STRUKTUR DATA
FUNGSI REKURSI DAN ALGORITMA SORTING

Oleh:
DANUARY BIMA HAMMAM MAYFALAH
24091397007

Program Studi D4 Manajemen Informatika
Fakultas Vokasi
Universitas Negeri Surabaya
2025

A. TUGAS

1. Rekursi dan Penerapannya dalam Pemecahan Masalah

Tujuan: Mahasiswa memahami konsep rekursi dan bagaimana rekursi digunakan untuk menyelesaikan berbagai masalah dalam pemrograman.

- Tulis dan analisis kode rekursif untuk menghitung faktorial (n = 15).
- Bandingkan implementasi rekursi dan iterasi dalam menghitung faktorial.
- Implementasikan rekursi untuk menghitung bilangan Fibonacci ke-25. • Analisis kompleksitas waktu dari algoritma rekursi Fibonacci.

2. Implementasi Rekursi dalam Python

Tujuan: Mahasiswa mampu mengimplementasikan rekursi dalam Python untuk berbagai permasalahan umum.

- Implementasikan fungsi rekursif menyelesaikan Tower of Hanoi dengan 2 cakram. • Ubah program Tower of Hanoi agar menerima input jumlah cakram dari user.

3. Algoritma Sorting

Tujuan: Mahasiswa memahami dan mengimplementasikan berbagai algoritma sorting dalam Python.

Data Input = [115, 18, 45, 29, 56, 1, 37]

- Implementasikan Bubble Sort dan analisis jumlah perbandingan serta pertukaran elemen yang terjadi.
- Implementasikan Selection Sort dan Bubble Sort. • Implementasikan juga Quick Sort, lalu bandingkan dengan sorting lainnya.

4. Implementasi Sorting dan Rekursi Menggunakan OOP

Tujuan: Mahasiswa mampu menerapkan konsep OOP dalam implementasi rekursi dan sorting.

- Modifikasi program Fibonacci menggunakan konsep class dan method.
- Implementasikan Class Sorting yang mencakup metode Bubble Sort, Insertion Sort, dan Selection Sort dalam satu class.
- Tambahkan Method tambahan dalam Class Sorting untuk mengurutkan dalam urutan sebaliknya, menurun (Descending).
- Bandingkan waktu eksekusi antara algoritma sorting menggunakan OOP dan tanpa OOP.

5. Perbandingan Efisiensi Algoritma Sorting

Tujuan: Mahasiswa membandingkan efisiensi berbagai algoritma sorting berdasarkankompleksitas waktu dan ruang.

- Gunakan Library Python time untuk mengukur waktu eksekusi dari setiap algoritma sorting pada data.
- Buat visualisasi grafik perbandingan efisiensi sorting menggunakan pustaka matplotlib.
- Analisis kompleksitas waktu dari masing-masing algoritma sorting dan simpulkan kapan sebaiknya menggunakan masing-masing algoritma.

B. JAWABAN

1) Source Code

```python
def faktorial (n):
    if n == 0 or n == 1:
        return 1
    else :
        return n * faktorial (n - 1)

print (faktorial(15))
```

Penjelasan:

- Basis kasus: Jika n == 0 atau n == 1, kembalikan 1.
- Rekurens: Untuk n > 1, hasil = n * faktorial_rekursif(n-1).

Output

```
faktorial dari 15 adalah: 1307674368000
```

- Perbandingan Rekursi vs Iterasi dalam Menghitung Faktorial

```python
def faktorial_iteratif(n):
    hasil = 1
    for i in range(2, n+1):
        hasil *= i
    return hasil

hasil_iteratif = faktorial_iteratif(15)
print("Faktorial 15 (iteratif):", hasil_iteratif)
```

Output

```
Faktorial 15 (iteratif): 1307674368000
```

Perbandingan:

Aspek	Rekursi	Iterasi
Kelebihan	Lebih elegan dan dekat dengan definisi matematis.	Umumnya lebih cepat dan hemat memori.
Kekurangan	Stack memory bertambah dengan setiap pemanggilan, berisiko stack overflow untuk n besar.	Harus secara eksplisit mengelola perulangan.
Komplektibitas Waktu	O(n)	O(n)
Komplektibitas Ruang	O(n) (karena stack rekursif)	O(1)

- Implementasi Rekursif untuk Menghitung Bilangan Fibonacci ke-25

Source Code

```
def fibonacci_rekursif(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_rekursif(n-1) + fibonacci_rekursif(n-2)

fibonacci = fibonacci_rekursif(25)
print("Bilangan Fibonacci ke-25:", fibonacci)
```

Output

```
Bilangan Fibonacci ke-25: 75025
```

- Analisis Kompleksitas Waktu Rekursi Fibonacci

Analisis:

- Setiap pemanggilan fibonacci_rekursif(n) memanggil 2 fungsi: fibonacci_rekursif(n-1) dan fibonacci_rekursif(n-2).
- Ini membentuk pohon biner di mana jumlah simpul ≈ 2^n.
- Kompleksitas waktu = O(2^n)
  (sangat lambat untuk nilai n besar seperti n = 25 atau lebih).

2) Fungsi Rekursif Menyelesaikan Tower of Hanoi dengan 2 Cakram

Source code

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print (f"Pindahkan cakram 1 dari {source} ke {target}")
        return
    tower_of_hanoi (n - 1, auxiliary, source, target)
    print(F"Pindahkan cakram {n} dari {source} ke {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)
print("Tower of Hanoi dengan 2 cakram:")
tower_of_hanoi(2, 'A','B','C')
```

Output

```
Tower of Hanoi dengan 2 cakram:
Pindahkan cakram 1 dari B ke C
Pindahkan cakram 2 dari A ke C
Pindahkan cakram 1 dari B ke C
```

- Modifikasi Tower of Hanoi agar Menerima Input Jumlah Cakram dari User

Source code

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print (f"Pindahkan cakram 1 dari {source} ke {target}")
        return
    tower_of_hanoi (n - 1, auxiliary, source, target)
    print(F"Pindahkan cakram {n} dari {source} ke {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)
jumlah_cakram = int(input("Masukkan jumlah cakram: "))
print(f"\nLangkah-langkah Tower of Hanoi dengan {jumlah_cakram} cakram:")
tower_of_hanoi(jumlah_cakram, 'A', 'B', 'C')
```

Output

```
Masukkan jumlah cakram: 3

Langkah-langkah Tower of Hanoi dengan 3 cakram:
Pindahkan cakram 1 dari A ke C
Pindahkan cakram 2 dari B ke C
Pindahkan cakram 1 dari A ke C
Pindahkan cakram 3 dari A ke C
Pindahkan cakram 1 dari A ke C
Pindahkan cakram 2 dari B ke C
Pindahkan cakram 1 dari A ke C
```

Alur Rekursi Tower of Hanoi:

- Jika hanya ada satu cakram, langkahnya langsung dipindahkan ke tiang tujuan.
  Jika lebih dari satu cakram:
  - Pertama, pindahkan n-1 cakram ke tiang bantu.
  - Lalu, pindahkan cakram terbesar ke tiang tujuan.
  - Akhirnya, pindahkan kembali n-1 cakram dari tiang bantu ke tiang tujuan.

3) Implementasi Bubble Sort

Source Code

```python
def bubble_sort(arr):
    n = len(arr)
    total_perbandingan = 0
    total_pertukaran = 0

    for i in range(n-1):
        for j in range(n-1-i):
            total_perbandingan += 1
            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]
                total_pertukaran += 1
    return arr, total_perbandingan, total_pertukaran

data = [115, 18, 45, 29, 56, 1, 37]
sorted_bubble, perbandingan_bubble, pertukaran_bubble = bubble_sort(data.copy())

print("Hasil Bubble Sort:", sorted_bubble)
print("Jumlah perbandingan:", perbandingan_bubble)
print("Jumlah pertukaran:", pertukaran_bubble)
```

Output

```
Hasil Bubble Sort: [1, 18, 29, 37, 45, 56, 115]
Jumlah perbandingan: 21
Jumlah pertukaran: 13
```

➢ Implementasi Selection Sort

Source Code

```python
def selection_sort(arr):
    n = len(arr)
    total_perbandingan = 0
    total_pertukaran = 0

    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            total_perbandingan += 1
            if arr[j] < arr[min_idx]:
                min_idx = j
        if min_idx != i:
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
            total_pertukaran += 1

    return arr, total_perbandingan, total_pertukaran

# Menjalankan Selection Sort
data = [115, 18, 45, 29, 56, 1, 37]
sorted_selection, perbandingan_selection, pertukaran_selection = selection_sort(data.copy())

print("\nHasil Selection Sort:", sorted_selection)
print("Jumlah perbandingan:", perbandingan_selection)
print("Jumlah pertukaran:", pertukaran_selection)
```

Output

```
Hasil Selection Sort: [1, 18, 29, 37, 45, 56, 115]
Jumlah perbandingan: 21
Jumlah pertukaran: 5
```

➢ Implementasi quick sort

Source Code

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x <= pivot]
        right = [x for x in arr[1:] if x > pivot]
        return quick_sort(left) + [pivot] + quick_sort(right)

data = [115, 18, 45, 29, 56, 1, 37]
sorted_quick = quick_sort(data.copy())

print("\nHasil Quick Sort:", sorted_quick)
```

Output

```
Hasil Quick Sort: [1, 18, 29, 37, 45, 56, 115]
```

Perbandingan:

Metode	Waktu Rata-rata	Perbandingan	Pertukaran	Keterangan
Bubble Sort	$O(n^2)$	Banyak	Banyak	Lambat Jika data besar
Selection Sort	$O(n^2)$	Banyak	Lebih sedikit	Tetap lambat untuk data besar
Quick Sort	O(n log n)	Optimal	Optimal	Cepat untuk data besar

4) Modifikasi Program Fibonacci menggunakan Konsep Class dan Method

Source Code

```
class Fibonacci:
    def __init__(self, n):
        self.n = n

    def hitung(self):
        if self.n <= 1:
            return self.n
        else:
            return self._fibonacci(self.n)

    def _fibonacci(self, n):
        if n <= 1:
            return n
        else:
            return self._fibonacci(n-1) + self._fibonacci(n-2)

# Contoh penggunaan:
fib = Fibonacci(10)
print("Fibonacci ke-10:", fib.hitung())
```

Output

```
Fibonacci ke-10: 55
```

➢ Implementasi Class Sorting dengan Bubble Sort, Insertion Sort, dan Selection Sort

Source Code

```
class Sorting:
    def __init__(self, data):
        self.data = data.copy()

    def bubble_sort(self, ascending=True):
        arr = self.data.copy()
        n = len(arr)
        for i in range(n-1):
            for j in range(n-1-i):
                if (ascending and arr[j] > arr[j+1]) or (not ascending and arr[j] < arr[j+1]):
                    arr[j], arr[j+1] = arr[j+1], arr[j]
        return arr

    def insertion_sort(self, ascending=True):
        arr = self.data.copy()
        n = len(arr)
        for i in range(1, n):
            key = arr[i]
            j = i - 1
            while j >= 0 and ((ascending and arr[j] > key) or (not ascending and arr[j] < key)):
                arr[j+1] = arr[j]
                j -= 1
            arr[j+1] = key
        return arr

    def selection_sort(self, ascending=True):
        arr = self.data.copy()
        n = len(arr)
        for i in range(n):
            idx = i
            for j in range(i+1, n):
                if (ascending and arr[j] < arr[idx]) or (not ascending and arr[j] > arr[idx]):
                    idx = j
            arr[i], arr[idx] = arr[idx], arr[i]
        return arr

data = [111, 20, 98, 78, 50, 9, 31]
sorter = Sorting(data)

print("Bubble Sort Ascending:", sorter.bubble_sort(ascending=True))
print("Bubble Sort Descending:", sorter.bubble_sort(ascending=False))

print("Insertion Sort Ascending:", sorter.insertion_sort(ascending=True))
print("Insertion Sort Descending:", sorter.insertion_sort(ascending=False))

print("Selection Sort Ascending:", sorter.selection_sort(ascending=True))
print("Selection Sort Descending:", sorter.selection_sort(ascending=False))
```

Output

```
Bubble Sort Ascending: [9, 20, 31, 50, 78, 98, 111]
Bubble Sort Descending: [111, 98, 78, 50, 31, 20, 9]
Insertion Sort Ascending: [9, 20, 31, 50, 78, 98, 111]
Insertion Sort Descending: [111, 98, 78, 50, 31, 20, 9]
Selection Sort Ascending: [9, 20, 31, 50, 78, 98, 111]
Selection Sort Descending: [111, 98, 78, 50, 31, 20, 9]
```

- Bandingkan Waktu Eksekusi: OOP vs Non-OOP

Source Code

```python
import time

def bubble_sort_non_oop(arr):
    arr = arr.copy()
    n = len(arr)
    for i in range(n-1):
        for j in range(n-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

start = time.time()
bubble_sort_non_oop(data)
end = time.time()
print("\nWaktu Non-OOP Bubble Sort:", end - start, "detik")

sorter = Sorting(data)
start = time.time()
sorter.bubble_sort()
end = time.time()
print("Waktu OOP Bubble Sort:", end - start, "detik")
```

Output

```
Waktu Non-OOP Bubble Sort: 6.67572021484375e-06 detik
Waktu OOP Bubble Sort: 5.7220458984375e-06 detik
```

Perbandingan:

Aspek	Non-oop	Oop
Struktur	Lebih simpel	Lebih terorganisasi (rapih)
Penulisan	Sedikit kode	Sedikit lebih panjang
Waktu Eksekusi	Lebih cepat sedikit	Sedikit lebih lambat (karena overhead class)
Kelebihan	Mudah, cepat dibuat	Mudah dikembangkan, maintenance lebih bagus

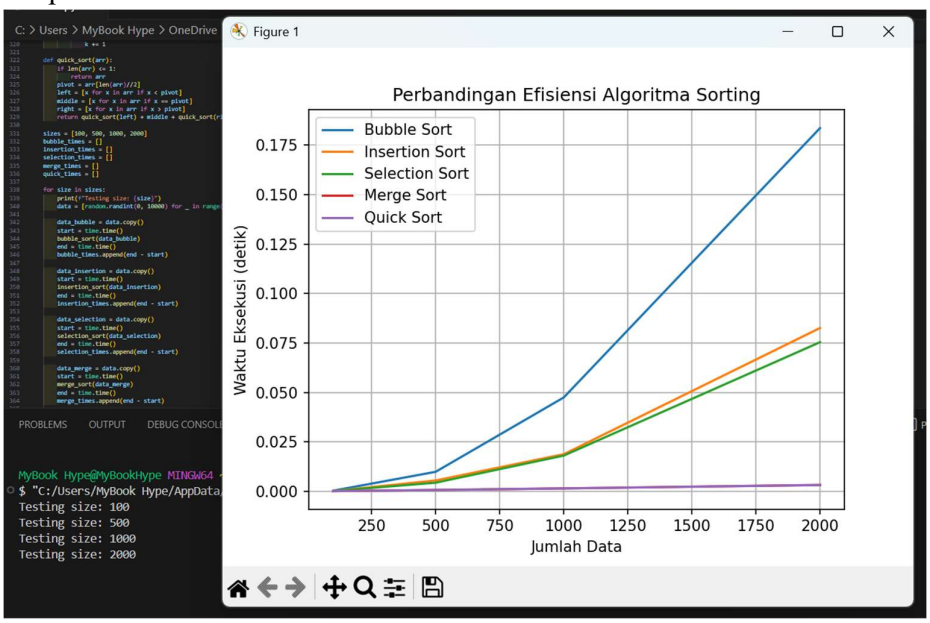
5) Mengukur Waktu Eksekusi Setiap Algoritma Sorting

Source Code

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

def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

sizes = [100, 500, 1000, 2000]
bubble_times = []
insertion_times = []
selection_times = []
merge_times = []
quick_times = []
```

```python
for size in sizes:
    print(f"Testing size: {size}")
    data = [random.randint(0, 10000) for _ in range(size)]

    data_bubble = data.copy()
    start = time.time()
    bubble_sort(data_bubble)
    end = time.time()
    bubble_times.append(end - start)

    data_insertion = data.copy()
    start = time.time()
    insertion_sort(data_insertion)
    end = time.time()
    insertion_times.append(end - start)

    data_selection = data.copy()
    start = time.time()
    selection_sort(data_selection)
    end = time.time()
    selection_times.append(end - start)

    data_merge = data.copy()
    start = time.time()
    merge_sort(data_merge)
    end = time.time()
    merge_times.append(end - start)

    data_quick = data.copy()
    start = time.time()
    quick_sort(data_quick)
    end = time.time()
    quick_times.append(end - start)

plt.plot(sizes, bubble_times, label='Bubble Sort')
plt.plot(sizes, insertion_times, label='Insertion Sort')
plt.plot(sizes, selection_times, label='Selection Sort')
plt.plot(sizes, merge_times, label='Merge Sort')
plt.plot(sizes, quick_times, label='Quick Sort')

plt.title('Perbandingan Efisiensi Algoritma Sorting')
plt.xlabel('Jumlah Data')
plt.ylabel('Waktu Eksekusi (detik)')
plt.legend()
plt.grid(True)
plt.show()
```

Output



Perbandingan:

Algoritma	Kompleksitas Terburuk	Kompleksitas Terbaik	Kompleksitas Rata-rata	Catatan Penggunaan
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	Gunakan hanya untuk dataset kecil atau hampir terurut
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	Lebih baik untuk dataset kecil atau hampir terurut
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Stabil tapi lambat; dipakai kalau ingin minim pertukaran
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Sangat efisien untuk data besar, tapi kurang stabil
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sangat efisien untuk data besar dan stabil