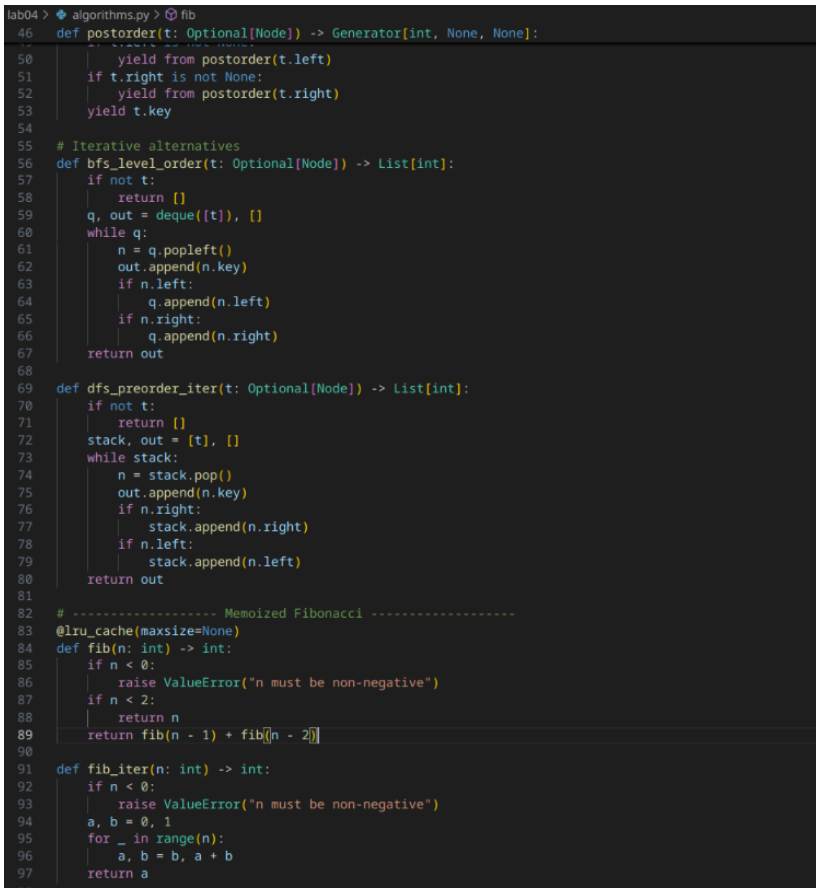
ruff check lab04

Скріншоти

algoritms.py



```
lab04 > algorithms.py > fib
from __future__ import annotations

from collections import deque
from dataclasses import dataclass
from functools import lru_cache
from random import choice
from typing import Generator, List, Optional

# ------------------- Quicksort (recursive, pure) -------------------
def quicksort(xs: List[int]) -> List[int]:
    """Functional quicksort with random pivot. Returns a new sorted list; does not mutate input."""
    n = len(xs)
    if n <= 1:
        return xs[:]
    p = choice(xs)
    left = [x for x in xs if x < p]
    mid = [x for x in xs if x == p]
    right = [x for x in xs if x > p]
    return quicksort(left) + mid + quicksort(right)

# ------------------- Binary tree and traversals -------------------
@dataclass(slots=True)
class Node:
    key: int
    left: Optional["Node"] = None
    right: Optional["Node"] = None

def preorder(t: Optional[Node]) -> Generator[int, None, None]:
    if t is None:
        return
    yield t.key
    if t.left is not None:
        yield from preorder(t.left)
    if t.right is not None:
        yield from preorder(t.right)

def inorder(t: Optional[Node]) -> Generator[int, None, None]:
    if t is None:
        return
    if t.left is not None:
        yield from inorder(t.left)
    yield t.key
    if t.right is not None:
        yield from inorder(t.right)

def postorder(t: Optional[Node]) -> Generator[int, None, None]:
    if t is None:
        return
    if t.left is not None:
        yield from postorder(t.left)
```



```
lab04 > algorithms.py > fib
def postorder(t: Optional[Node]) -> Generator[int, None, None]:
        yield from postorder(t.left)
    if t.right is not None:
        yield from postorder(t.right)
    yield t.key

# Iterative alternatives
def bfs_level_order(t: Optional[Node]) -> List[int]:
    if not t:
        return []
    q, out = deque([t]), []
    while q:
        n = q.popleft()
        out.append(n.key)
        if n.left:
            q.append(n.left)
        if n.right:
            q.append(n.right)
    return out

def dfs_preorder_iter(t: Optional[Node]) -> List[int]:
    if not t:
        return []
    stack, out = [t], []
    while stack:
        n = stack.pop()
        out.append(n.key)
        if n.right:
            stack.append(n.right)
        if n.left:
            stack.append(n.left)
    return out

# ------------------- Memoized Fibonacci -------------------
@lru_cache(maxsize=None)
def fib(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

def fib_iter(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative")
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Test_fib.py

lab04 > tests > test_fib.py > ...

```python
from __future__ import annotations

from lab04.algorithms import fib, fib_iter


def test_fib_small() -> None:
    expected = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    got = [fib(i) for i in range(10)]
    assert got == expected


def test_fib_iter_matches() -> None:
    for n in range(50):
        assert fib(n) == fib_iter(n)


def test_cache_clear() -> None:
    fib.cache_clear()
    assert fib(0) == 0

```

Test_quicksort.py

lab04 > tests > test_quicksort.py > ...

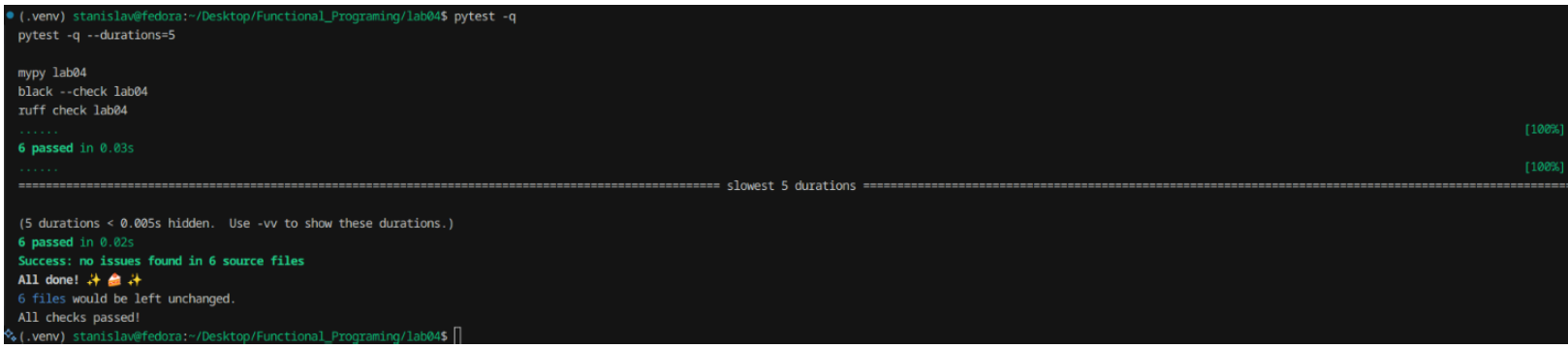
```python
from __future__ import annotations

import random

from lab04.algorithms import quicksort


def test_quicksort_basic() -> None:
    assert quicksort([]) == []
    assert quicksort([1]) == [1]
    assert quicksort([3, 2, 1]) == [1, 2, 3]
    assert quicksort([2, 2, 1, 3, 2]) == [1, 2, 2, 2, 3]


def test_quicksort_random() -> None:
    xs = [random.randint(-1000, 1000) for _ in range(1000)]
    assert quicksort(xs) == sorted(xs)

```

Test_tree.py

lab04 > tests > test_tree.py > ...

```python
from __future__ import annotations

from lab04.algorithms import (
    Node,
    bfs_level_order,
    dfs_preorder_iter,
    inorder,
    postorder,
    preorder,
)


def sample_tree() -> Node:
    #      4
    #     / \
    #    2   6
    #   / \   \
    #  1   3   7
    return Node(4, left=Node(2, Node(1), Node(3)), right=Node(6, None, Node(7)))


def test_traversals() -> None:
    t = sample_tree()
    assert list(preorder(t)) == [4, 2, 1, 3, 6, 7]
    assert list(inorder(t)) == [1, 2, 3, 4, 6, 7]
    assert list(postorder(t)) == [1, 3, 2, 7, 6, 4]
    assert bfs_level_order(t) == [4, 2, 6, 1, 3, 7]
    assert dfs_preorder_iter(t) == [4, 2, 1, 3, 6, 7]
```

Консоль



Висновки

Рекурсивні рішення з чіткою базою та мемоізацією дають коректність і продуктивність; для глибоких структур — ітеративні альтернативи.