

LAPORAN TUGAS STRUKTUR DATA GRAPH DIJKSTRA



DISUSUN OLEH :

1. Dimas Wijil Pamungkas (23091397201)
2. M. Raka Phaedra Agus Putra (23091397210)
3. Vani Fransiska (23091397193)

DOSEN PENGAMPU :

I Gde Agung Sri Sidhimantra, S.Kom., M.Kom

PRODI D4 MANAJEMEN INFORMATIKA
FAKULTAS VOKASI
UNIVERSITAS NEGERI SURABAYA
/2023

source Code :

```
class Peta:
    def __init__(self):
        self.listKota = {}

    def printPeta(self):
        for kota in self.listKota:
            print(kota, ":", self.listKota[kota])

    def tambahkanKota(self, kota):
        if kota not in self.listKota:
            self.listKota[kota] = {}
            return True
        return False

# Metode untuk mencetak informasi peta, termasuk kota-kota yang ada dan
# jarak antar kota
    def hapusKota(self, hapusKota):
        if hapusKota in self.listKota:
            for kotaLain in self.listKota:
                if hapusKota in self.listKota[kotaLain]:
                    del self.listKota[kotaLain][hapusKota]
            del self.listKota[hapusKota]
            return True
        return False

# Metode untuk menghapus jalan antara dua kota
    def tambahkanJalan(self, kota1, kota2, jarak):
        if kota1 in self.listKota and kota2 in self.listKota:
            self.listKota[kota2][kota1] = jarak, 'KM'
            self.listKota[kota1][kota2] = jarak, 'KM'
            return True
        return False
        # Hapus jalur dari kota2 ke kota1
    def hapusJalan(self, kota1, kota2):
        if kota1 in self.listKota and kota2 in self.listKota:
            if kota1 in self.listKota[kota2]:
                del self.listKota[kota2][kota1]
            if kota2 in self.listKota[kota1]:
                del self.listKota[kota1][kota2]
            return True
        return False
        # Implementasi algoritma Dijkstra untuk mencari jarak terpendek
        # dari suatu kota ke kota-kota lainnya

    def djikstra (self, source):
        jarak = {kota: float('inf') for kota in self.listKota}
        jarak[source] = 0

        unvisitedCities = list(self.listKota.keys())
        while unvisitedCities:
```

```

        minJarak = float('inf')
        dekatKota = None

        for kota in unvisitedCities:
            if jarak[kota] < minJarak:
                minJarak = jarak[kota]
                dekatKota = kota
        unvisitedCities.remove(dekatKota)

        for neighbor, weight in self.listKota[dekatKota].items():
            jarakNeighbor = jarak[dekatKota] + weight[0]
            if jarakNeighbor < jarak[neighbor]:
                jarak[neighbor] = jarakNeighbor

    return jarak

# Menambahkan beberapa kota ke dalam peta
petajawabarat = Peta()
petajawabarat.tambahkanKota("Cikampek")
petajawabarat.tambahkanKota("Purwakarta")
petajawabarat.tambahkanKota("Subang")
petajawabarat.tambahkanKota("Bandung")
petajawabarat.tambahkanKota("Sumedang")
petajawabarat.tambahkanKota("Garut")
petajawabarat.tambahkanKota("Cianjur")
petajawabarat.tambahkanKota("Sukabumi")
petajawabarat.tambahkanKota("Tasikmalaya")
petajawabarat.tambahkanKota("Kuningan")
petajawabarat.tambahkanKota("Cirebon")
petajawabarat.tambahkanKota("Banjar")
petajawabarat.tambahkanKota("Indramayu")
print('-----')
print('Jalan yang terhubung')
# Menggunakan algoritma Dijkstra untuk mencari jarak terpendek dari
suatu kota ke kota-kota lainnya
petajawabarat.tambahkanJalan("Cikampek", "Purwakarta", 22)
petajawabarat.tambahkanJalan("Subang", "Cikampek", 54)
petajawabarat.tambahkanJalan("Subang", "Purwakarta", 57)
petajawabarat.tambahkanJalan("Bandung", "Subang", 61)
petajawabarat.tambahkanJalan("Cianjur", "Sukabumi", 31)
petajawabarat.tambahkanJalan("Cianjur", "Bandung", 65)
petajawabarat.tambahkanJalan("Bandung", "Sumedang", 54)
petajawabarat.tambahkanJalan("Bandung", "Purwakarta", 62)
petajawabarat.tambahkanJalan("Bandung", "Garut", 131)
petajawabarat.tambahkanJalan("Garut", "Tasikmalaya", 88)
petajawabarat.tambahkanJalan("Garut", "Kuningan", 161)
petajawabarat.tambahkanJalan("Cirebon", "Kuningan", 35)
petajawabarat.tambahkanJalan("Sumedang", "Kuningan", 120)
petajawabarat.tambahkanJalan("Tasikmalaya", "Kuningan", 80)

```

```

petajawabarat.tambahkanJalan("Banjar","Kuningan", 73)
petajawabarat.tambahkanJalan("Banjar","Tasikmalaya", 44)
petajawabarat.tambahkanJalan("Cirebon","Sumedang", 95)
petajawabarat.tambahkanJalan("Cirebon","Indramayu", 55)
petajawabarat.tambahkanJalan("Subang","Indramayu", 81)
petajawabarat.tambahkanJalan("Sumedang","Indramayu", 99)
petajawabarat.printPeta()
print('-----')
#print peta jawa barat kota ('Bandung')

petajawabarat.dijkstra("Bandung")
jarakSemuaKota = petajawabarat.dijkstra("Bandung")
print("Jarak Kota Berikut Nya dari Bandung")
for kota, jarak in jarakSemuaKota.items():
    print(kota, "adalah", jarak, "KM")

```

Penjelasan Step By Step:

- Membuat sebuah kelas dengan nama **Peta**
 - Kelas adalah sebuah blueprint untuk membuat objek-objek yang memiliki properti dan metode tertentu.
 - **def __init__(self)** adalah metode khusus yang disebut **__init__**. Metode ini adalah konstruktor kelas yang dipanggil secara otomatis setiap kali sebuah objek dari kelas dibuat. **self** merujuk pada objek yang sedang dibuat. Dalam konstruktor ini, kita memulai pembuatan objek dengan menginisialisasi properti-properti awalnya.
 - **self.listKota = {}**: Di dalam metode konstruktor, kita membuat atribut **listKota** untuk setiap objek yang dibuat dari kelas **Peta**. **listKota** adalah sebuah kamus (dictionary) kosong. Dengan menginisialisasi **cityList** menjadi kamus kosong, kita siap untuk menambahkan kota-kota ke dalamnya nanti.

```

class Peta:
    def __init__(self):
        self.listKota = {}

```

- Membuat fungsi **printPeta**.
 - **def printPeta(self)** adalah definisi metode baru dalam kelas **Peta** yang disebut **printPeta**. Metode ini tidak memiliki argumen selain **self**, yang merujuk pada objek kelas itu sendiri.

- o **for kota in self.listKota** adalah pernyataan loop **for** yang digunakan untuk mengulangi setiap elemen dalam kamus **listKota** dari objek **Peta**. Dalam setiap iterasi, variabel kota akan mewakili kunci (nama kota) dalam kamus.
- o **print(kota, ":", self.listKota[kota])** digunakan untuk mencetak nama kota (kunci) diikuti oleh nilai yang terkait (mungkin merupakan informasi tambahan tentang kota tersebut) dari kamus **listKota**. Dengan menggunakan format print, kita mencetak kunci dan nilai dengan menggunakan **:** sebagai pemisah di antara keduanya.

```
def printPeta(self):
    for kota in self.listKota:
        print(kota, ":", self.listKota[kota])
```

- fungsi **tambahkanKota** bertujuan untuk menambahkan kota baru ke dalam peta yang direpresentasikan oleh kelas **Peta**
 - o **def tambahkanKota(self, kota):** Ini adalah definisi dari metode **tambahkanKota**. Metode ini memiliki dua parameter: **self**, yang merujuk pada objek instan dari kelas **Peta**, dan **kota**, yang merupakan kota yang akan ditambahkan ke dalam peta.
 - o **if kota not in self.listKota:** Ini adalah pernyataan kondisional yang memeriksa apakah kota sudah ada dalam kamus **listKota**. Jika tidak ada, itu berarti kota tersebut belum ditambahkan ke dalam peta.
 - o **self.listKota[kota] = {}:** Jika kota belum ada dalam peta (**listKota**), maka kita menambahkannya ke dalam peta dengan menetapkan nilai awal berupa kamus kosong **{}** untuk kota tersebut. Ini menunjukkan bahwa kita telah menambahkan kota baru ke dalam peta.
 - o **return True:** Setelah menambahkan kota ke dalam peta, kita mengembalikan **True** untuk menandakan bahwa operasi penambahan berhasil dilakukan.
 - o **return False:** Jika kota sudah ada dalam peta, maka metode ini langsung mengembalikan **False**, menandakan bahwa kota tersebut sudah ada sebelumnya dalam peta dan tidak perlu ditambahkan kembali.

```
def tambahkanKota(self, kota):
    if kota not in self.listKota:
        self.listKota[kota] = {}
        return True
    return False
```

- Membuat sebuah metode untuk mencetak informasi peta
 - o **def hapusKota(self, hapusKota):** adalah definisi sebuah metode (fungsi yang terkait dengan suatu objek) bernama **hapusKota**. Metode ini menerima dua parameter, yaitu **self** (objek itu sendiri) dan **hapusKota**, yang merupakan kota yang ingin dihapus dari peta.

- o **if hapusKota in self.listKota:** adalah kondisional (if statement) yang memeriksa apakah **hapusKota** ada dalam struktur data **listKota** yang terdapat dalam objek yang memanggil metode ini. **self.listKota** mungkin adalah kamus (dictionary) di mana kunci adalah nama kota dan nilai adalah informasi terkait dengan kota tersebut.
- o **for kotaLain in self.listKota:** adalah loop (perulangan) yang akan mengulang semua kota dalam **self.listKota**. **kotaLain** mungkin adalah variabel sementara yang mewakili setiap kota dalam loop ini.
- o **if hapusKota in self.listKota[kotaLain]:** adalah kondisional lagi yang memeriksa apakah **hapusKota** adalah tetangga dari **kotaLain** dalam peta. Jika iya, maka jarak antara **kotaLain** dan **hapusKota** akan dihapus dari peta.
- o **del self.listKota[kotaLain][hapusKota]** merupakan pernyataan untuk menghapus kota **hapusKota** dari daftar tetangga **kotaLain**.
- o **del self.listKota[hapusKota]** digunakan untuk menghapus kota **hapusKota** secara keseluruhan dari peta, karena tidak ada lagi kota yang terhubung dengannya.

```
# Metode untuk mencetak informasi peta, termasuk kota-kota yang ada dan jarak antar kota
def hapusKota(self, hapusKota):
    if hapusKota in self.listKota:
        for kotaLain in self.listKota:
            if hapusKota in self.listKota[kotaLain]:
                del self.listKota[kotaLain][hapusKota]
        del self.listKota[hapusKota]
    return True
return False
```

- Membuat fungsi untuk menghapus jalan di antara 2 kota
 - o **def tambahkanJalan :** Merupakan fungsi yang bertujuan untuk menambahkan jalan dua arah antara dua kota bersama dengan jaraknya. Jika kedua kota sudah ada dalam **self.listKota**, maka jarak antara kota1 dan kota2 ditambahkan ke kamus kota2 di bawah kunci kota1 dan sebaliknya. Ini dilakukan untuk memastikan bahwa jarak antara kota1 dan kota2 dapat diakses dari keduanya. Pengembalian **True** menandakan bahwa penambahan jalan berhasil dilakukan, sedangkan **False** menandakan bahwa setidaknya satu dari kota tidak ditemukan dalam daftar kota.
 - o **def hapusJalan ::** Metode ini digunakan untuk menghapus jalan antara dua kota.

```
# Metode untuk menghapus jalan antara dua kota
def tambahkanJalan(self, kota1, kota2, jarak):
    if kota1 in self.listKota and kota2 in self.listKota:
        self.listKota[kota2][kota1] = jarak, 'KM'
        self.listKota[kota1][kota2] = jarak, 'KM'
```

```

        return True
    return False
    # Hapus jalur dari kota2 ke kota1
def hapusJalan(self, kota1, kota2):
    if kota1 in self.listKota and kota2 in self.listKota:
        if kota1 in self.listKota[kota2]:
            del self.listKota[kota2][kota1]
        if kota2 in self.listKota[kota1]:
            del self.listKota[kota1][kota2]
        return True
    return False

```

- Membuat fungsi untuk mengImplementasi algoritma Dijkstra untuk mencari jarak terpendek dari suatu kota ke kota-kota lainnya
 - o menginisialisasi variabel **jarak** sebagai kamus di mana setiap kota memiliki jarak awal yang tak terbatas (**float('inf')**). Ini dilakukan dengan menggunakan ekspresi kamus (dictionary) Python **{kota: float('inf') for kota in self.listKota}**. Kemudian, jarak dari sumber (source) ke sumber itu sendiri diatur menjadi 0.
 - o Semua kota yang belum dikunjungi diatur sebagai kota yang harus dikunjungi selanjutnya. Ini dilakukan dengan membuat daftar **unvisitedCities** yang berisi semua kota dalam **self.listKota**.

```

def djikstra (self, source):
    jarak = {kota: float('inf') for kota in self.listKota}
    jarak[source] = 0

    unvisitedCities = list(self.listKota.keys())
    while unvisitedCities:
        minJarak = float('inf')
        dekatKota = None

        for kota in unvisitedCities:
            if jarak[kota] < minJarak:
                minJarak = jarak[kota]
                dekatKota = kota
        unvisitedCities.remove(dekatKota)

        for neighbor, weight in self.listKota[dekatKota].items():
            jarakNeighbor = jarak[dekatKota] + weight[0]
            if jarakNeighbor < jarak[neighbor]:
                jarak[neighbor] = jarakNeighbor

    return jarak

```

- Membuat fungsi untuk menambahkan kota dalam peta
 - o `petajawabarat = Peta()`: digunakan untuk membuat objek baru dari kelas `Peta` dan menyimpannya dalam variabel `petajawabarat`.
 - o `tambahkanKota` digunakan untuk menambahkan kota dari objek `petajawabarat` sehingga dalam setiap baris perlu dimasukkan kota-kota yang diinginkan.

```
petajawabarat = Peta()
petajawabarat.tambahkanKota("Cikampek")
petajawabarat.tambahkanKota("Purwakarta")
petajawabarat.tambahkanKota("Subang")
petajawabarat.tambahkanKota("Bandung")
petajawabarat.tambahkanKota("Sumedang")
petajawabarat.tambahkanKota("Garut")
petajawabarat.tambahkanKota("Cianjur")
petajawabarat.tambahkanKota("Sukabumi")
petajawabarat.tambahkanKota("Tasikmalaya")
petajawabarat.tambahkanKota("Kuningan")
petajawabarat.tambahkanKota("Cirebon")
petajawabarat.tambahkanKota("Banjar")
petajawabarat.tambahkanKota("Indramayu")
```

- Membuat fungsi print

```
print('-----')
print('Jalan yang terhubung')
```

- Membuat fungsi Menambahkan jarak pada semua jalan di peta agar dapat dicari jalan terdekat
 - o `petajawabarat.tambahkanJalan("Cikampek","Purwakarta", 22)` bermaksud untuk menambahkan jarak antara cikampek dan purwakarta adalah 22 Km

```
petajawabarat.tambahkanJalan("Cikampek","Purwakarta", 22)
petajawabarat.tambahkanJalan("Subang","Cikampek",54)
petajawabarat.tambahkanJalan("Subang","Purwakarta", 57)
petajawabarat.tambahkanJalan("Bandung","Subang", 61)
petajawabarat.tambahkanJalan("Cianjur","Sukabumi", 31)
petajawabarat.tambahkanJalan("Cianjur","Bandung", 65)
petajawabarat.tambahkanJalan("Bandung","Sumedang", 54)
petajawabarat.tambahkanJalan("Bandung","Purwakarta", 62)
petajawabarat.tambahkanJalan("Bandung","Garut", 131)
petajawabarat.tambahkanJalan("Garut","Tasikmalaya", 88)
petajawabarat.tambahkanJalan("Garut","Kuningan", 161)
petajawabarat.tambahkanJalan("Cirebon","Kuningan", 35)
petajawabarat.tambahkanJalan("Sumedang","Kuningan", 120)
petajawabarat.tambahkanJalan("Tasikmalaya","Kuningan", 80)
petajawabarat.tambahkanJalan("Banjar","Kuningan", 73)
petajawabarat.tambahkanJalan("Banjar","Tasikmalaya", 44)
petajawabarat.tambahkanJalan("Cirebon","Sumedang", 95)
```

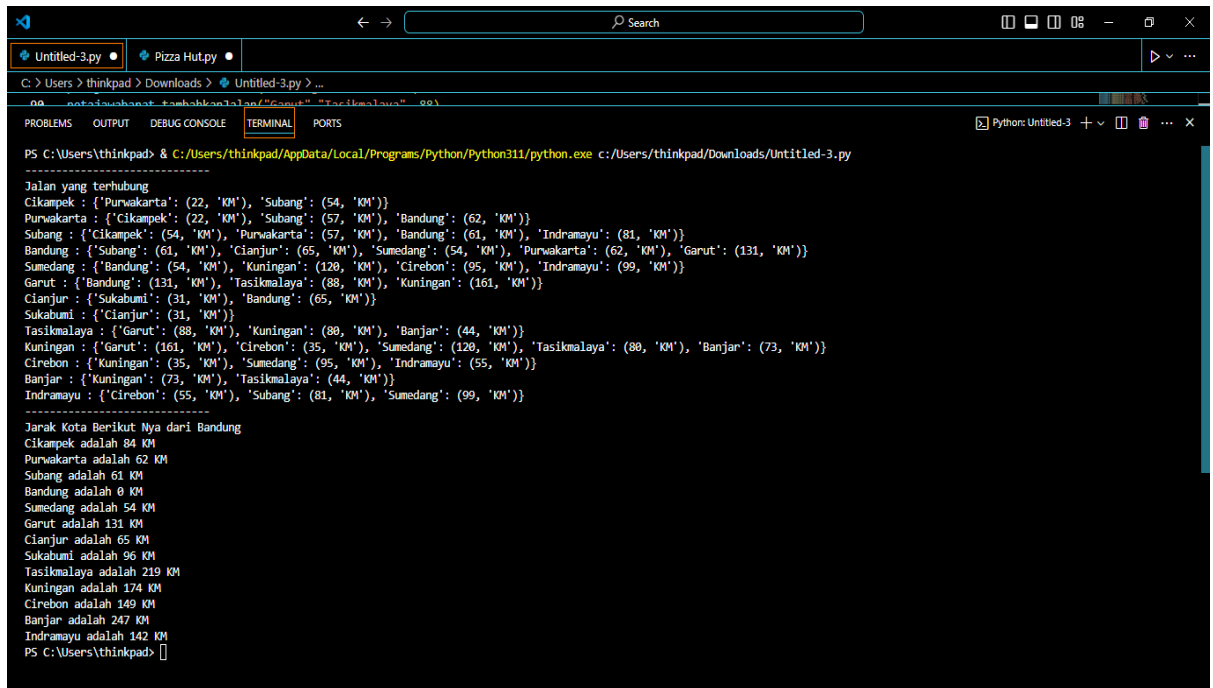


```
petajawabarat.tambahkanJalan("Cirebon","Indramayu", 55)
petajawabarat.tambahkanJalan("Subang","Indramayu", 81)
petajawabarat.tambahkanJalan("Sumedang","Indramayu", 99)
petajawabarat.printPeta()
print('-----')
```

- Membuat fungsi `dijkstra` untuk kota bandung
 - o `petajawabarat.dijkstra("Bandung")`: Ini memanggil fungsi **dijkstra** dari objek **petajawabarat** dengan parameter "Bandung". Ini diasumsikan bahwa **petajawabarat** adalah sebuah objek yang memiliki metode **dijkstra** untuk menghitung jarak terpendek dari satu kota ke semua kota lainnya.
 - o `jarakSemuaKota = petajawabarat.dijkstra("Bandung")`: Hasil dari pemanggilan fungsi di atas disimpan dalam variabel **jarakSemuaKota**. Ini kemungkinan besar akan berupa kamus (dictionary) di mana kunci adalah nama kota dan nilai adalah jarak dari Bandung ke kota tersebut.
 - o `print("Jarak Kota Berikut Nya dari Bandung")`: Ini hanya mencetak judul atau pesan yang memberi tahu bahwa daftar jarak kota akan dicetak.
 - o `for kota, jarak in jarakSemuaKota.items():` Ini adalah loop **for** yang akan mengulang melalui semua item dalam kamus **jarakSemuaKota**, di mana **kota** akan berisi nama kota dan **jarak** akan berisi jarak dari Bandung ke kota tersebut.
 - o `print(kota, "adalah", jarak, "KM")`: Ini mencetak nama kota, diikuti dengan kata "adalah", kemudian jarak dalam kilometer.

```
petajawabarat.dijkstra("Bandung")
jarakSemuaKota = petajawabarat.dijkstra("Bandung")
print("Jarak Kota Berikut Nya dari Bandung")
for kota, jarak in jarakSemuaKota.items():
    print(kota, "adalah", jarak, "KM")
```

Output :



```
C: > Users\thinkpad> Downloads> Untitled-3.py > ...
00 ...otaisa...banar...tambahkanjalan("Garut", "Tasikmalaya", 89)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python: Untitled-3 + - - - x

PS C:\Users\thinkpad> & C:/Users/thinkpad/AppData/Local/Programs/Python/Python311/python.exe c:/Users/thinkpad/Downloads/Untitled-3.py

-----
Jalan yang terhubung
Cikampek : {'Purwakarta': (22, 'KM'), 'Subang': (54, 'KM')}
Purwakarta : {'Cikampek': (22, 'KM'), 'Subang': (57, 'KM'), 'Bandung': (62, 'KM')}
Subang : {'Cikampek': (54, 'KM'), 'Purwakarta': (57, 'KM'), 'Bandung': (61, 'KM'), 'Indramayu': (81, 'KM')}
Bandung : {'Subang': (61, 'KM'), 'Cianjur': (65, 'KM'), 'Sumedang': (54, 'KM'), 'Purwakarta': (62, 'KM'), 'Garut': (131, 'KM')}
Sumedang : {'Bandung': (54, 'KM'), 'Kuningan': (120, 'KM'), 'Cirebon': (95, 'KM'), 'Indramayu': (99, 'KM')}
Garut : {'Bandung': (131, 'KM'), 'Tasikmalaya': (88, 'KM'), 'Kuningan': (161, 'KM')}
Cianjur : {'Sukabumi': (31, 'KM'), 'Bandung': (65, 'KM')}
Sukabumi : {'Cianjur': (31, 'KM')}
Tasikmalaya : {'Garut': (88, 'KM'), 'Kuningan': (80, 'KM'), 'Banjar': (44, 'KM')}
Kuningan : {'Garut': (161, 'KM'), 'Cirebon': (35, 'KM'), 'Sumedang': (120, 'KM'), 'Tasikmalaya': (80, 'KM'), 'Banjar': (73, 'KM')}
Cirebon : {'Kuningan': (35, 'KM'), 'Sumedang': (95, 'KM'), 'Indramayu': (55, 'KM')}
Banjar : {'Kuningan': (73, 'KM'), 'Tasikmalaya': (44, 'KM')}
Indramayu : {'Cirebon': (55, 'KM'), 'Subang': (81, 'KM'), 'Sumedang': (99, 'KM')}
-----
Jarak Kota Berikut Nya dari Bandung
Cikampek adalah 84 KM
Purwakarta adalah 62 KM
Subang adalah 61 KM
Bandung adalah 0 KM
Sumedang adalah 54 KM
Garut adalah 131 KM
Cianjur adalah 65 KM
Sukabumi adalah 96 KM
Tasikmalaya adalah 219 KM
Kuningan adalah 174 KM
Cirebon adalah 149 KM
Banjar adalah 247 KM
Indramayu adalah 142 KM
PS C:\Users\thinkpad> []
```

Link Github : <https://github.com/dimaswijil/Struktur-Data->