Câu 1

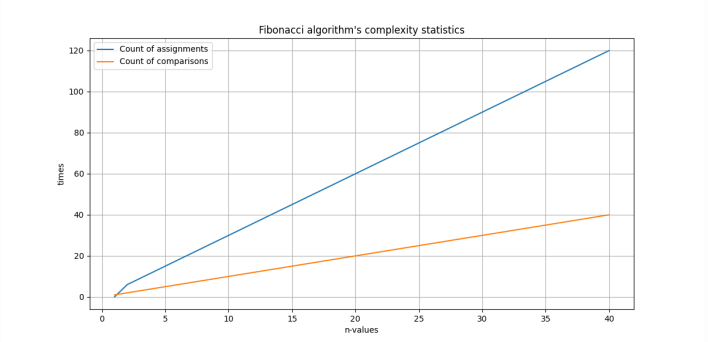
Thuật toán Fibonacci sau khi được bổ sung **số phép gán** và **số phép so sánh**:

```python
def modified_fibonacci(n: int) -> tuple[int, int, int]:
    if n <= 1:
        return (1, 0, 1) # 1 comparison for if statement

    last = 1
    nextToLast = 1
    answer = 1

    comparisons = 1  # 1 comparison for if statement
    assignments = 3  # 3 assignments for initial values

    for i in range(2, n + 1):
        answer = last + nextToLast
        nextToLast = last
        last = answer

        comparisons += 1  # 1 comparison for loop condition
        assignments += 3  # 3 assignments for updating values

    return (answer, assignments, comparisons)
```

Hàm thống kê **số phép gán** và **số phép so sánh** cho thuật toán Fibonacci trên:

```python
import matplotlib.pyplot as plt

def plot_statistics():
    n_values = list(range(1, 41))
    assignment_counts = []
    comparison_counts = []

    for n in n_values:
        _, assignments, comparisons = modified_fibonacci(n)
        assignment_counts.append(assignments)
        comparison_counts.append(comparisons)

    plt.figure(figsize=(12, 6))
    plt.plot(n_values, assignment_counts, label='Count of assignments')
    plt.plot(n_values, comparison_counts, label='Count of comparisons')
    plt.xlabel('n-values')
    plt.ylabel('times')
    plt.legend()
    plt.title('Fibonacci algorithm\'s complexity statistics')
    plt.grid(True)
    plt.show()
```

Kết quả sau khi chạy `plot_statistics()`:



Fibonacci algorithm’s complexity statistics

Đồ thị trên cho thấy **số phép gán** và **số phép so sánh** của thuật toán Fibonacci tăng tuyến tính theo `n`.

Câu 2

a - Xây dựng chương trình

Bên dưới là triển khai của 2 ý tưởng được đề xuất trong đề bài, kèm theo decorator `on_measure` để thống kê thời gian thực thi của hàm (dùng cho câu sau):

```python
import time

def on_measure(func: callable) -> callable:
    def wrapper(s1: str, s2: str):
        start_time = time.time()
        func(s1, s2)
        elapsed_time = time.time() - start_time
        return elapsed_time
    return wrapper

@on_measure
def can_be_rearranged_idea1(s1: str, s2: str) -> bool:
    if len(s1) != len(s2):
        return False

    for char in s1:
        for i in range(len(s2)):
            if char == s2[i]:
                continue
            break

    return True

@on_measure
def can_be_rearranged_idea2(s1: str, s2: str) -> bool:
    if len(s1) != len(s2):
        return False

    char_count_s1 = [0] * 26
    char_count_s2 = [0] * 26

    for char in s1:
        char_count_s1[ord(char) - ord('A')] += 1

    for char in s2:
        char_count_s2[ord(char) - ord('A')] += 1

    return char_count_s1 == char_count_s2
```

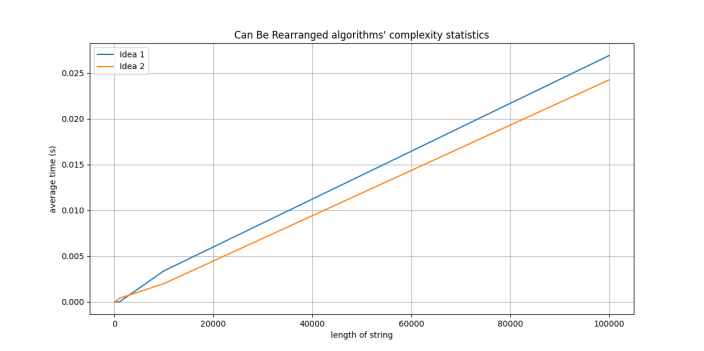
b - Thống kê thời gian thực thi

Bên dưới là hàm thống kê thời gian thực thi của 2 ý tưởng được đề xuất trong đề bài, kèm theo hàm sinh chuỗi ngẫu nhiên `random_string`.

Cho 5 mốc giá trị của `n` là 10^1, 10^2, 10^3, 10^4, 10^5. Trong mỗi mốc giá trị của `n`, lặp lại 5 lần để tìm trung bình thời gian thực thi của 2 thuật toán:

```python
import random
import matplotlib.pyplot as plt

def random_string(n: int) -> str:
    return ''.join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ', k=n))

def plot_statistics():
    n_values = [10 ** i for i in range(1, 6)] # 1e1, 1e2, 1e3, 1e4, 1e5
    avg_times_idea1 = []
    avg_times_idea2 = []

    for n in n_values:
        times_idea1 = []
        times_idea2 = []

        for j in range(5): # for finding average time
            s1 = random_string(n)
            s2 = random_string(n)

            times_idea1.append(can_be_rearranged_idea1(s1, s2))
            times_idea2.append(can_be_rearranged_idea2(s1, s2))

        avg_times_idea1.append(sum(times_idea1) / len(times_idea1))
        avg_times_idea2.append(sum(times_idea2) / len(times_idea2))


    plt.figure(figsize=(12, 6))
    plt.plot(n_values, avg_times_idea1, label='Idea 1')
    plt.plot(n_values, avg_times_idea2, label='Idea 2')
    plt.xlabel('length of string')
    plt.ylabel('average time (s)')
    plt.legend()
    plt.title('Can Be Rearranged algorithms\' complexity statistics')
    plt.grid(True)
    plt.show()
```

Kết quả sau khi chạy `plot_statistics()`:



Can Be Rearranged algorithms' complexity statistics

Đồ thị trên có thể trông khó nhìn vì khoảng cách từ 10^4 đến 10^5 quá lớn để có thể xem được chi tiết từ 10^4 trở xuống. Giảng viên có thể tự chạy code để biết thêm chi tiết.

c - Đánh giá

Ý tưởng 1 có thời gian thực thi tốt hơn khi `n` nhỏ. Tuy nhiên, khi `n` lớn (từ 10^4), ý tưởng 2 lại tỏ ra hiệu quả hơn.

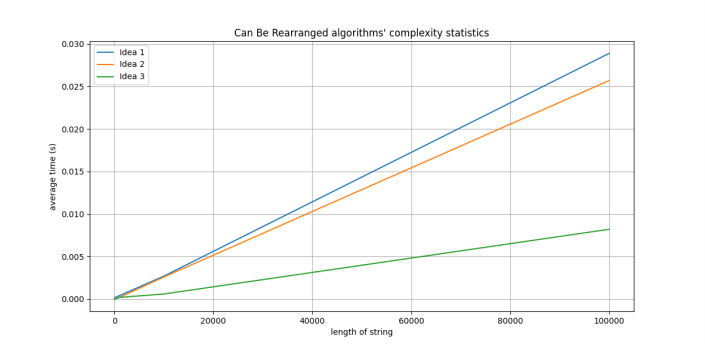
Lý do chính là do ý tưởng 1 duyệt qua 2 chuỗi lồng nhau có điều kiện `break`, độ phức tạp là $O(n\log(n))$. Trong khi ý tưởng 2 duyệt qua 2 chuỗi rời nhau, do đó độ phức tạp là $O(2n)$. Tuy nhiên mỗi lần lặp đều thực hiện các phép gán.

d - Ý tưởng khác

Ý tưởng khác là chuyển chuỗi 2 thành `set` trong Python, bản chất đây là một `hash table` với các key là các ký tự trong chuỗi 2, do đó việc kiểm tra xem một ký tự có trong chuỗi 2 hay không có độ phức tạp là $O(1)$.

```python
@on_measure
def can_be_rearranged_idea3(s1: str, s2: str) -> bool:
    if len(s1) != len(s2):
        return False

    set_s2 = set()

    for char in s2:
        set_s2.add(char)

    for char in s1:
        if char not in set_s2:
            return False

    return True
```

Kết quả sau khi chạy `plot_statistics()`:



Can Be Rearranged algorithms' complexity statistics

Đồ thị trên cho thấy ý tưởng 3 có thời gian thực thi tốt hơn nhiều so với 2 ý tưởng trong đề bài. Theo quan sát, con số này là gấp khoảng 3 lần.

Lí do chính là do ý tưởng 3 duyệt qua 2 chuỗi rời nhau. Do đó độ phức tạp là $O(2n)$. Tuy nhiên khác với ý tưởng 2 ở chỗ, ý tưởng 3 không thực hiện nhiều phép gán như ý tưởng 2.