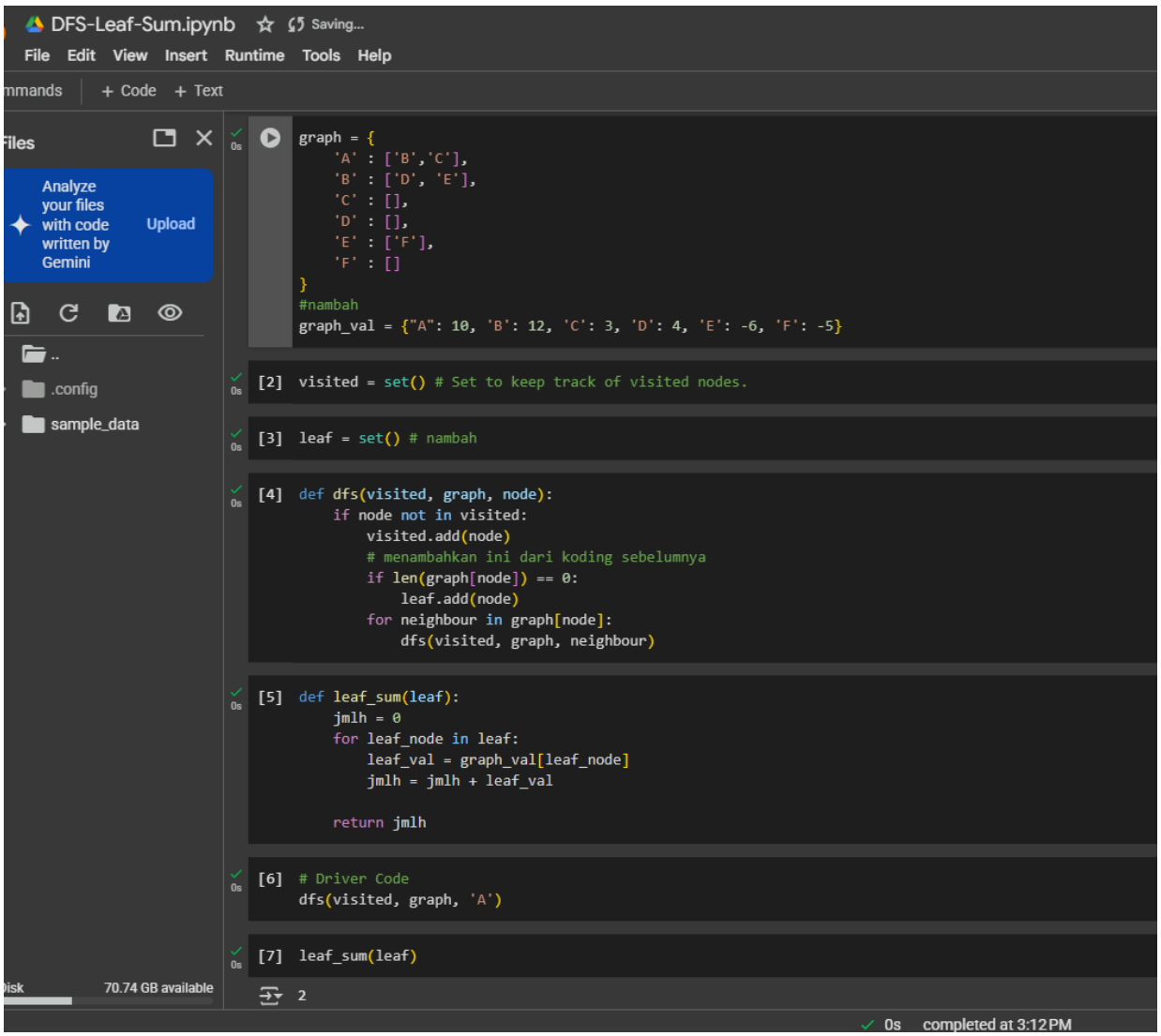
MUHAMMAD TARMIDZI BARIQ

51422161

3IA11

M6



graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : [],

'D' : [],

'E' : ['F'],

```
    'F' : []
}
#nambah
graph_val = {"A": 10, 'B': 12, 'C': 3, 'D': 4, 'E': -6, 'F': -5}
```

Di sini, graph adalah representasi graf menggunakan dictionary. Setiap kunci (key) dalam dictionary merujuk pada sebuah node (simpul) dalam graf, dan nilai (value) yang berhubungan dengan kunci tersebut adalah daftar (list) yang berisi node-node yang terhubung langsung (atau anak) dari node tersebut.

```
visited = set() # Set to keep track of visited nodes.
```

visited adalah sebuah set (himpunan) yang digunakan untuk melacak node-node yang sudah dikunjungi dalam graf atau pohon.

Set adalah struktur data di Python yang berfungsi untuk menyimpan elemen-elemen yang unik. Artinya, tidak ada elemen yang dapat muncul lebih dari sekali dalam set tersebut.

```
leaf = set() # nambah
```

leaf adalah sebuah set yang digunakan untuk melacak node-node yang merupakan leaf nodes (simpul daun) dalam sebuah graf atau pohon.

Simpul daun (leaf node) adalah node yang tidak memiliki anak (child nodes). Dalam struktur pohon, leaf nodes berada pada kedalaman terjauh dan tidak memiliki cabang lebih lanjut.

```
 def dfs(visited, graph, node):
    if node not in visited:
```

Fungsi ini memeriksa apakah node yang saat ini akan dikunjungi sudah ada di dalam set visited. Jika sudah ada, maka kita tidak akan mengunjungi node tersebut lagi untuk menghindari loop atau kunjungan ganda.

```
        visited.add(node)
```

Jika node belum dikunjungi, maka node tersebut ditambahkan ke dalam set visited yang berfungsi untuk menyimpan node-node yang telah dikunjungi. Dengan ini, kita memastikan tidak ada node yang dikunjungi lebih dari sekali.

```
        # menambahkan ini dari koding sebelumnya
        if len(graph[node]) == 0:
            leaf.add(node)
```

Pada bagian ini, kode memeriksa apakah node yang sedang diperiksa adalah sebuah leaf node (simpul daun).

Graph node] adalah daftar yang berisi anak-anak dari node tersebut.

len(graph[node]) == 0 berarti node tersebut tidak memiliki anak, jadi ini adalah simpul daun.

Jika node adalah leaf, maka node tersebut ditambahkan ke dalam set leaf.

```
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
```

Bagian ini adalah inti dari pencarian DFS. Untuk setiap tetangga (neighbor) dari node saat ini, fungsi dfs dipanggil secara rekursif untuk mengunjungi tetangga tersebut, memastikan bahwa seluruh pohon atau graf ditelusuri.

```
def leaf_sum(leaf):
    jmlh = 0
```

Di sini, kita memulai variabel jmlh dengan nilai 0. Variabel ini akan digunakan untuk menyimpan total jumlah nilai dari semua leaf nodes.

```
    for leaf_node in leaf:
```

Kita melakukan iterasi pada set leaf, yang berisi semua node daun (leaf nodes) yang telah ditemukan sebelumnya dalam graf. Setiap iterasi, variabel leaf_node akan berisi satu elemen dari set leaf.

```
        leaf_val = graph_val[leaf_node]
```

Untuk setiap node daun (leaf_node), kita mengambil nilai yang terkait dengan node tersebut dari dictionary graph_val. Dictionary ini menyimpan nilai untuk setiap node dalam graf. Misalnya, jika leaf_node adalah 'C', maka leaf_val = graph_val['C'], yang akan menghasilkan nilai sesuai dengan yang ada di graph_val.

```
        jmlh = jmlh + leaf_val
```

Nilai yang diambil untuk node daun (leaf_val) ditambahkan ke dalam variabel jmlh, yang menyimpan total jumlah nilai dari semua leaf nodes.

```
    return jmlh
```

Setelah selesai melakukan iterasi untuk semua leaf nodes, fungsi mengembalikan total jumlah nilai yang telah dihitung.

```
# Driver Code

dfs(visited, graph, 'A')
```

Berfungsi untuk memanggil fungsi dfs yang telah dijelaskan sebelumnya, untuk memulai pencarian Depth First Search (DFS) pada graf.

```
leaf_sum(leaf)
```

Fungsi leaf_sum(leaf) digunakan untuk menghitung jumlah nilai dari semua leaf node dalam graf berdasarkan nilai yang disediakan dalam dictionary graph_val.

Graph-Height.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text

```
[ ]  # A binary tree node
     class Node:
         # Constructor to create a new node
         def __init__(self, data):
             self.data = data
             self.left = None
             self.right = None
```

```
[ ]  # Compute the "maxDepth" of a tree -- the number of nodes
     # along the longest path from the root node down to the
     # farthest leaf node
     def maxDepth(node):
         if node is None:
             return -1 ;

         else :

             # Compute the depth of each subtree
             lDepth = maxDepth(node.left)
             rDepth = maxDepth(node.right)

             # Use the larger one
             if (lDepth > rDepth):
                 return lDepth+1
             else:
                 return rDepth+1
```

```
[ ]  # Driver program to test above function
     root = Node(1)
     root.left = Node(2)
     root.right = Node(3)
     root.left.left = Node(4)
     root.left.right = Node(5)
```

```
print ("Height of tree is %d" %(maxDepth(root)))

# This code is contributed by Amit Srivastav
```

```
Height of tree is 2
```

```
# A binary tree node

class Node:

    # Constructor to create a new node

    def __init__(self, data):

        self.data = data

        self.left = None

        self.right = None
```

node kelas yang merepresentasikan sebuah node dalam binary tree (pohon biner)

```
# Compute the "maxDepth" of a tree -- the number of nodes

# along the longest path from the root node down to the

# farthest leaf node

def maxDepth(node):


    if node is None:

        return -1 ;
```

Jika node adalah None, berarti kita sedang berada pada ujung pohon (tidak ada node lagi), maka kita mengembalikan -1.

```
    else :


        # Compute the depth of each subtree

        lDepth = maxDepth(node.left)

        rDepth = maxDepth(node.right)
```

Fungsi maxDepth dipanggil secara rekursif untuk anak kiri (node.left) dan anak kanan (node.right).

Fungsi ini akan menghitung kedalaman dari kedua sisi pohon (subtree kiri dan kanan).

```
        # Use the larger one
        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1
```

Fungsi ini membandingkan kedalaman dari kedua subtree (kiri dan kanan).

Fungsi akan mengembalikan kedalaman dari subtree yang lebih dalam (lebih besar).

Kedalaman dihitung dengan menambahkan 1 untuk node saat ini, karena kita menghitung total kedalaman sepanjang path dari akar hingga daun.

```
# Driver program to test above function
root = Node(1)          # Membuat node dengan nilai 1 sebagai root
root.left = Node(2)     # Membuat child kiri dari root (node 2)
root.right = Node(3)    # Membuat child kanan dari root (node 3)
root.left.left = Node(4) # Membuat child kiri dari node 2 (node 4)
root.left.right = Node(5) # Membuat child kanan dari node 2 (node 5)
print("Height of tree is %d" % (maxDepth(root)))
```

Fungsi ini dipanggil untuk menghitung kedalaman atau height dari pohon biner yang dimulai dari node root.