Modul_5_PBF

April 14, 2025

1 Modul 5 Praktikum Pemrograman Berbasis Fungsi: Rekursi dan Reduksi

1.1 Tujuan Praktikum

1. Mahasiswa mampu menjelaskan cara kerja fungsi rekursif dan mengidentifikasi struktur dasar dari fungsi rekursif.
2. Mahasiswa dapat mengoptimalkan fungsi rekursif agar efisien menggunakan teknik TCO dan accumulator untuk mencegah stack overflow.
3. Mahasiswa memahami prinsip kerja fungsi reduce (reduksi) dan penggunaannya untuk merangkum kumpulan data.

1.2 Rekursi dan Tail-Call Optimization

Banyak bahasa fungsional melakukan tail-call optimization (TCO), yaitu mengubah panggilan rekursif terakhir menjadi iterasi untuk meningkatkan performa. Python tidak otomatis melakukan TCO, sehingga untuk efisiensi, kita perlu mengubah rekursi menjadi iterasi secara manual.

Teknik manual ini umumnya dilakukan dengan: - Menggunakan struktur iteratif eksplisit (for, while) - Menggunakan struktur data seperti deque atau stack

1.2.1 Apa itu Rekursi?

Rekursi adalah teknik dalam pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah dalam sub-bagian yang lebih kecil.

1.2.2 Ciri-ciri fungsi rekursif:

- Base case (kondisi berhenti): kondisi yang akan menghentikan pemanggilan rekursif.
- Recursive case: bagian di mana fungsi memanggil dirinya sendiri dengan argumen berbeda yang mendekati base case.

Contoh Dasar Rekursi 1. Faktorial (n!)

[1]:
```
def factorial(n):
    if n == 0:
        return 1  # base case
    return n * factorial(n - 1)  # recursive call

print(factorial(5))  # Output: 120
```

```
120
```

2. Deret Fibonacci

[2]:
```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(6))  # Output: 8
```

```
8
```

1.2.3 Apa itu Tail-Call Optimization (TCO)?

Tail-Call Optimization adalah teknik di mana pemanggilan rekursif terakhir (tail call) dioptimalkan menjadi bentuk iteratif oleh compiler atau interpreter untuk menghemat penggunaan stack frame dan mencegah stack overflow.

`Tail call terjadi ketika pemanggilan fungsi rekursif berada di posisi terakhir sebelum return.`

Tujuan Tail-Call Optimization:

Supaya lebih hemat memori dan tidak membuat call stack menumpuk. Dalam TCO, kita menggunakan accumulator (penampung hasil) dan memastikan pemanggilan rekursi ada di langkah terakhir. Accumulator (dalam konteks rekursi) adalah variabel tambahan yang digunakan untuk menyimpan hasil sementara dari proses rekursif. Tujuannya agar hasil akhir bisa dibentuk secara bertahap tanpa harus menunggu semua rekursi selesai.

Python tidak mendukung TCO secara otomatis

Bahasa seperti Scheme atau Haskell mengoptimalkan tail call, tapi Python tidak.

Solusi: Tail-Call Optimization Manual

Kita bisa mengubah rekursi tail-call menjadi bentuk iteratif eksplisit untuk mencegah kelebihan stack (stack overflow).

1.2.4 Contoh Perbandingan: Rekursi Biasa vs Tail-Recursive

1. Rekursi Biasa (Faktorial)

[3]:
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

2. Rekursi Tail (masih belum efisien di Python)

[4]:
```
def tail_factorial(n, acc=1):
    if n == 0:
        return acc
    return tail_factorial(n - 1, acc * n)
```

3. Iteratif (Manual TCO)

[5]:
```python
def iterative_factorial(n):
    acc = 1
    while n > 0:
        acc *= n
        n -= 1
    return acc

print(iterative_factorial(5))  # Output: 120
```

```
120
```

1.2.5 Contoh Optimasi Manual: Fibonacci

1. Rekursi Biasa

[6]:
```python
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

2. Iteratif

[7]:
```python
def fib_iter(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

print(fib_iter(50))  # Output: 12586269025
```

```
12586269025
```

- Rekursi adalah teknik penting dalam pemrograman fungsional.
- Python tidak mengoptimasi tail call secara otomatis, sehingga perlu manual TCO dengan cara iterasi.
- Tail-recursive functions lebih aman terhadap stack overflow, tapi tetap perlu diubah ke bentuk iteratif di Python.
- Fungsi seperti faktorial dan fibonacci bisa menjadi contoh baik dalam menjelaskan perbedaan antara rekursi biasa, tail-recursive, dan iterasi.

Cara Mengubah Rekursi Biasa ke TCO 1. Tambahkan parameter accumulator untuk menyimpan hasil sementara. 2. Pindahkan perhitungan ke sebelum pemanggilan rekursif. 3. Pastikan pemanggilan rekursif adalah langkah terakhir.

Contoh: Jumlah Deret 1 sampai n - Rekursif Biasa:

[22]:
```python
def sum_recursive(n):
    if n == 0:
        return 0
    return n + sum_recursive(n - 1)
#tanpa accumulator
print(sum_recursive(5))  # Output: 15
```

15

- Tail-call Optimized:

[23]:
```python
def sum_tco(n, acc=0):
    if n == 0:
        return acc
    return sum_tco(n - 1, acc + n)
# menggunakan accumulator atau acc
# acc + n adalah nilai yang diakumulasikan
# pada setiap pemanggilan fungsi
# acc = 0 adalah nilai awal dari accumulator
# pada pemanggilan pertama
# nilai accumulator akan bertambah 1 pada setiap pemanggilan fungsi
# sehingga pada pemanggilan terakhir akan menghasilkan nilai 15
# pada pemanggilan pertama n = 5, acc = 0
# pada pemanggilan kedua n = 4, acc = 5
# pada pemanggilan ketiga n = 3, acc = 9
# pada pemanggilan keempat n = 2, acc = 12
# pada pemanggilan kelima n = 1, acc = 14
# pada pemanggilan keenam n = 0, acc = 15
# pada pemanggilan terakhir n = 0, acc = 15
# sehingga hasilnya adalah 15

print(sum_tco(5))  # Output: 15
```

15

1.2.6 Latihan Rekursi

1. Hitung Mundur

Deskripsi: Cetak angka dari n ke 1 menggunakan rekursi.

Contoh: Input: 5 → Output: 5 4 3 2 1

[ ]:
```python
def countdown(n):
    # TODO: tulis base case dan pemanggilan rekursif
```

[9]:
```python
def countdown(n):
    if n == 0:
        return
    print(n)
```

```
    countdown(n - 1)

# Example
countdown(5)
```

```
5
4
3
2
1
```

2. Jumlah Deret 1 sampai n

Deskripsi: Hitung jumlah dari 1 + 2 + … + n menggunakan rekursi.

Contoh: Input: 4 → Output: 10

[ ]:
```
def sum_to_n(n):
    # TODO: base case dan rekursi
```

[10]:
```
def sum_to_n(n):
    if n == 0:
        return 0
    return n + sum_to_n(n - 1)

# Example
print(sum_to_n(4))  # Output: 10
```

```
10
```

3. Faktorial

Deskripsi: Hitung faktorial dari n (n!).

Contoh: Input: 5 → Output: 120

[ ]:
```
def factorial(n):
    # TODO: base case dan recursive call
```

[11]:
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

# Example
print(factorial(5))  # Output: 120
```

```
120
```

4. Palindrome Checker

Deskripsi: Cek apakah sebuah string adalah palindrome (dibaca sama dari depan & belakang).

Contoh: Input: “katak” → Output: True

[ ]:
```python
def is_palindrome(s):
    # Petunjuk: cek karakter pertama dan terakhir, lalu rekur pada bagian tengah
```

[12]:
```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])

# Example
print(is_palindrome("katak"))  # Output: True
```

```
True
```

5. Jumlah Digit Angka

Deskripsi: Hitung jumlah dari semua digit dalam angka.

Contoh: Input: 1234 → Output: 10

[ ]:
```python
def digit_sum(n):
    # Petunjuk: gunakan modulo 10 dan pembagian integer
```

[13]:
```python
def digit_sum(n):
    if n == 0:
        return 0
    return (n % 10) + digit_sum(n // 10)

# Example
print(digit_sum(1234))  # Output: 10
```

```
10
```

6. Binary Representation

Deskripsi: Ubah bilangan desimal ke biner menggunakan rekursi.

Contoh: Input: 6 → Output: ‘110’

[ ]:
```python
def to_binary(n):
    # Petunjuk: n % 2 dan return ke n // 2
```

[14]:
```python
def to_binary(n):
    if n == 0:
        return ""
    return to_binary(n // 2) + str(n % 2)

# Example
print(to_binary(6))  # Output: '110'
```

```
110
```

7. Reverse String

Deskripsi: Balik string menggunakan rekursi.

Contoh: Input: “halo” → Output: “olah”

[ ]:
```python
def reverse_string(s):
    # Petunjuk: ambil karakter terakhir dan rekur sisanya
```

[15]:
```python
def reverse_string(s):
    if len(s) == 0:
        return ""
    return s[-1] + reverse_string(s[:-1])

# Example
print(reverse_string("halo"))  # Output: 'olah'
```

```
olah
```

8. Permutasi Karakter

Deskripsi: Cetak semua kemungkinan urutan karakter dari string.

Contoh: Input: “abc” → Output: abc, acb, bac, bca, cab, cba

[ ]:
```python
def permute(s, answer=""):
    # Petunjuk: gunakan loop dan panggil permute untuk sisa karakter
```

[16]:
```python
def permute(s, answer=""):
    if len(s) == 0:
        print(answer)
        return
    for i in range(len(s)):
        ch = s[i]
        left = s[:i]
        right = s[i+1:]
        rest = left + right
        permute(rest, answer + ch)

# Example
permute("abc")
```

```
abc
acb
bac
bca
cab
cba
```

9. Maze Paths (tanpa backtracking)

Deskripsi: Hitung jumlah cara untuk mencapai ujung kanan bawah dari grid n x m hanya bisa ke kanan dan bawah.

Contoh: Input: (2, 2) → Output: 2

[ ]:
```
def count_paths(r, c):
    # base case: jika satu baris atau kolom, hanya 1 cara
```

[17]:
```
def count_paths(r, c):
    if r == 1 or c == 1:
        return 1
    return count_paths(r - 1, c) + count_paths(r, c - 1)

# Example
print(count_paths(2, 2))  # Output: 2
```

```
2
```

10. Tower of Hanoi

Deskripsi: Tampilkan langkah-langkah memindahkan n cakram dari tiang A ke C menggunakan B sebagai bantuan.

Contoh: Input: n = 3 Output:

```
Move disk 1 from A to C
Move disk 2 from A to B
...
```

[ ]:
```
def hanoi(n, source, helper, target):
    # Petunjuk: rekur pindahkan n-1 dari source ke helper, lalu pindahkan␣
 ↪cakram ke target
```

[18]:
```
def hanoi(n, source, helper, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    hanoi(n - 1, source, target, helper)
    print(f"Move disk {n} from {source} to {target}")
    hanoi(n - 1, helper, source, target)

# Example
hanoi(3, "A", "B", "C")
```

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

1.2.7 Versi TCO dari sum_to_n(n)

[21]:
```python
# Rekursi Biasa

def sum_to_n(n):
    if n == 0:
        return 0
    return n + sum_to_n(n - 1)

sum_to_n(4)
```

[21]: 10

[20]:
```python
# Tail-Call Optimized:

def sum_to_n_tco(n, acc=0):
    if n == 0:
        return acc
    return sum_to_n_tco(n - 1, acc + n)

# Example
print(sum_to_n_tco(4))  # Output: 10
```

```
10
```

1.3 Reduksi (Reduction)

Reduksi adalah proses mengubah koleksi data menjadi nilai tunggal atau struktur data baru.

Contoh fungsi reduksi bawaan Python: - sum(), max(), min(), len() - functools.reduce()

Fungsi seperti collections.Counter() dan itertools.groupby() juga dianggap sebagai bentuk reduksi, karena fungsi ini merangkum data menjadi bentuk yang lebih terorganisir. Proses parsing seperti pengubahan token menjadi struktur data juga merupakan bentuk reduksi.

Catatan Penting

Teknik reduksi bisa didefinisikan secara rekursif, dan teknik optimasi tail-call dapat digunakan. Parsing sering dibagi dua tahap: - Level rendah: mengubah teks mentah menjadi tuple - Level tinggi: mengubah tuple menjadi objek dengan struktur kompleks

1.3.1 Apa itu Reduksi (Reduction)?

Reduksi adalah proses di mana kita menggabungkan elemen-elemen dari sebuah struktur data (biasanya list/array) menjadi satu nilai tunggal menggunakan operasi tertentu, seperti penjumlahan, perkalian, minimum, maksimum, logika, atau penggabungan string.

Reduksi sering digunakan dalam functional programming dan biasanya dilakukan dengan fungsi reduce().

1.3.2 Ciri-ciri Operasi Reduksi

- Melibatkan iterasi atas seluruh elemen.
- Menggunakan fungsi akumulator.
- Mengembalikan satu nilai akhir.
- Bersifat associative, artinya urutan penggabungan tidak memengaruhi hasil (kecuali dalam kasus tertentu, misalnya string).

1.3.3 Fungsi reduce() di Python

Untuk menggunakan reduce() di Python, perlu mengimpor dari modul functools:

```
from functools import reduce
```

Bentuk Umum:

```
reduce(function, iterable, initial_value)
```

1.3.4 Contoh Dasar Reduksi

1. Jumlah Semua Angka

[ ]:
```python
from functools import reduce

data = [1, 2, 3, 4, 5]

result = reduce(lambda acc, x: acc + x, data, 0)
print(result)  # Output: 15

# Menggunakan fungsi reduce untuk menjumlahkan elemen dalam list
# Fungsi reduce menerima 3 argumen
# 1. Fungsi yang akan diterapkan pada setiap elemen
# 2. Iterable yang akan diproses
# 3. Nilai awal dari accumulator (opsional)
# Fungsi lambda digunakan untuk menjumlahkan elemen satu per satu
# Fungsi reduce akan mengembalikan hasil akhir dari penjumlahan
# Cara kerja lambda:
# Fungsi lambda menerima 2 argumen, acc dan x
# acc adalah nilai akumulasi yang dihasilkan dari iterasi sebelumnya
# x adalah elemen saat ini dari iterable
# Pada iterasi pertama, acc = 0 dan x = 1
# Pada iterasi kedua, acc = 1 dan x = 2
# Pada iterasi ketiga, acc = 3 dan x = 3
# Pada iterasi keempat, acc = 6 dan x = 4
# Pada iterasi kelima, acc = 10 dan x = 5
# Pada iterasi terakhir, acc = 15 dan x = None (tidak ada elemen lagi)
# Sehingga hasil akhirnya adalah 15
# Fungsi lambda digunakan untuk menjumlahkan elemen satu per satu
# Fungsi reduce akan mengembalikan hasil akhir dari penjumlahan
# Output: 15
```

15

2. Perkalian Semua Angka

[ ]:
```
result = reduce(lambda acc, x: acc * x, data, 1)
print(result)  # Output: 120

# Menggunakan fungsi reduce untuk mengalikan elemen dalam list
# Fungsi reduce menerima 3 argumen
# 1. Fungsi yang akan diterapkan pada setiap elemen
# 2. Iterable yang akan diproses
# 3. Nilai awal dari accumulator (opsional)
# Cara kerja lambda sama seperti sebelumnya
# Fungsi reduce akan mengembalikan hasil akhir dari perkalian
# Output: 120
```

120

3. Gabung Semua String

[17]:
```
words = ["Hello", " ", "World", "!"]

result = reduce(lambda acc, x: acc + x, words)
print(result)  # Output: "Hello World!"
```

Hello World!

4. Cari Nilai Maksimum

[18]:
```
numbers = [3, 9, 2, 8, 5]

result = reduce(lambda a, b: a if a > b else b, numbers)
print(result)  # Output: 9
```

9

5. Cari Nilai Minimum

[20]:
```
result = reduce(lambda a, b: a if a < b else b, numbers)
print(result)  # Output: 2
```

2

6. Menghitung Panjang Total String

[19]:
```
strings = ["hello", "world", "python"]

total_length = reduce(lambda acc, s: acc + len(s), strings, 0)
print(total_length)  # Output: 16
```

16

7. Menghitung Total Harga Barang

[21]:
```
cart = [
    {"item": "Book", "price": 100},
    {"item": "Pen", "price": 20},
    {"item": "Notebook", "price": 50}
]

total_price = reduce(lambda acc, item: acc + item["price"], cart, 0)
print(total_price)  # Output: 170
```

```
170
```

8. Hitung Frekuensi Huruf

[22]:
```
from collections import Counter

text = "functional programming"

frequency = reduce(lambda acc, char: acc + Counter(char), text, Counter())
print(frequency)
# Output: Counter({' ': 1, 'g': 2, 'r': 2, ...})
```

```
Counter({'n': 3, 'i': 2, 'o': 2, 'a': 2, 'r': 2, 'g': 2, 'm': 2, 'f': 1, 'u': 1,
'c': 1, 't': 1, 'l': 1, ' ': 1, 'p': 1})
```

9. Membuat Dictionary Reduksi dari List

[23]:
```
pairs = [("a", 1), ("b", 2), ("c", 3)]

result = reduce(lambda acc, pair: {**acc, pair[0]: pair[1]}, pairs, {})
print(result)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

```
{'a': 1, 'b': 2, 'c': 3}
```

Gunakan reduce() jika: - Kamu ingin menggabungkan nilai-nilai menjadi satu. - Operasinya bisa diwakili dalam bentuk akumulator. - Operasinya sederhana dan tidak perlu banyak kondisi. - Kamu ingin menghindari loop eksplisit untuk kejelasan kode.

1.3.5 Latihan Reduksi

1. Diberikan array berisi harga barang, hitung total semua harga.

[ ]:
```
prices = [10000, 15000, 20000, 5000]

# Output yang diharapkan: 50000
# Hint: Gunakan reduce() dengan penjumlahan.
```

[24]:
```
prices = [10000, 15000, 20000, 5000]
total = reduce(lambda acc, x: acc + x, prices)
print("Total harga:", total)  # Output: 50000
```

```
Total harga: 50000
```

2. Gabungkan semua string dalam list menjadi satu kalimat.

[ ]:
```
words = ["Data", "science", "is", "awesome"]

# Output yang diharapkan: "Data science is awesome"
# Hint: Gunakan reduce() dengan penggabungan string.
```

[25]:
```
words = ["Data", "science", "is", "awesome"]
sentence = reduce(lambda acc, word: acc + " " + word, words)
print("Kalimat:", sentence)  # Output: "Data science is awesome"
```

```
Kalimat: Data science is awesome
```

3. Hitung total jumlah huruf dari semua kata dalam list (tanpa spasi).

[ ]:
```
words = ["machine", "learning", "python"]

# Output yang diharapkan: 21
# Hint: Gunakan len() dalam fungsi reduce.
```

[26]:
```
words = ["machine", "learning", "python"]
total_chars = reduce(lambda acc, word: acc + len(word), words, 0)
print("Jumlah karakter:", total_chars)  # Output: 21
```

```
Jumlah karakter: 21
```

4. Gabungkan list of dictionary menjadi satu dictionary. Jika ada kunci yang sama, simpan nilai terakhir.

[ ]:
```
dicts = [{"a": 1}, {"b": 2}, {"a": 3}, {"c": 4}]

# Output yang diharapkan: {"a": 3, "b": 2, "c": 4}
# Hint: Gunakan unpacking {**acc, **x} atau update pada dict.
```

[27]:
```
dicts = [{"a": 1}, {"b": 2}, {"a": 3}, {"c": 4}]
combined = reduce(lambda acc, d: {**acc, **d}, dicts)
print("Hasil merge dict:", combined)  # Output: {'a': 3, 'b': 2, 'c': 4}
```

```
Hasil merge dict: {'a': 3, 'b': 2, 'c': 4}
```

[28]:
```
# Pakai Update

def merge_dicts(acc, d):
    acc.update(d)
    return acc

combined = reduce(merge_dicts, dicts, {})
print("Hasil merge dict:", combined)
```

```
Hasil merge dict: {'a': 3, 'b': 2, 'c': 4}
```

5. Buat dictionary yang menunjukkan jumlah kemunculan tiap huruf dari sebuah kalimat (abaikan spasi dan kapitalisasi).

Hint: - Ubah ke lowercase - Hapus spasi - Gunakan reduce() dan dict.get(char, 0) untuk akumulasi

[ ]:
```
sentence = "Data Science"

# Output yang diharapkan: {'d': 1, 'a': 2, 't': 1, 's': 1, 'c': 2, 'i': 1, 'e':␣
↪2, 'n': 1}
```

[29]:
```
sentence = "Data Science"
clean = sentence.replace(" ", "").lower()

def count_chars(acc, char):
    acc[char] = acc.get(char, 0) + 1
    return acc

frequencies = reduce(count_chars, clean, {})
print("Frekuensi huruf:", frequencies)
# Output: {'d': 1, 'a': 2, 't': 1, 's': 1, 'c': 2, 'i': 1, 'e': 2, 'n': 1}
```

```
Frekuensi huruf: {'d': 1, 'a': 2, 't': 1, 's': 1, 'c': 2, 'i': 1, 'e': 2, 'n':
1}
```

versi collections.Counter() Hitung Frekuensi Huruf dalam Kalimat (dengan Counter)

[30]:
```
sentence = "Data Science"

from collections import Counter
# Buang spasi dan ubah ke huruf kecil
clean = sentence.replace(" ", "").lower()

# Gunakan Counter
frequencies = Counter(clean)

print("Frekuensi huruf:", frequencies)
```

```
Frekuensi huruf: Counter({'a': 2, 'c': 2, 'e': 2, 'd': 1, 't': 1, 's': 1, 'i':
1, 'n': 1})
```

Frekuensi kata

[31]:
```
text = "data science is fun and data is powerful"
words = text.split()
frequencies = Counter(words)

print("Frekuensi kata:", frequencies)
```

```
Frekuensi kata: Counter({'data': 2, 'is': 2, 'science': 1, 'fun': 1, 'and': 1,
'powerful': 1})
```

Frekuensi Huruf dari Banyak Kalimat

[32]:
```
sentences = ["Machine Learning", "Data Science", "Deep Learning"]

# Gabung semua kalimat, buang spasi, dan ubah ke huruf kecil
all_text = ''.join(sentences).replace(" ", "").lower()

frequencies = Counter(all_text)
print("Frekuensi huruf:", frequencies)
```

```
Frekuensi huruf: Counter({'e': 7, 'n': 6, 'a': 5, 'i': 4, 'c': 3, 'l': 2, 'r':
2, 'g': 2, 'd': 2, 'm': 1, 'h': 1, 't': 1, 's': 1, 'p': 1})
```