**Modul Algoritma dan Struktur Data dengan Python**

****

**Oleh:**

**Muhammad Ari Rifqi, S. Kom., M. Kom.**

**Bab 1: Pendahuluan**

**1.1 Pengantar Algoritma dan Struktur Data**

**1. Definisi Algoritma**

**Algoritma** adalah sekumpulan langkah-langkah logis yang digunakan untuk menyelesaikan suatu masalah secara sistematis dan efisien.

**Karakteristik algoritma:**

1. **Input** – Memiliki nol atau lebih masukan.
2. **Output** – Menghasilkan setidaknya satu keluaran.
3. **Definiteness** – Setiap langkah jelas dan tidak ambigu.
4. **Finiteness** – Algoritma memiliki akhir yang jelas.
5. **Effectiveness** – Langkah-langkahnya dapat dilakukan dalam waktu yang wajar.

**Contoh algoritma sederhana:**  
Mencari nilai terbesar dari dua bilangan:

1. Masukkan dua bilangan: a dan b.
2. Bandingkan a dan b.
3. Jika a > b, maka a adalah bilangan terbesar.
4. Jika b > a, maka b adalah bilangan terbesar.
5. Jika a == b, maka kedua bilangan sama besar.

**Implementasi dalam Python:**

python

CopyEdit

def cari\_nilai\_terbesar(a, b):

if a > b:

return a

elif b > a:

return b

else:

return "Kedua bilangan sama"

# Contoh penggunaan

print(cari\_nilai\_terbesar(10, 20))

**2. Definisi Struktur Data**

**Struktur Data** adalah cara mengorganisir dan menyimpan data agar dapat diakses dan dimodifikasi secara efisien.

**Jenis-Jenis Struktur Data**

1. **Struktur Data Linear**
   * **Array** → Kumpulan elemen dengan indeks tetap.
   * **Linked List** → Kumpulan node yang saling terhubung.
   * **Stack** → Struktur LIFO (Last In, First Out).
   * **Queue** → Struktur FIFO (First In, First Out).
2. **Struktur Data Non-Linear**
   * **Tree** → Struktur hierarkis dengan root dan cabang.
   * **Graph** → Kumpulan simpul (node) dan sisi (edge) yang terhubung.

**Contoh struktur data list dalam Python:**

python

CopyEdit

# Contoh List dalam Python

data = [10, 20, 30, 40]

print("Elemen pertama:", data[0]) # Mengakses elemen pertama

**3. Hubungan antara Algoritma dan Struktur Data**

Algoritma dan struktur data saling berkaitan dalam menyelesaikan masalah pemrograman.

|  |  |  |
| --- | --- | --- |
| **Aspek** | **Algoritma** | **Struktur Data** |
| **Definisi** | Langkah-langkah penyelesaian masalah | Cara menyimpan dan mengatur data |
| **Tujuan** | Mengoptimalkan proses komputasi | Memudahkan pengelolaan data |
| **Contoh** | Sorting, searching, graph traversal | Stack, queue, tree, graph |

**Contoh hubungan algoritma dan struktur data:**

* **Sorting (pengurutan)** menggunakan **Array atau List**.
* **Pencarian rute (Pathfinding)** menggunakan **Graph**.
* **Manajemen Undo/Redo** menggunakan **Stack**.

**Contoh implementasi algoritma pencarian menggunakan struktur data List (Linear Search):**

python

CopyEdit

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i # Mengembalikan indeks elemen yang ditemukan

return -1 # Jika tidak ditemukan

# Contoh penggunaan

data = [10, 20, 30, 40, 50]

target = 30

hasil = linear\_search(data, target)

if hasil != -1:

print(f"Elemen ditemukan di indeks {hasil}")

else:

print("Elemen tidak ditemukan")

**4. Peran Algoritma dalam Pemrograman**

Algoritma memiliki peran penting dalam pemrograman, yaitu:

1. **Efisiensi** – Mengurangi waktu eksekusi dan penggunaan memori.
2. **Optimasi** – Meningkatkan kinerja aplikasi, misalnya sorting yang lebih cepat.
3. **Keamanan** – Digunakan dalam enkripsi data dan keamanan sistem.
4. **Pemecahan Masalah** – Memudahkan penyelesaian masalah kompleks seperti AI dan Big Data.

**Contoh implementasi peran algoritma dalam sorting (Bubble Sort):**

python

CopyEdit

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]: # Tukar jika elemen sebelumnya lebih besar

arr[j], arr[j+1] = arr[j+1], arr[j]

# Contoh penggunaan

data = [64, 34, 25, 12, 22, 11, 90]

bubble\_sort(data)

print("Data setelah sorting:", data)

**Kesimpulan**

1. **Algoritma** adalah serangkaian langkah-langkah yang digunakan untuk menyelesaikan masalah.
2. **Struktur data** adalah cara penyimpanan dan pengorganisasian data.
3. **Algoritma dan struktur data saling berkaitan** dalam membangun solusi pemrograman yang efisien.
4. **Pemilihan algoritma dan struktur data yang tepat** sangat penting untuk meningkatkan kinerja program.

**Latihan**

1. **Buat algoritma** untuk menghitung jumlah bilangan dalam sebuah list menggunakan loop.
2. **Implementasikan algoritma rekursif** untuk menghitung faktorial suatu bilangan.
3. **Buat algoritma pencarian biner** untuk menemukan elemen dalam array yang sudah diurutkan.

**Selamat belajar! 🚀**

**1.2 Pengantar Python untuk Algoritma dan Struktur Data**

**1. Dasar-Dasar Python**

**1.1 Variabel**

Variabel digunakan untuk menyimpan data dalam program. Python tidak memerlukan deklarasi tipe data eksplisit.

**Contoh:**

python

CopyEdit

# Deklarasi variabel

nama = "Andi" # String

usia = 20 # Integer

tinggi = 1.75 # Float

is\_student = True # Boolean

# Menampilkan variabel

print("Nama:", nama)

print("Usia:", usia)

print("Tinggi:", tinggi)

print("Mahasiswa:", is\_student)

**1.2 Tipe Data dalam Python**

Python memiliki beberapa tipe data utama:

|  |  |
| --- | --- |
| **Tipe Data** | **Contoh** |
| **Integer** (bilangan bulat) | 10, -5, 100 |
| **Float** (bilangan desimal) | 3.14, -2.5, 0.99 |
| **String** (teks) | "Halo", 'Python' |
| **Boolean** (true/false) | True, False |
| **List** (kumpulan data) | [1, 2, 3] |
| **Tuple** (seperti list tapi immutable) | (1, 2, 3) |
| **Dictionary** (pasangan kunci-nilai) | {"nama": "Budi", "umur": 20} |

**Contoh:**

python

CopyEdit

# List

angka = [10, 20, 30]

print("Elemen pertama:", angka[0])

# Dictionary

data\_mahasiswa = {"nama": "Budi", "umur": 20}

print("Nama mahasiswa:", data\_mahasiswa["nama"])

**1.3 Operator dalam Python**

Python memiliki beberapa jenis operator:

1. **Operator Aritmatika** (+, -, \*, /, //, %, \*\*)
2. **Operator Perbandingan** (==, !=, >, <, >=, <=)
3. **Operator Logika** (and, or, not)

**Contoh penggunaan operator:**

python

CopyEdit

a = 10

b = 3

# Operator Aritmatika

print("Penjumlahan:", a + b)

print("Modulus:", a % b)

print("Pangkat:", a \*\* b)

# Operator Perbandingan

print("Apakah a lebih besar dari b?", a > b)

# Operator Logika

x = True

y = False

print("Hasil AND:", x and y)

print("Hasil OR:", x or y)

**2. Struktur Kontrol dalam Python**

**2.1 Percabangan (if, elif, else)**

Percabangan digunakan untuk mengambil keputusan berdasarkan kondisi tertentu.

**Sintaks:**

python

CopyEdit

if kondisi:

perintah

elif kondisi\_lain:

perintah\_lain

else:

perintah\_default

**Contoh:**

python

CopyEdit

nilai = 85

if nilai >= 90:

print("A")

elif nilai >= 80:

print("B")

elif nilai >= 70:

print("C")

else:

print("D")

**2.2 Perulangan (for dan while)**

Perulangan digunakan untuk mengulang eksekusi kode beberapa kali.

**Perulangan for**

Digunakan untuk mengulang dalam rentang atau koleksi data.

python

CopyEdit

for i in range(5): # Mengulang dari 0 hingga 4

print("Iterasi ke-", i)

**Perulangan while**

Digunakan untuk mengulang selama kondisi masih **True**.

python

CopyEdit

x = 0

while x < 5:

print("Angka:", x)

x += 1 # Increment

**Loop dengan break dan continue**

* **break** → Menghentikan loop.
* **continue** → Melewati iterasi saat ini dan lanjut ke iterasi berikutnya.

python

CopyEdit

for i in range(10):

if i == 5:

break # Menghentikan loop saat i == 5

print(i)

**3. Fungsi dan Rekursi**

**3.1 Fungsi dalam Python**

Fungsi digunakan untuk membagi program menjadi bagian-bagian kecil agar lebih mudah dipahami dan digunakan kembali.

**Sintaks:**

python

CopyEdit

def nama\_fungsi(parameter):

perintah

return hasil

**Contoh:**

python

CopyEdit

def tambah(a, b):

return a + b

hasil = tambah(10, 5)

print("Hasil penjumlahan:", hasil)

**3.2 Fungsi Rekursif**

Rekursi adalah teknik di mana sebuah fungsi memanggil dirinya sendiri.

**Contoh: Menghitung Faktorial dengan Rekursi**

python

CopyEdit

def faktorial(n):

if n == 1:

return 1

else:

return n \* faktorial(n-1)

print("Faktorial 5:", faktorial(5))

**Kesimpulan**

1. Python memiliki **sintaks sederhana** untuk memudahkan pemrograman algoritma dan struktur data.
2. **Variabel dan tipe data** adalah dasar dalam menyimpan dan memanipulasi informasi.
3. **Struktur kontrol** seperti **percabangan dan perulangan** memungkinkan eksekusi kode yang fleksibel.
4. **Fungsi dan rekursi** digunakan untuk modularisasi program dan menyelesaikan masalah kompleks.

**Latihan**

1. Buat fungsi yang menerima list angka dan mengembalikan jumlah totalnya.
2. Implementasikan algoritma pencarian bilangan terbesar dalam list menggunakan perulangan.
3. Buat fungsi rekursif untuk menghitung deret Fibonacci hingga elemen ke-n.

**Selamat belajar! 🚀**

**Latihan**

1. Buatlah program Python sederhana yang menerima dua bilangan dan menampilkan hasil penjumlahan, pengurangan, perkalian, dan pembagian.
2. Buat fungsi rekursif untuk menghitung faktorial dari suatu bilangan.

**Bab 2: Tipe Data Abstrak (ADT)**

**2.1 Definisi dan Konsep ADT**

**1. Stack (Tumpukan)**

* **Definisi**: Stack adalah struktur data linier yang mengikuti prinsip **LIFO (Last-In, First-Out)**. Elemen yang terakhir dimasukkan akan menjadi elemen yang pertama keluar.
* **Konsep**: Stack dapat dianalogikan seperti tumpukan piring. Piring yang terakhir diletakkan di atas tumpukan akan menjadi piring yang pertama diambil.
* **Operasi Dasar**:\*
  + **Push**: Menambahkan elemen ke puncak stack.
  + **Pop**: Menghapus elemen dari puncak stack.
  + **Peek/Top**: Mengintip elemen yang berada di puncak stack tanpa menghapusnya.
  + **IsEmpty**: Memeriksa apakah stack kosong.
  + **IsFull**: Memeriksa apakah stack penuh (jika ada batasan ukuran).
* **Contoh**:\*
  + **Notasi Infix ke Postfix**: Stack digunakan dalam konversi notasi matematika dari infix (misalnya, 2 + 3 \* 4) ke postfix (misalnya, 2 3 4 \* +).
  + **Undo/Redo**: Stack digunakan untuk menyimpan riwayat perubahan dalam aplikasi, sehingga pengguna dapat melakukan undo atau redo.
  + **Pemanggilan Fungsi**: Stack digunakan untuk menyimpan informasi tentang fungsi-fungsi yang sedang aktif dipanggil dalam program.

**2. Queue (Antrian)**

* **Definisi**: Queue adalah struktur data linier yang mengikuti prinsip **FIFO (First-In, First-Out)**. Elemen yang pertama dimasukkan akan menjadi elemen yang pertama keluar.
* **Konsep**: Queue dapat dianalogikan seperti antrian orang di loket. Orang yang pertama datang akan menjadi orang yang pertama dilayani.
* **Operasi Dasar**:\*
  + **Enqueue**: Menambahkan elemen ke belakang antrian.
  + **Dequeue**: Menghapus elemen dari depan antrian.
  + **Front**: Mengintip elemen yang berada di depan antrian tanpa menghapusnya.
  + **Rear**: Mengintip elemen yang berada di belakang antrian tanpa menghapusnya.
  + **IsEmpty**: Memeriksa apakah antrian kosong.
  + **IsFull**: Memeriksa apakah antrian penuh (jika ada batasan ukuran).
* **Contoh**:\*
  + **Antrian Printer**: Dokumen yang dikirim untuk dicetak akan dimasukkan ke dalam antrian printer.
  + **Antrian Layanan Pelanggan**: Panggilan telepon ke pusat layanan pelanggan akan diatur dalam antrian.
  + **Breadth-First Search (BFS)**: Queue digunakan dalam algoritma BFS untuk menelusuri graf atau pohon.

**3. Linked List (Daftar Bertaut)**

* **Definisi**: Linked list adalah struktur data yang terdiri dari kumpulan node (simpul) yang saling terhubung. Setiap node berisi data dan pointer (penunjuk) ke node berikutnya dalam daftar.
* **Konsep**: Linked list berbeda dengan array karena elemen-elemennya tidak harus disimpan secara berurutan dalam memori. Setiap node dapat dialokasikan secara dinamis.
* **Jenis-jenis Linked List**:\*
  + **Singly Linked List**: Setiap node memiliki satu pointer ke node berikutnya.
  + **Doubly Linked List**: Setiap node memiliki dua pointer, satu ke node berikutnya dan satu ke node sebelumnya.
  + **Circular Linked List**: Node terakhir menunjuk kembali ke node pertama, membentuk lingkaran.
* **Operasi Dasar**:\*
  + **Insert**: Menambahkan node baru ke dalam daftar.
  + **Delete**: Menghapus node dari daftar.
  + **Search**: Mencari node dengan data tertentu dalam daftar.
  + **Traverse**: Mengunjungi setiap node dalam daftar secara berurutan.
* **Contoh**:\*
  + **Implementasi Stack dan Queue**: Linked list dapat digunakan untuk mengimplementasikan stack dan queue.
  + **Manajemen Memori Dinamis**: Linked list dapat digunakan untuk mengelola alokasi memori dinamis dalam program.
  + **Representasi Polinomial**: Linked list dapat digunakan untuk merepresentasikan polinomial dalam matematika.

**4. Tree (Pohon)**

* **Definisi**: Tree adalah struktur data hierarkis yang terdiri dari node-node yang saling terhubung. Setiap node memiliki satu parent (induk) (kecuali root) dan nol atau lebih child (anak).
* **Konsep**: Tree sering digunakan untuk merepresentasikan data yang memiliki hubungan hierarkis, seperti struktur organisasi, pohon keluarga, atau sistem file.
* **Jenis-jenis Tree**:\*
  + **Binary Tree**: Setiap node memiliki paling banyak dua anak (kiri dan kanan).
  + **Binary Search Tree (BST)**: Binary tree yang diurutkan berdasarkan nilai.
  + **Balanced Tree**: Binary tree yang tinggi sub-sub pohonnya seimbang.
* **Operasi Dasar**:\*
  + **Insert**: Menambahkan node baru ke dalam tree.
  + **Delete**: Menghapus node dari tree.
  + **Search**: Mencari node dengan data tertentu dalam tree.
  + **Traverse**: Mengunjungi setiap node dalam tree (inorder, preorder, postorder).
* **Contoh**:\*
  + **Sistem File**: Direktori dan file dalam sistem operasi diorganisasikan dalam bentuk tree.
  + **Algoritma Pencarian**: Binary search tree digunakan dalam algoritma pencarian yang efisien.
  + **Pengambilan Keputusan**: Tree digunakan dalam kecerdasan buatan untuk merepresentasikan aturan pengambilan keputusan.

**5. Graph (Graf)**

* **Definisi**: Graph adalah struktur data yang terdiri dari kumpulan vertex (simpul) dan edge (rusuk) yang menghubungkan antar vertex.
* **Konsep**: Graph digunakan untuk merepresentasikan hubungan antara objek-objek. Edge dapat berarah (directed) atau tidak berarah (undirected).
* **Jenis-jenis Graph**:\*
  + **Directed Graph (Digraph)**: Edge memiliki arah (misalnya, jalan satu arah).
  + **Undirected Graph**: Edge tidak memiliki arah (misalnya, pertemanan).
  + **Weighted Graph**: Edge memiliki bobot (misalnya, jarak antar kota).
* **Representasi Graph**:\*
  + **Adjacency Matrix**: Matriks yang merepresentasikan hubungan antar vertex.
  + **Adjacency List**: Daftar yang berisi tetangga dari setiap vertex.
* **Algoritma Graph**:\*
  + **Shortest Path**: Mencari jalur terpendek antar dua vertex (misalnya, Dijkstra's algorithm).
  + **Spanning Tree**: Mencari pohon yang mencakup semua vertex dalam graf (misalnya, Prim's algorithm).
* **Contoh**:\*
  + **Peta**: Jalan dan kota dalam peta dapat direpresentasikan sebagai graf.
  + **Jaringan Sosial**: Pertemanan antar pengguna dalam jaringan sosial dapat direpresentasikan sebagai graf.
  + **Internet**: Halaman web dan link antar halaman web dapat direpresentasikan sebagai graf.

**2.2 Implementasi Stack di Python**

class Stack:

def \_\_init\_\_(self):

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

return self.items.pop() if not self.is\_empty() else None

def is\_empty(self):

return len(self.items) == 0

**Latihan**

1. Buat program untuk menampilkan elemen stack dari atas ke bawah.
2. Implementasikan Stack menggunakan **deque** dari modul collections.

**Bab 3: Algoritma Pencarian dan Pengurutan**

**3.1 Algoritma Pencarian**

Mari kita bahas mengenai Linear Search dan Binary Search, dua algoritma pencarian yang fundamental dalam ilmu komputer.

**1. Linear Search (Pencarian Linear)**

* **Konsep**: Linear search adalah algoritma pencarian paling sederhana. Ia bekerja dengan cara memeriksa setiap elemen dalam daftar satu per satu, mulai dari awal hingga akhir, sampai elemen yang dicari ditemukan atau seluruh daftar telah diperiksa.
* **Cara Kerja**:
  1. Mulai dari elemen pertama dalam daftar.
  2. Bandingkan elemen saat ini dengan nilai yang dicari.
  3. Jika sama, pencarian selesai. Kembalikan indeks elemen saat ini.
  4. Jika tidak sama, lanjutkan ke elemen berikutnya.
  5. Ulangi langkah 2-4 sampai elemen yang dicari ditemukan atau mencapai akhir daftar.
  6. Jika mencapai akhir daftar tanpa menemukan elemen yang dicari, kembalikan nilai yang menandakan "tidak ditemukan" (biasanya -1).
* **Contoh**: Misalnya, kita punya daftar angka: [5, 2, 9, 1, 5, 6] dan kita ingin mencari angka 9.
  1. Kita mulai dari 5. 5 != 9.
  2. Kita pindah ke 2. 2 != 9.
  3. Kita pindah ke 9. 9 == 9. Pencarian selesai. Kita kembalikan indeks 2.
* **Kelebihan**:
  1. Mudah diimplementasikan.
  2. Bekerja pada daftar yang tidak terurut.
* **Kekurangan**:
  1. Tidak efisien untuk daftar yang besar. Waktu pencarian terburuk adalah O(n), di mana n adalah jumlah elemen dalam daftar. Ini berarti, dalam kasus terburuk (elemen yang dicari berada di akhir daftar atau tidak ada), algoritma harus memeriksa semua elemen.

**2. Binary Search (Pencarian Biner)**

* **Konsep**: Binary search jauh lebih efisien daripada linear search, tetapi *hanya bekerja pada daftar yang sudah terurut*. Ia bekerja dengan berulang kali membagi dua bagian daftar yang sedang dicari.
* **Cara Kerja**:
  1. Pastikan daftar sudah terurut.
  2. Tentukan bagian tengah daftar.
  3. Bandingkan elemen tengah dengan nilai yang dicari.
  4. Jika sama, pencarian selesai. Kembalikan indeks elemen tengah.
  5. Jika nilai yang dicari lebih kecil dari elemen tengah, cari di setengah bagian kiri daftar.
  6. Jika nilai yang dicari lebih besar dari elemen tengah, cari di setengah bagian kanan daftar.
  7. Ulangi langkah 2-6 sampai elemen yang dicari ditemukan atau bagian daftar yang diperiksa kosong.
* **Contoh**: Misalnya, kita punya daftar angka terurut: [1, 2, 5, 6, 9, 15] dan kita ingin mencari angka 9.
  1. Tengah daftar adalah 6. 9 > 6.
  2. Kita cari di setengah bagian kanan: [9, 15].
  3. Tengah dari bagian kanan adalah 9. 9 == 9. Pencarian selesai. Kita kembalikan indeks 4.
* **Kelebihan**:
  1. Sangat efisien untuk daftar yang besar. Waktu pencarian terburuk adalah O(log n). Ini berarti, untuk daftar dengan jutaan elemen, binary search dapat menemukan elemen yang dicari hanya dalam beberapa langkah.
* **Kekurangan**:
  1. Membutuhkan daftar yang terurut. Jika daftar tidak terurut, kita perlu mengurutkannya terlebih dahulu, yang membutuhkan waktu.
  2. Lebih kompleks untuk diimplementasikan daripada linear search.

**Perbandingan**

|  |  |  |
| --- | --- | --- |
| **Fitur** | **Linear Search** | **Binary Search** |
| Kondisi Data | Tidak harus terurut | Harus terurut |
| Kompleksitas Waktu Terburuk | O(n) | O(log n) |
| Implementasi | Sederhana | Lebih kompleks |

Ekspor ke Spreadsheet

**Kapan Menggunakan?**

* **Linear Search**: Digunakan ketika daftar kecil atau tidak terurut.
* **Binary Search**: Digunakan ketika daftar besar dan sudah terurut. Jika Anda sering melakukan pencarian pada data yang sama, mengurutkan data sekali dan kemudian menggunakan binary search akan jauh lebih efisien daripada selalu menggunakan linear search.

**3.2 Algoritma Pengurutan**

**1. Bubble Sort (Pengurutan Gelembung)**

* **Konsep**: Bubble sort adalah algoritma pengurutan sederhana yang berulang kali membandingkan pasangan elemen yang berdekatan dan menukar posisinya jika urutannya salah. Proses ini diulangi sampai tidak ada lagi penukaran yang diperlukan, yang berarti daftar sudah terurut. Nama "bubble" berasal dari cara elemen-elemen besar "mengapung" ke akhir daftar seperti gelembung.
* **Cara Kerja**:
  1. Mulai dari awal daftar.
  2. Bandingkan dua elemen yang berurutan.
  3. Jika elemen pertama lebih besar dari elemen kedua (untuk pengurutan naik), tukar posisinya.
  4. Pindah ke pasangan elemen berikutnya.
  5. Ulangi langkah 2-4 sampai akhir daftar.
  6. Ulangi langkah 1-5 sampai tidak ada lagi penukaran dalam satu iterasi penuh.
* **Contoh**: Misalnya, kita ingin mengurutkan daftar [5, 1, 4, 2, 8] secara naik.
  1. Iterasi 1: [1, 5, 4, 2, 8] -> [1, 4, 5, 2, 8] -> [1, 4, 2, 5, 8] -> [1, 4, 2, 5, 8] (tidak ada perubahan karena 8 sudah paling besar)
  2. Iterasi 2: [1, 4, 2, 5, 8] -> [1, 4, 2, 5, 8] -> [1, 2, 4, 5, 8] -> [1, 2, 4, 5, 8]
  3. Iterasi 3: [1, 2, 4, 5, 8] (tidak ada perubahan) Daftar sekarang terurut: [1, 2, 4, 5, 8]
* **Kelebihan**:
  1. Mudah diimplementasikan dan dipahami.
* **Kekurangan**:
  1. Tidak efisien untuk daftar yang besar. Kompleksitas waktu terburuk dan rata-rata adalah O(n^2). Ini berarti waktu pengurutan meningkat secara kuadratik dengan bertambahnya jumlah elemen.

**2. Merge Sort (Pengurutan Gabung)**

* **Konsep**: Merge sort adalah algoritma pengurutan *divide and conquer*. Ia bekerja dengan cara membagi daftar menjadi sub-sub daftar yang lebih kecil, mengurutkan masing-masing sub-daftar, dan kemudian menggabungkan sub-sub daftar yang terurut kembali menjadi satu daftar yang terurut.
* **Cara Kerja**:
  1. Bagi daftar menjadi dua bagian sama besar.
  2. Urutkan masing-masing bagian secara rekursif menggunakan merge sort.
  3. Gabungkan dua bagian yang terurut menjadi satu daftar yang terurut. Proses penggabungan ini dilakukan dengan membandingkan elemen pertama dari kedua bagian dan memilih yang lebih kecil untuk dimasukkan ke dalam daftar hasil penggabungan.
* **Contoh**: Mengurutkan [5, 1, 4, 2, 8]
  1. Bagi: [5, 1, 4] dan [2, 8]
  2. Bagi lagi: [5, 1] dan [4] dan [2] dan [8]
  3. Urutkan (basis rekursi): [1, 5], [4], [2], [8]
  4. Gabung: [1, 4, 5] dan [2, 8]
  5. Gabung lagi: [1, 2, 4, 5, 8]
* **Kelebihan**:
  1. Efisien untuk daftar yang besar. Kompleksitas waktu terburuk, rata-rata, dan terbaik adalah O(n log n).
  2. Stabil (mempertahankan urutan elemen-elemen yang sama).
* **Kekurangan**:
  1. Membutuhkan ruang memori tambahan untuk proses penggabungan.

**3. Quick Sort (Pengurutan Cepat)**

* **Konsep**: Quick sort juga merupakan algoritma pengurutan *divide and conquer*. Ia memilih sebuah elemen sebagai *pivot* dan kemudian mempartisi daftar sehingga semua elemen yang lebih kecil dari pivot berada di sebelah kiri pivot dan semua elemen yang lebih besar dari pivot berada di sebelah kanan pivot. Proses ini diulangi secara rekursif untuk sub-sub daftar.
* **Cara Kerja**:
  1. Pilih sebuah pivot (biasanya elemen pertama, tengah, atau acak).
  2. Partisi daftar: Pindahkan semua elemen yang lebih kecil dari pivot ke sebelah kiri pivot dan semua elemen yang lebih besar ke sebelah kanan pivot.
  3. Urutkan sub-daftar di sebelah kiri pivot dan sub-daftar di sebelah kanan pivot secara rekursif menggunakan quick sort.
* **Contoh**: Mengurutkan [5, 1, 4, 2, 8] (pivot = 5)
  1. Partisi: [1, 4, 2] [5] [8]
  2. Urutkan [1, 4, 2] (misalnya, pivot = 1): [1] [4, 2]
  3. Urutkan [4, 2] (misalnya, pivot = 4): [2] [4]
  4. Gabung: [1, 2, 4, 5, 8]
* **Kelebihan**:
  1. Sangat efisien dalam praktiknya. Kompleksitas waktu rata-rata adalah O(n log n).
  2. Tidak membutuhkan ruang memori tambahan sebanyak merge sort (in-place).
* **Kekurangan**:
  1. Kompleksitas waktu terburuk adalah O(n^2) (terjadi jika pivot selalu elemen terkecil atau terbesar). Namun, ini jarang terjadi dalam praktiknya.
  2. Tidak stabil.

**Perbandingan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Fitur** | **Bubble Sort** | **Merge Sort** | **Quick Sort** |
| Kompleksitas Waktu Terburuk | O(n^2) | O(n log n) | O(n^2) |
| Kompleksitas Waktu Rata-rata | O(n^2) | O(n log n) | O(n log n) |
| Kompleksitas Waktu Terbaik | O(n) | O(n log n) | O(n log n) |
| Memori Tambahan | O(1) | O(n) | O(log n) |
| Stabil | Ya | Ya | Tidak |

Ekspor ke Spreadsheet

**Kapan Menggunakan?**

* **Bubble Sort**: Untuk daftar yang sangat kecil atau untuk tujuan pembelajaran.
* **Merge Sort**: Untuk daftar yang besar dan ketika stabilitas penting.
* **Quick Sort**: Untuk sebagian besar kasus pengurutan karena efisiensinya yang baik dalam praktiknya. Perhatikan risiko kasus terburuk O(n^2), tetapi ini dapat dihindari dengan pemilihan pivot yang baik (misalnya, pivot acak).

# Contoh Binary Search di Python

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

**Latihan**

1. Implementasikan Bubble Sort dan analisis kompleksitasnya.
2. Bandingkan Merge Sort dan Quick Sort dalam hal efisiensi waktu.

**Bab 4: Teknik Pemrograman Dinamis**

**Prinsip Dasar Dynamic Programming**

Dynamic Programming (DP) adalah teknik pemecahan masalah dengan cara memecah masalah yang kompleks menjadi submasalah yang lebih kecil dan sederhana, menyelesaikannya sekali, dan menyimpan hasilnya untuk digunakan kembali di kemudian hari. Dengan demikian, DP menghindari perhitungan ulang yang tidak perlu dan meningkatkan efisiensi.

**Dua Karakteristik Utama DP**

1. **Overlapping Subproblems (Submasalah yang Tumpang Tindih)**: Masalah besar dapat dipecah menjadi submasalah yang lebih kecil, dan beberapa submasalah ini mungkin tumpang tindih atau muncul berulang kali dalam proses pemecahan masalah. DP memanfaatkan fakta ini dengan menyimpan solusi dari submasalah yang telah dipecahkan sehingga dapat digunakan kembali jika diperlukan, alih-alih menghitungnya lagi.
2. **Optimal Substructure (Struktur Optimal)**: Solusi optimal untuk masalah yang kompleks dapat dibangun dari solusi optimal untuk submasalahnya. Dengan kata lain, jika kita telah menemukan solusi optimal untuk submasalah, kita dapat menggunakannya untuk membangun solusi optimal untuk masalah yang lebih besar.

**Dua Pendekatan Utama DP**

1. **Top-down (Memoization)**: Pendekatan ini dimulai dari masalah utama dan memecahnya menjadi submasalah yang lebih kecil. Solusi untuk setiap submasalah disimpan dalam "memo" (biasanya berupa array atau tabel). Jika submasalah yang sama muncul lagi, kita cukup mengambil solusinya dari memo, bukan menghitungnya lagi.
2. **Bottom-up (Tabulation)**: Pendekatan ini dimulai dari submasalah terkecil dan membangun solusi untuk submasalah yang lebih besar secara bertahap. Solusi untuk setiap submasalah disimpan dalam tabel. Pendekatan ini biasanya lebih efisien daripada pendekatan top-down karena tidak melibatkan rekursi.

**Contoh Kasus: Fibonacci dengan DP**

Bilangan Fibonacci adalah urutan bilangan di mana setiap bilangan adalah jumlah dari dua bilangan sebelumnya. Dua bilangan pertama dalam urutan Fibonacci adalah 0 dan 1.

**Contoh**

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

**Perhitungan Fibonacci Naif (Rekursif)**

Python

def fibonacci\_naive(n):

if n <= 1:

return n

else:

return fibonacci\_naive(n-1) + fibonacci\_naive(n-2)

**Masalah**

Pendekatan rekursif naif di atas sangat tidak efisien karena banyak perhitungan yang diulang. Misalnya, untuk menghitung fibonacci\_naive(5), kita perlu menghitung fibonacci\_naive(4) dan fibonacci\_naive(3). Namun, untuk menghitung fibonacci\_naive(4), kita juga perlu menghitung fibonacci\_naive(3) lagi, dan seterusnya. Hal ini menyebabkan banyak perhitungan yang tumpang tindih.

**Solusi dengan DP**

**1. Top-down (Memoization)**

Python

memo = {}

def fibonacci\_memo(n):

if n in memo:

return memo[n]

if n <= 1:

result = n

else:

result = fibonacci\_memo(n-1) + fibonacci\_memo(n-2)

memo[n] = result

return result

**2. Bottom-up (Tabulation)**

Python

def fibonacci\_tab(n):

tab = [0] \* (n + 1)

tab[0] = 0

tab[1] = 1

for i in range(2, n + 1):

tab[i] = tab[i-1] + tab[i-2]

return tab[n]

**Perbandingan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Fitur** | **Fibonacci Naif** | **Fibonacci Memo** | **Fibonacci Tab** |
| Kompleksitas Waktu | Eksponensial | Linear | Linear |
| Kompleksitas Ruang | Tergantung kedalaman rekursi | Linear | Linear |
| Efisiensi | Sangat tidak efisien | Efisien | Paling efisien |

**Kesimpulan**

Dynamic Programming adalah teknik yang sangat berguna untuk memecahkan masalah yang memiliki *overlapping subproblems* dan *optimal substructure*. Dengan menyimpan solusi untuk submasalah, DP dapat meningkatkan efisiensi secara signifikan. Dalam kasus Fibonacci, DP mengubah kompleksitas waktu dari eksponensial menjadi linear.

# Fibonacci menggunakan DP

def fibonacci(n, memo={}):

if n in memo:

return memo[n]

if n <= 2:

return 1

memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)

return memo[n]

**Latihan**

1. Implementasikan algoritma Knapsack dengan DP.
2. Buat solusi Shortest Path menggunakan DP.

**Bab 5: Pencarian Heuristik**

**Konsep Heuristik dalam Pencarian Solusi**

Dalam konteks ilmu komputer dan kecerdasan buatan (AI), **heuristik** adalah teknik atau strategi untuk menemukan solusi yang "cukup baik" atau mendekati optimal, terutama ketika mencari solusi yang benar-benar optimal tidak mungkin atau terlalu memakan waktu. Heuristik tidak menjamin solusi terbaik, tetapi memberikan solusi yang memadai dalam waktu yang wajar.

**Ciri-ciri Heuristik**

* **Tidak selalu optimal**: Heuristik tidak menjamin menemukan solusi terbaik, tetapi solusi yang "baik" atau "cukup baik".
* **Bergantung pada masalah**: Heuristik yang baik untuk satu masalah mungkin tidak baik untuk masalah lainnya.
* **Trade-off antara kualitas dan waktu**: Heuristik mengorbankan sedikit kualitas solusi untuk mendapatkan solusi dalam waktu yang lebih cepat.

**Kegunaan Heuristik**

Heuristik sangat berguna dalam situasi berikut:

* **Ruang pencarian yang besar**: Ketika ruang kemungkinan solusi terlalu besar untuk dieksplorasi secara menyeluruh (misalnya, dalam masalah optimasi yang kompleks).
* **Keterbatasan waktu**: Ketika waktu untuk menemukan solusi terbatas.
* **Solusi yang "cukup baik"**: Ketika solusi yang "cukup baik" sudah memadai dan solusi optimal tidak diperlukan.

**Contoh Heuristik Sederhana**

* **Perkiraan Jarak**: Dalam navigasi, kita bisa menggunakan jarak garis lurus antara dua titik sebagai perkiraan jarak terpendek, meskipun jalan yang sebenarnya mungkin berliku-liku.

**Implementasi Algoritma A\* (A Star)**

Algoritma A\* adalah algoritma pencarian jalan terpendek yang menggunakan heuristik untuk meningkatkan efisiensinya. A\* sering digunakan dalam permainan, robotika, dan navigasi.

**Konsep Dasar A\***

A\* mencari jalur terpendek antara dua titik (titik awal dan titik tujuan) dalam sebuah graf (atau peta). A\* menggunakan fungsi evaluasi yang menggabungkan dua faktor:

1. **g(n)**: Biaya (jarak) dari titik awal ke titik n.
2. **h(n)**: Perkiraan biaya (jarak) dari titik n ke titik tujuan (ini adalah fungsi heuristik).

Fungsi evaluasi A\* adalah:

**f(n) = g(n) + h(n)**

**Cara Kerja A\***

1. Mulai dari titik awal.
2. Masukkan titik awal ke dalam "antrian terbuka" (biasanya diimplementasikan sebagai *priority queue*).
3. Selama antrian terbuka tidak kosong:
   * Ambil titik dengan nilai f(n) terendah dari antrian terbuka.
   * Jika titik ini adalah titik tujuan, maka jalur telah ditemukan.
   * Jika tidak, hapus titik ini dari antrian terbuka dan masukkan ke dalam "antrian tertutup".
   * Untuk setiap tetangga dari titik ini:
     + Jika tetangga belum ada di antrian tertutup:
       - Hitung g(n), h(n), dan f(n) untuk tetangga.
       - Masukkan tetangga ke dalam antrian terbuka.

**Contoh Implementasi Sederhana (Python)**

Python

import heapq

def heuristic(a, b):

# Fungsi heuristik (contoh: jarak Manhattan)

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(start, goal):

open\_set = [(0, start)] # Antrian terbuka (f\_score, node)

came\_from = {} # Menyimpan jalur yang dilalui

g\_score = {start: 0} # Biaya dari start ke node

f\_score = {start: heuristic(start, goal)} # Perkiraan total biaya

while open\_set:

current = heapq.heappop(open\_set)[1] # Ambil node dengan f\_score terendah

if current == goal:

# Rekonstruksi jalur

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path

for neighbor in get\_neighbors(current): # Fungsi untuk mendapatkan tetangga

tentative\_g\_score = g\_score[current] + cost\_between(current, neighbor) # Biaya ke tetangga

if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:

# Jalur lebih baik ditemukan

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, goal)

heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

return None # Tidak ada jalur ditemukan

**Pentingnya Heuristik dalam A\***

Kualitas heuristik (h(n)) sangat penting dalam A\*. Heuristik yang baik akan mempercepat pencarian, sementara heuristik yang buruk dapat memperlambat atau bahkan membuat A\* gagal menemukan solusi.

**Kesimpulan**

Heuristik adalah alat yang ampuh dalam pencarian solusi, terutama ketika menghadapi masalah yang kompleks. Algoritma A\* adalah contoh yang baik tentang bagaimana heuristik dapat digunakan untuk meningkatkan efisiensi pencarian jalan terpendek.

import heapq

def a\_star(graph, start, goal):

queue = [(0, start)]

visited = set()

while queue:

cost, node = heapq.heappop(queue)

if node in visited:

continue

if node == goal:

return cost

visited.add(node)

for neighbor, weight in graph[node]:

heapq.heappush(queue, (cost + weight, neighbor))

return float('inf')

**Latihan**

1. Implementasikan Hill Climbing untuk optimasi fungsi matematis.
2. Buat program Simulated Annealing sederhana.

**Bab 6: Pemrograman Berorientasi Objek dalam Struktur Data**

**Konsep OOP dalam Implementasi ADT**

OOP (Object-Oriented Programming) adalah paradigma pemrograman yang berfokus pada "objek", yang merupakan instansi dari kelas. Dalam konteks ADT, OOP memungkinkan kita untuk merepresentasikan ADT sebagai kelas dengan atribut (data) dan metode (operasi) yang sesuai.

**Keuntungan Menggunakan OOP dalam ADT**

* **Enkapsulasi**: Data dan operasi yang terkait dengan ADT dienkapsulasi (dibungkus) dalam kelas, sehingga terlindungi dari akses yang tidak sah dan meminimalkan risiko kesalahan.
* **Abstraksi**: Detail implementasi ADT disembunyikan dari pengguna. Pengguna hanya berinteraksi dengan antarmuka (interface) yang disediakan oleh kelas.
* **Modularitas**: Kode ADT menjadi modular dan mudah dikelola. Perubahan pada implementasi internal kelas tidak akan memengaruhi kode yang menggunakannya, selama antarmuka kelas tetap sama.
* **Reusability**: Kelas ADT dapat digunakan kembali (reuse) dalam berbagai konteks dan aplikasi.
* **Polimorfisme**: OOP mendukung polimorfisme, yang memungkinkan penggunaan objek dari kelas yang berbeda melalui antarmuka yang sama.

**Implementasi Linked List dengan OOP**

Berikut adalah contoh implementasi Linked List dengan OOP dalam Python:

Python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def tambah\_di\_akhir(self, data):

node\_baru = Node(data)

if self.head is None:

self.head = node\_baru

return

node\_terakhir = self.head

while node\_terakhir.next:

node\_terakhir = node\_terakhir.next

node\_terakhir.next = node\_baru

def tampilkan(self):

node\_sekarang = self.head

while node\_sekarang:

print(node\_sekarang.data, end=" ")

node\_sekarang = node\_sekarang.next

print()

**Implementasi Doubly Linked List**

Doubly Linked List adalah Linked List di mana setiap node memiliki dua pointer: satu ke node berikutnya dan satu ke node sebelumnya. Berikut adalah contoh implementasinya dalam Python:

Python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

class DoublyLinkedList:

# ... (metode-metode lain mirip dengan Singly Linked List, dengan penyesuaian untuk prev)

**Implementasi Circular Linked List**

Circular Linked List adalah Linked List di mana pointer next dari node terakhir menunjuk kembali ke node pertama, membentuk lingkaran. Berikut adalah contoh implementasinya dalam Python:

Python

class CircularLinkedList:

# ... (metode-metode lain mirip dengan Singly Linked List, dengan penyesuaian untuk head dan tail)

**Contoh Penggunaan**

Python

# Singly Linked List

linked\_list = LinkedList()

linked\_list.tambah\_di\_akhir(10)

linked\_list.tambah\_di\_akhir(20)

linked\_list.tambah\_di\_akhir(30)

linked\_list.tampilkan() # Output: 10 20 30

# Doubly Linked List

doubly\_linked\_list = DoublyLinkedList()

# ...

# Circular Linked List

circular\_linked\_list = CircularLinkedList()

# ...

**Kesimpulan**

Dengan menggunakan OOP, implementasi ADT Linked List menjadi lebih terstruktur, mudah dibaca, dan mudah dikembangkan. Konsep enkapsulasi, abstraksi, dan modularitas membantu dalam pengelolaan kode yang kompleks dan meminimalkan risiko kesalahan.

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

**Latihan**

1. Buat implementasi **Doubly Linked List**.
2. Implementasikan **Circular Linked List**.

**Bab 7: Graf dan Implementasinya**

**Representasi Graf**

Graf adalah struktur data yang terdiri dari simpul (vertex) dan sisi (edge) yang menghubungkan antar simpul. Graf dapat direpresentasikan dengan dua cara utama:

**1. List Adjacency (Daftar Ketetanggaan)**

List Adjacency adalah representasi graf di mana setiap simpul memiliki daftar tetangga (simpul yang terhubung langsung dengannya).

**Contoh:**

A: B, C

B: A, D

C: A, E

D: B

E: C

**Keuntungan:**

* Efisien untuk graf dengan jumlah sisi yang sedikit (graf jarang).
* Membutuhkan memori yang lebih sedikit daripada matriks adjacency untuk graf yang besar dan jarang.

**Kekurangan:**

* Tidak efisien untuk memeriksa apakah dua simpul terhubung langsung (membutuhkan pencarian dalam daftar tetangga).

**2. Matriks Adjacency (Matriks Ketetanggaan)**

Matriks Adjacency adalah representasi graf di mana hubungan antar simpul dinyatakan dalam matriks. Jika ada sisi antara simpul i dan j, maka elemen matriks[i][j] bernilai 1 (atau nilai lainnya yang sesuai). Jika tidak ada sisi, maka elemen bernilai 0.

**Contoh:**

A B C D E

A 0 1 1 0 0

B 1 0 0 1 0

C 1 0 0 0 1

D 0 1 0 0 0

E 0 0 1 0 0

**Keuntungan:**

* Efisien untuk memeriksa apakah dua simpul terhubung langsung (hanya perlu mengakses elemen matriks).
* Operasi matriks dapat digunakan untuk memanipulasi graf.

**Kekurangan:**

* Membutuhkan memori yang lebih besar daripada list adjacency untuk graf yang besar dan jarang.
* Tidak efisien untuk iterasi melalui tetangga suatu simpul (perlu memeriksa seluruh baris atau kolom matriks).

**Algoritma BFS dan DFS**

**1. Breadth-First Search (BFS)**

BFS adalah algoritma penelusuran graf yang dimulai dari simpul awal dan mengunjungi semua tetangganya terlebih dahulu, kemudian mengunjungi tetangga dari tetangga, dan seterusnya. BFS menggunakan antrian (queue) untuk menyimpan simpul-simpul yang akan dikunjungi.

**Cara Kerja BFS:**

1. Masukkan simpul awal ke dalam antrian.
2. Selama antrian tidak kosong:
   * Ambil simpul dari depan antrian.
   * Kunjungi simpul tersebut.
   * Masukkan semua tetangganya yang belum dikunjungi ke dalam antrian.

**Contoh Implementasi (Python):**

Python

from collections import deque

def bfs(graf, mulai):

dikunjungi = set()

antrian = deque([mulai])

dikunjungi.add(mulai)

while antrian:

simpul = antrian.popleft()

print(simpul, end=" ")

for tetangga in graf[simpul]:

if tetangga not in dikunjungi:

dikunjungi.add(tetangga)

antrian.append(tetangga)

**2. Depth-First Search (DFS)**

DFS adalah algoritma penelusuran graf yang dimulai dari simpul awal dan menjelajahi sedalam mungkin sepanjang satu jalur sebelum kembali dan menjelajahi jalur lainnya. DFS menggunakan tumpukan (stack) atau rekursi untuk menyimpan simpul-simpul yang akan dikunjungi.

**Cara Kerja DFS:**

1. Mulai dari simpul awal.
2. Kunjungi simpul saat ini.
3. Untuk setiap tetangga yang belum dikunjungi:
   * Lakukan DFS secara rekursif pada tetangga tersebut.

**Contoh Implementasi (Python):**

Python

def dfs(graf, mulai, dikunjungi=None):

if dikunjungi is None:

dikunjungi = set()

dikunjungi.add(mulai)

print(mulai, end=" ")

for tetangga in graf[mulai]:

if tetangga not in dikunjungi:

dfs(graf, tetangga, dikunjungi)

**Algoritma Dijkstra dan Floyd-Warshall**

**1. Algoritma Dijkstra**

Algoritma Dijkstra adalah algoritma untuk mencari jalur terpendek antara simpul awal dan semua simpul lainnya dalam graf berbobot (weighted graph) dengan bobot non-negatif.

**Cara Kerja Dijkstra:**

1. Inisialisasi jarak dari simpul awal ke semua simpul lainnya dengan tak hingga.
2. Tetapkan jarak dari simpul awal ke dirinya sendiri menjadi 0.
3. Selama ada simpul yang belum dikunjungi:
   * Pilih simpul dengan jarak terkecil yang belum dikunjungi.
   * Untuk setiap tetangganya:
     + Hitung jarak baru melalui simpul saat ini.
     + Jika jarak baru lebih kecil dari jarak sebelumnya, perbarui jarak tersebut.

**2. Algoritma Floyd-Warshall**

Algoritma Floyd-Warshall adalah algoritma untuk mencari jalur terpendek antara semua pasangan simpul dalam 1 graf berbobot. Algoritma ini dapat menangani graf dengan bobot negatif, tetapi tidak boleh ada siklus negatif (siklus dengan total bobot negatif).

[1. id.scribd.com](https://id.scribd.com/document/664077863/Gabungan-Quiz-Dan-Materi-TT" \t "_blank)

[id.scribd.com](https://id.scribd.com/document/664077863/Gabungan-Quiz-Dan-Materi-TT" \t "_blank)

**Cara Kerja Floyd-Warshall:**

1. Inisialisasi matriks jarak dengan jarak antar simpul langsung (jika ada) atau tak hingga (jika tidak ada).
2. Untuk setiap simpul k:
   * Untuk setiap simpul i:
     + Untuk setiap simpul j:
       - Perbarui jarak[i][j] dengan minimum dari jarak[i][j] dan jarak[i][k] + jarak[k][j].

**Perbandingan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fitur** | **BFS** | **DFS** | **Dijkstra** | **Floyd-Warshall** |
| Jenis Graf | Tidak berbobot | Tidak berbobot | Berbobot non-negatif | Berbobot (bisa negatif) |
| Tujuan | Menemukan jalur terpendek (unweighted) | Menjelajahi seluruh graf | Menemukan jalur terpendek (weighted) | Menemukan jalur terpendek (all-pairs) |
| Kompleksitas | O(V + E) | O(V + E) | O(V^2 + E) atau O(E log V) | O(V^3) |

Ekspor ke Spreadsheet

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(graph[node])

**Latihan**

1. Implementasikan DFS dan BFS untuk graf tertentu.
2. Buat solusi Dijkstra untuk menemukan jalur terpendek.

**Bab 8: Penerapan Praktis dan Studi Kasus**

**1. Pengelolaan Antrian Pelanggan Menggunakan Queue**

**Konsep Queue**

Queue (antrian) adalah struktur data yang mengikuti prinsip FIFO (First-In, First-Out). Elemen yang pertama masuk akan menjadi yang pertama keluar. Analogi sederhananya adalah antrian orang di loket.

**Penerapan dalam Pengelolaan Pelanggan**

* **Customer Service**: Antrian panggilan telepon di call center. Panggilan yang masuk pertama akan dilayani pertama.
* **Restoran**: Antrian pelanggan yang menunggu meja. Pelanggan yang datang lebih awal akan dipersilakan duduk lebih dulu.
* **E-commerce**: Antrian pesanan. Pesanan yang masuk lebih dulu akan diproses lebih dulu.

**Contoh Implementasi (Python)**

Python

from collections import deque

antrian = deque()

# Menambahkan pelanggan ke antrian

antrian.append("Pelanggan A")

antrian.append("Pelanggan B")

antrian.append("Pelanggan C")

# Melayani pelanggan dari antrian

pelanggan = antrian.popleft()

print(f"Melayani: {pelanggan}")

# Menampilkan antrian yang tersisa

print(f"Sisa antrian: {antrian}")

**2. Pencarian Rute Optimal dengan Algoritma Graf**

**Konsep Graf**

Graf adalah struktur data yang terdiri dari simpul (vertex) dan sisi (edge) yang menghubungkan antar simpul. Graf dapat digunakan untuk merepresentasikan berbagai macam hubungan, seperti peta jalan, jaringan komputer, atau hubungan sosial.

**Algoritma Pencarian Rute**

Terdapat berbagai macam algoritma untuk mencari rute terpendek dalam graf, di antaranya:

* **Dijkstra**: Untuk graf dengan bobot non-negatif.
* **Bellman-Ford**: Untuk graf dengan bobot negatif (tetapi tidak ada siklus negatif).
* **A**\*: Menggunakan heuristik untuk mempercepat pencarian.

**Penerapan**

* **Navigasi**: Aplikasi peta seperti Google Maps menggunakan algoritma graf untuk mencari rute terpendek antara dua lokasi.
* **Logistik**: Perusahaan logistik menggunakan algoritma graf untuk merencanakan rute pengiriman yang efisien.
* **Jaringan Komputer**: Protokol routing dalam jaringan komputer menggunakan algoritma graf untuk menemukan jalur terbaik untuk mengirim data.

**Contoh Implementasi (Dijkstra dengan Python)**

Python

import heapq

def dijkstra(graf, mulai):

jarak = {simpul: float('inf') for simpul in graf}

jarak[mulai] = 0

prioritas\_queue = [(0, mulai)]

while prioritas\_queue:

jarak\_sekarang, simpul\_sekarang = heapq.heappop(prioritas\_queue)

if jarak\_sekarang > jarak[simpul\_sekarang]:

continue

for tetangga, bobot in graf[simpul\_sekarang].items():

jarak\_baru = jarak\_sekarang + bobot

if jarak\_baru < jarak[tetangga]:

jarak[tetangga] = jarak\_baru

heapq.heappush(prioritas\_queue, (jarak\_baru, tetangga))

return jarak

# Contoh graf (dictionary bersarang)

graf = {

'A': {'B': 4, 'C': 2},

'B': {'D': 5, 'E': 12},

'C': {'F': 10, 'G': 7},

'D': {'H': 3},

'E': {'H': 2},

'F': {'H': 5},

'G': {'H': 4}

}

# Mencari jarak terpendek dari simpul 'A'

jarak\_terpendek = dijkstra(graf, 'A')

print(jarak\_terpendek)

**3. Implementasi Pemrosesan Data Menggunakan Struktur Data Python**

**Struktur Data Python**

Python memiliki berbagai macam struktur data yang dapat digunakan untuk memproses data, di antaranya:

* **List**: Untuk menyimpan urutan elemen.
* **Tuple**: Mirip dengan list, tetapi immutable (tidak dapat diubah).
* **Set**: Untuk menyimpan kumpulan elemen unik.
* **Dictionary**: Untuk menyimpan pasangan kunci-nilai.

**Penerapan**

* **Analisis Data**: Struktur data Python sangat berguna dalam analisis data, seperti untuk menyimpan data dari file CSV, melakukan manipulasi data, dan menghitung statistik.
* **Machine Learning**: Struktur data Python digunakan dalam implementasi algoritma machine learning, seperti untuk menyimpan data training dan model.
* **Web Development**: Struktur data Python digunakan dalam pengembangan web, seperti untuk menyimpan data dari database dan menampilkan data kepada pengguna.

**Contoh Implementasi (Analisis Data Sederhana)**

Python

data = [

{"nama": "Andi", "nilai": 80},

{"nama": "Budi", "nilai": 90},

{"nama": "Caca", "nilai": 75}

]

# Menghitung rata-rata nilai

total\_nilai = sum(item["nilai"] for item in data)

rata\_rata = total\_nilai / len(data)

print(f"Rata-rata nilai: {rata\_rata}")

# Mencari siswa dengan nilai tertinggi

nilai\_tertinggi = max(item["nilai"] for item in data)

siswa\_tertinggi = [item["nama"] for item in data if item["nilai"] == nilai\_tertinggi]

print(f"Siswa dengan nilai tertinggi: {siswa\_tertinggi}")

**Kesimpulan**

Struktur data dan algoritma adalah fondasi penting dalam pemrograman. Dengan memahami dan menguasai konsep-konsep ini, Anda dapat membangun aplikasi yang efisien dan efektif untuk berbagai macam kebutuhan.

**Latihan Akhir**

1. Buat sistem manajemen antrean menggunakan Queue.
2. Implementasikan simulasi navigasi kota menggunakan Dijkstra.
3. Kembangkan aplikasi sederhana berbasis Python dengan menerapkan semua konsep yang telah dipelajari.