

LAPORAN PRAKTIKUM 6

ANALISIS ALGORITMA



DISUSUN OLEH:

Hadiza Cahya Firdaus

140810180042

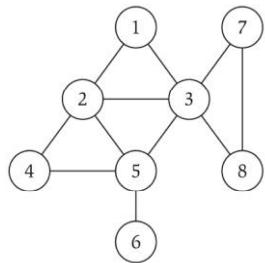
**Program Studi S-1 Teknik Informatika
Departemen Ilmu Komputer
Fakultas Matematika dan Ilmu Pengetahuan Alam
Universitas Padjadjaran 2020**

Pendahuluan

Graf Tak Berarah (Undirected Graf)

(Undirected) graph: $G=(V,E)$

- V = sekumpulan node (vertex, simpul, titik, sudut)
- E = sekumpulan edge (garis, tepi)
- Menangkap hubungan berpasangan antar objek.
- Parameter ukuran Graf: $n = |V|$, $m = |E|$



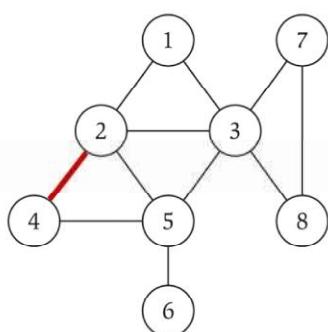
$V = \{1,2,3,4,5,6,7,8\}$
 $E = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6), (7,8)\}$ $n = 8$ $M = 11$

Dalam pemrograman, Graf dapat direpresentasikan dengan **adjacency matrix** dan **adjacency list**

Representasi Graf dengan Adjacency Matrix

Adjacency Matrix: n -ke- n matriks dengan $a_{ij} = 1$ jika (u,v) adalah sebuah garis

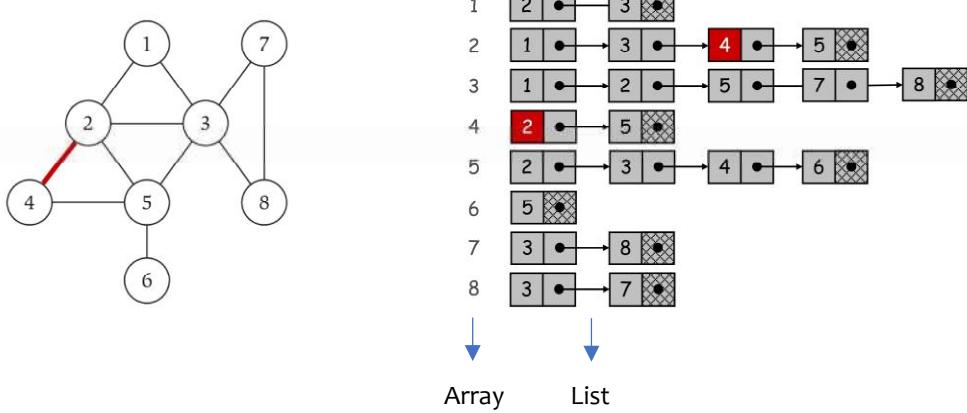
- Dua representasi dari setiap sisi
- Ruang berukuran sebesar n^2
- Memeriksa apakah (u, v) edge membutuhkan waktu $\Theta(1)$
- Mengidentifikasi semua tepi membutuhkan $\Theta(n^2)$ waktu



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Representasi Graf dengan Adjacency List Adjacency List: node diindeks sebagai list

- Dua representasi untuk setiap sisi
- Ukuran ruang $m + n$
- Memeriksa apakah (u, v) edge membutuhkan $O(\deg(u))$. Degree = jumlah tetangga u.
- Mengidentifikasi semua tepi membutuhkan $\Theta(m + n)$.

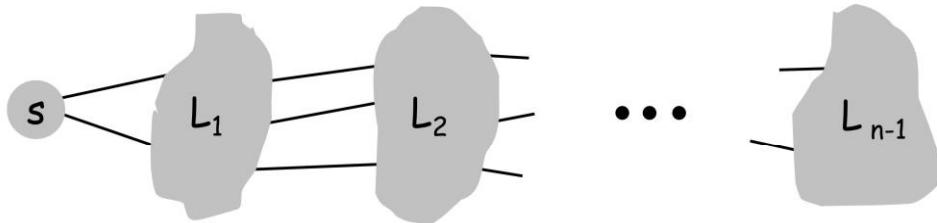


Breadth First Search

Intuisi BFS. Menjelajahi alur keluar dari s ke semua arah yang mungkin, tambahkan node satu "layer" sekaligus.

Algoritma BFS

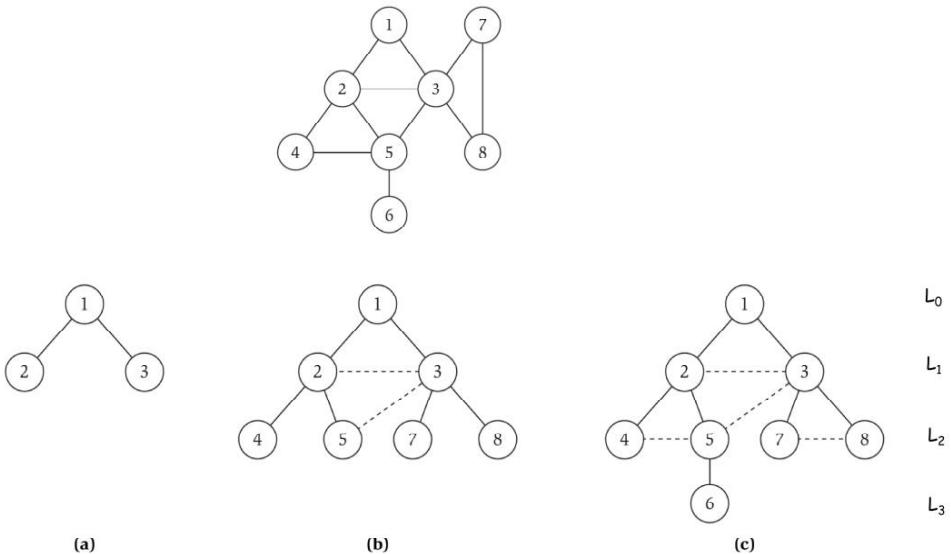
- $L = \{ \}$
- $L =$ semua tetangga dari s
- $L = L + 1$ atau L , dan yang mempunyai edge ke sebuah node di L
- $+ 1 =$ semua node yang bukan milik layer sebelumnya, dan yang memiliki edge ke node di L



Gambar 1. Ilustrasi algoritma BFS

Teorema 1.0

Untuk setiap t , L_t terdiri dari semua node pada jarak tepat t ke s . Ada path dari s ke t jika t muncul di beberapa layer.



Gambar 2. Ilustrasi pembentukan tree BFS dari undirected Graf

Implementasi BFS dalam Koding Program

- Adjacency list adalah representasi struktur data paling ideal untuk BFS
 - Algoritma memeriksa setiap ujung yang meninggalkan node satu per satu. Ketika kita memindai edge yang meninggalkan u dan mencapai $\text{edge}(u, v)$, kita perlu tahu apakah node v telah ditemukan sebelumnya oleh pencarian.
 - Untuk menyederhanakan ini, kita maintain array yang ditemukan dengan panjang n dan mengatur $\text{Discovered}[v] = \text{true}$ segera setelah pencarian kita pertama kali melihat v . Algoritma BFS membangun lapisan node L_1, L_2, \dots , di mana L_i adalah set node pada jarak i dari sumber .
 - Untuk mengelola node dalam layer , kami memiliki daftar $[]$ untuk setiap $= 0, 1, 2, \dots$

BFS(s);

```

Set Discovered[s] = true and Discovered[v] = false for all other  $v$ 
Initialize  $L[0]$  to consist of the single element  $s$ 
Set the layer counter  $i=0$ 
Set the current BFS tree  $T=\emptyset$ 
While  $L[i]$  is not empty
    Initialize an empty list  $L[i+1]$ 
    For each node  $u \in L[i]$ 
        Consider each edge  $(u,v)$  incident to  $u$ 
        If  $\text{Discovered}[v] = \text{false}$  then
            Set  $\text{Discovered}[v] = \text{true}$ 
            Add edge  $(u,v)$  to the tree  $T$ 
            Add  $v$  to the list  $L[i+1]$ 
        Endif
    Endfor
    Increment the layer counter  $i$  by one
Endwhile

```

Depth First Search

Algoritma BFS muncul, khususnya, sebagai cara tertentu mengurutkan node yang kita kunjungi — dalam lapisan berurutan, berdasarkan pada jarak node lain dari s . Metode alami lain untuk menemukan node yang

dapat dijangkau dari s adalah pendekatan yang mungkin Anda dilakukan jika grafik G benar-benar sebuah labirin dari kamar yang saling berhubungan dan kita berjalan-jalan di dalamnya.

Kita akan mulai dari s dan mencoba edge pertama yang mengarah ke node v . Kita kemudian akan mengikuti edge pertama yang mengarah keluar dari v , dan melanjutkan dengan cara ini sampai kita mencapai "jalan buntu" —sebuah node di mana Anda sudah menjelajahi semua tetangganya. Kita kemudian akan mundur sampai kita mencapai node dengan tetangga yang belum diperlakukan, dan melanjutkan dari sana. Kita menyebutnya Depth-first search (DFS), karena ini mengeksplorasi G dengan masuk sedalam mungkin dan hanya mundur jika diperlukan.

DFS juga merupakan implementasi khusus dari algoritma component-growing generik yang dijelaskan sebelumnya. Kita dapat memulai DFS dari titik awal mana pun tetapi mempertahankan pengetahuan global tentang node yang telah dieksplorasi.

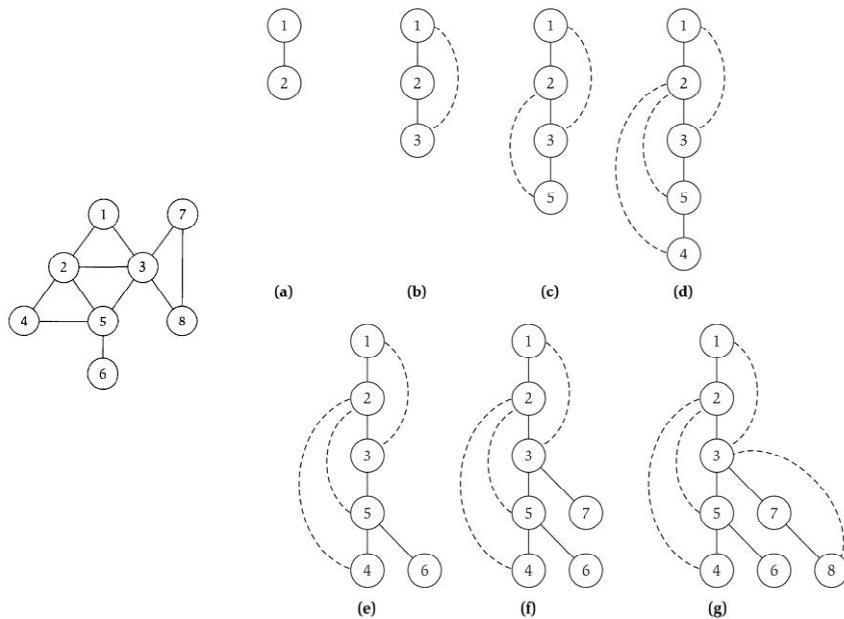
```

DFS( $u$ ):
    Mark  $u$  as "Explored" and add  $u$  to  $R$ 
    For each edge  $(u, v)$  incident to  $u$ 
        If  $v$  is not marked "Explored" then
            Recursively invoke DFS( $v$ )
        Endif
    Endfor

```

Untuk menerapkan ini pada problem konektivitas $s-t$, kita cukup mendeklarasikan semua node pada awalnya untuk tidak dieksplorasi, dan memanggil DFS (s).

Ada beberapa kesamaan dan beberapa perbedaan mendasar antara DFS dan BFS. Kesamaan didasarkan pada fakta bahwa mereka berdua membangun komponen terhubung yang mengandung s , dan bahwa mereka mencapai tingkat efisiensi yang serupa secara kualitatif. Sementara DFS akhirnya mengunjungi set node yang sama persis seperti BFS, ia biasanya melakukannya dalam urutan yang sangat berbeda; menyelidiki jalan panjang, berpotensi menjadi sangat jauh dari s , sebelum membuat cadangan untuk mencoba lebih dekat node yang belum diperlakukan.



Gambar 3. Ilustrasi pembentukan tree DFS dari undirected graph

Implementasi BFS dalam Koding Program

Implementasi DFS paling ideal adalah dengan menggunakan stack. Adapun algoritma DFS dengan stack adalah sebagai berikut:

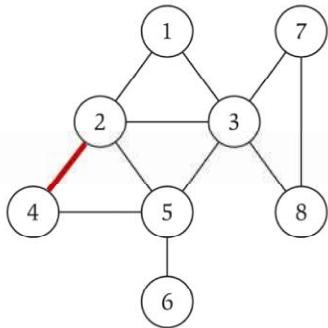
```

DFS(s):
    Initialize S to be a stack with one element s
    While S is not empty
        Take a node u from S
        If Explored[u] = false then
            Set Explored[u] = true
            For each edge (u,v) incident to u
                Add v to the stack S
        Endfor
    Endif
Endwhile

```

Tugas Anda

1. Dengan menggunakan *undirected graph* dan *adjacency matrix* berikut, buatlah koding programnya menggunakan bahasa C++.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

```

/*
Nama      : Hadiza Cahya Firdaus
Npm       : 140810180042
Kelas     : B
Tanggal   : 7 April 2020
Deskripsi : Adjacency Matrix
*/

```

```

#include <iostream>
#include <cstdlib>
using namespace std;
#define MAX 20

class AdjacencyMatrix
{
private:
    int n;
    int **adj;
    bool *visited;
public:
    AdjacencyMatrix(int n)
    {
        this->n = n;
        visited = new bool [n];
        adj = new int* [n];
    }
}
```

```

for (int i = 0; i < n; i++)
{
    adj[i] = new int [n];
    for(int j = 0; j < n; j++)
    {
        adj[i][j] = 0;
    }
}

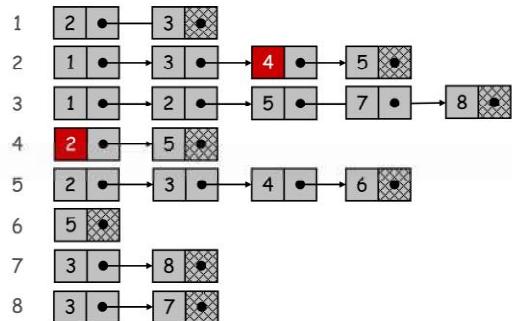
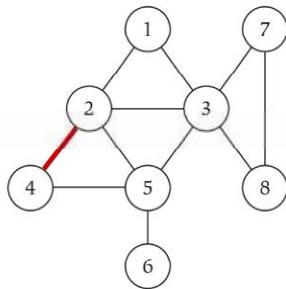
void add_edge(int origin, int destin)
{
    if( origin > n || destin > n || origin < 0 || destin < 0)
    {
        cout<<"Invalid edge!\n";
    }
    else
    {
        adj[origin - 1][destin - 1] = 1;
    }
}

void display()
{
    int i,j;
    for(i = 0;i < n;i++)
    {
        for(j = 0; j < n; j)
            cout<<adj[i][j]<<" ";
        cout<<endl;
    }
};

int main()
{
    int nodes, max_edges, origin, destin;
    cout<<"Enter number of nodes: ";
    cin>>nodes;
    AdjacencyMatrix am(nodes);
    max_edges = nodes * (nodes - 1);
    for (int i = 0; i < max_edges; i++)
    {
        cout<<"Enter edge (-1 -1 to exit): ";
        cin>>origin>>destin;
        if((origin == -1) && (destin == -1))
            break;
        am.add_edge(origin, destin);
    }
    am.display();
    return 0;
}

```

2. Dengan menggunakan *undirected graph* dan representasi *adjacency list*, buatlah koding programnya menggunakan bahasa C++.



```
/*
Nama      : Hadiza Cahya Firdaus
Npm       : 140810180042
Kelas     : B
Tanggal   : 7 April 2020
Deskripsi : Adjacency List
*/
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};

struct AdjList
{
    struct AdjListNode *head;
};

class Graph
{
private:
    int V;
    struct AdjList* array;
public:
    Graph(int V)
    {
        this->V = V;
        array = new AdjList [V];
        for (int i = 0; i < V; ++i)
            array[i].head = NULL;
    }

    AdjListNode* newAdjListNode(int dest)
    {
```

```

AdjListNode* newNode = new AdjListNode;
newNode->dest = dest;
newNode->next = NULL;
return newNode;
}

void addEdge(int src, int dest)
{
    AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = array[src].head;
    array[src].head = newNode;
    newNode = newAdjListNode(src);
    newNode->next = array[dest].head;
    array[dest].head = newNode;
}

void printGraph()
{
    int v;
    for (v = 1; v < V; ++v)
    {
        AdjListNode* pCrawl = array[v].head;
        cout<<"\n Adjacency list of vertex "<<v<<"\n head ";
        while (pCrawl)
        {
            cout<<"-> "<<pCrawl->dest;
            pCrawl = pCrawl->next;
        }
        cout<<endl;
    }
}

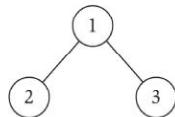
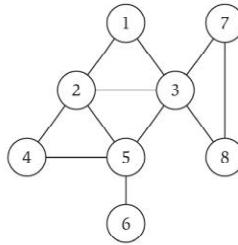
int main()
{
    Graph gh(8);
    gh.addEdge(1, 2);
    gh.addEdge(1, 3);
    gh.addEdge(2, 4);
    gh.addEdge(2, 5);
    gh.addEdge(2, 3);
    gh.addEdge(3, 7);
    gh.addEdge(3, 8);
    gh.addEdge(4, 5);
    gh.addEdge(5, 3);
    gh.addEdge(5, 6);
    gh.addEdge(7, 8);

    // print the adjacency list representation of the above graph
    gh.printGraph();

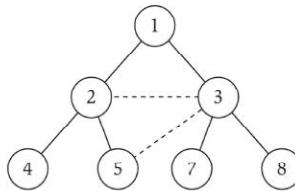
    return 0;
}

```

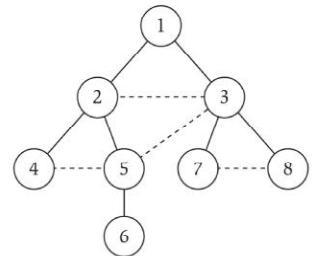
3. Buatlah program Breadth First Search dari algoritma BFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree BFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big- Θ !



(a)



(b)



(c)

L_0
 L_1
 L_2
 L_3

```
/*
Nama      : Hadiza Cahya Firdaus
Npm       : 140810180042
Kelas     : B
Tanggal   : 7 April 2020
Deskripsi : BFS
*/
```

```
#include<iostream>
using namespace std;

int main(){
    int vertexSize = 8;
    int adjacency[8][8] = {
        {0,1,1,0,0,0,0,0},
        {1,0,1,1,1,0,0,0},
        {1,1,0,0,1,0,1,1},
        {0,1,0,0,1,0,0,0},
        {0,1,1,1,0,1,0,0},
        {0,0,0,0,1,0,0,0},
        {0,0,1,0,0,0,0,1},
        {0,0,1,0,0,0,1,0}
    };

    bool discovered[vertexSize];
    for(int i = 0; i < vertexSize; i++){
        discovered[i] = false;
    }
    int output[vertexSize];

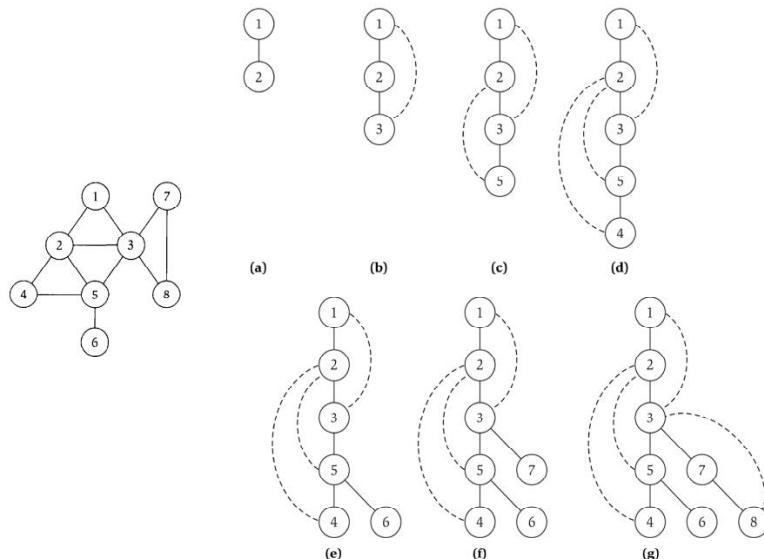
    discovered[0] = true;
    output[0] = 1;
```

```

int counter = 1;
for(int i = 0; i < vertexSize; i++){
    for(int j = 0; j < vertexSize; j++){
        if((adjacency[i][j] == 1)&&(discovered[j] == false)){
            output[counter] = j+1;
            discovered[j] = true;
            counter++;
        }
    }
}
cout<<"BFS : "<<endl;
for(int i = 0; i < vertexSize; i++){
    cout<<output[i]<<" ";
}
}

```

4. Buatlah program Depth First Search dari algoritma DFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree DFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big- Θ !



```

/*
Nama      : Hadiza Cahya Firdaus
Npm       : 140810180042
Kelas     : B
Tanggal   : 7 April 2020
Deskripsi : DFS
*/

```

```
#include <iostream>
```

```
#include <list>
```

```
using namespace std;
```

```
class Graph{
    int N;
```

```
list<int> *adj;
```

```

void DFSUtil(int u, bool visited[]){
    visited[u] = true;
    cout << u << " ";

    list<int>::iterator i;
    for(i = adj[u].begin(); i != adj[u].end(); i++){
        if(!visited[*i]){
            DFSUtil(*i, visited);
        }
    }
}

public :
Graph(int N){
    this->N = N;
    adj = new list<int>[N];
}

void addEdge(int u, int v){
    adj[u].push_back(v);
}

void DFS(int u){
    bool *visited = new bool[N];
    for(int i = 0; i < N; i++){
        visited[i] = false;
    }
    DFSUtil(u, visited);
}
};

int main(){
    Graph g(8);

    g.addEdge(1,2);
    g.addEdge(1,3);
    g.addEdge(2,3);
    g.addEdge(2,4);
    g.addEdge(2,5);
    g.addEdge(3,7);
    g.addEdge(3,8);
    g.addEdge(4,5);
    g.addEdge(5,3);
    g.addEdge(5,6);
    g.addEdge(7,8);

    cout << "\nDFS Traversal Starts from Node 1" << endl;
    g.DFS(1);

    return 0;
}

```

Teknik Pengumpulan

- Lakukan push ke github/gitlab untuk semua program dan laporan hasil analisa yang berisi jawaban dari pertanyaan-pertanyaan yang diajukan. Silahkan sepakati dengan asisten praktikum.

Penutup

- Ingat, berdasarkan Peraturan Rektor No 46 Tahun 2016 tentang Penyelenggaraan Pendidikan, mahasiswa wajib mengikuti praktikum 100%
- Apabila tidak hadir pada salah satu kegiatan praktikum segeralah minta tugas pengganti ke asisten praktikum
- Kurangnya kehadiran Anda di praktikum, memungkinkan nilai praktikum Anda tidak akan dimasukkan ke nilai mata kuliah.