



T5

# Graph Traversals dan Shortest Path

Dosen: Hafara Firdausi, S.Kom., M.Kom.

Struktur data dan pemrograman berbasis object  
Kelompok 5



Dwiyasa Nakula Michael Wayne Sylvia Febrianti Muhammad Afif Agas Ananta

5027221001

5027221037

5027221019

5027221032

5027221004

# Algoritma Shortest Path

Bellman-Ford



# Algoritma Graph Traversals

Depth-First  
Search (DFS)



# Depth-First Search (DFS)

Algoritma Graph Traversals

Struktur data dan pemrograman berbasis object

Kelompok 5

# TABLE OF CONTENTS

01  
Overview

02  
Konsep

03  
Cara kerja

04  
Implementasi rekrusif

05  
Code

# Apa Itu Depth-First Search (DFS) ?

```
complement(A, B,
display(B, &vertex);

}
25
26 void gen()
27 {
28     int i, j;
29     for (i = 0; i < n; i++)
30     {
31         for (j = 0; j < i; j++)
32         {
33             if ((A[i][j] == 0) && (A[j][i] == 0))
34             {
35                 res = rand() % 2;
36                 A[i][j] = res;
37                 A[j][i] = res;
38             }
39         }
40     }
41 }
```

```
res= rand()%2;
A[i][j]= res;
A[j][i]= res;
```

# 01 OVERVIEW

# INTRODUCTION

DFS adalah algoritma sistematis yang mengeksplorasi struktur tree atau grafik dengan bergerak sejauh mungkin di sepanjang setiap cabang sebelum melakukan backtracking. Ini mengikuti gerakan depth search, menjelajah sejauh mungkin di sepanjang satu cabang sebelum menarik kembali.

# Tujuan

DFS digunakan untuk memeriksa secara sistematis simpul dalam grafik atau Tree dengan menavigasi sedalam mungkin sebelum memeriksa cabang berikutnya, memastikan eksplorasi menyeluruh terhadap keseluruhan struktur.



# Ciri-Ciri

## Tidak Optimal

DFS tidak serta merta menemukan jalur terpendek; sebaliknya, ini berfokus pada eksplorasi menyeluruh.



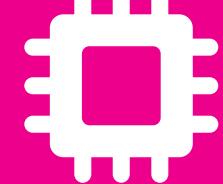
## Menyeluruh

Dalam graf tak berarah, setiap vertex dapat dijangkau dari vertex lainnya, sehingga memastikan eksplorasi menyeluruh.



## Efisiensi Memori

Seringkali memerlukan lebih sedikit memori dibandingkan dengan penelusuran luas, sehingga cocok untuk struktur skala besar.



## Backtracking

Backtracking melibatkan memundurkan suatu pilihan dan membuat pilihan lain; sifat stack DFS secara alami mendukung proses ini



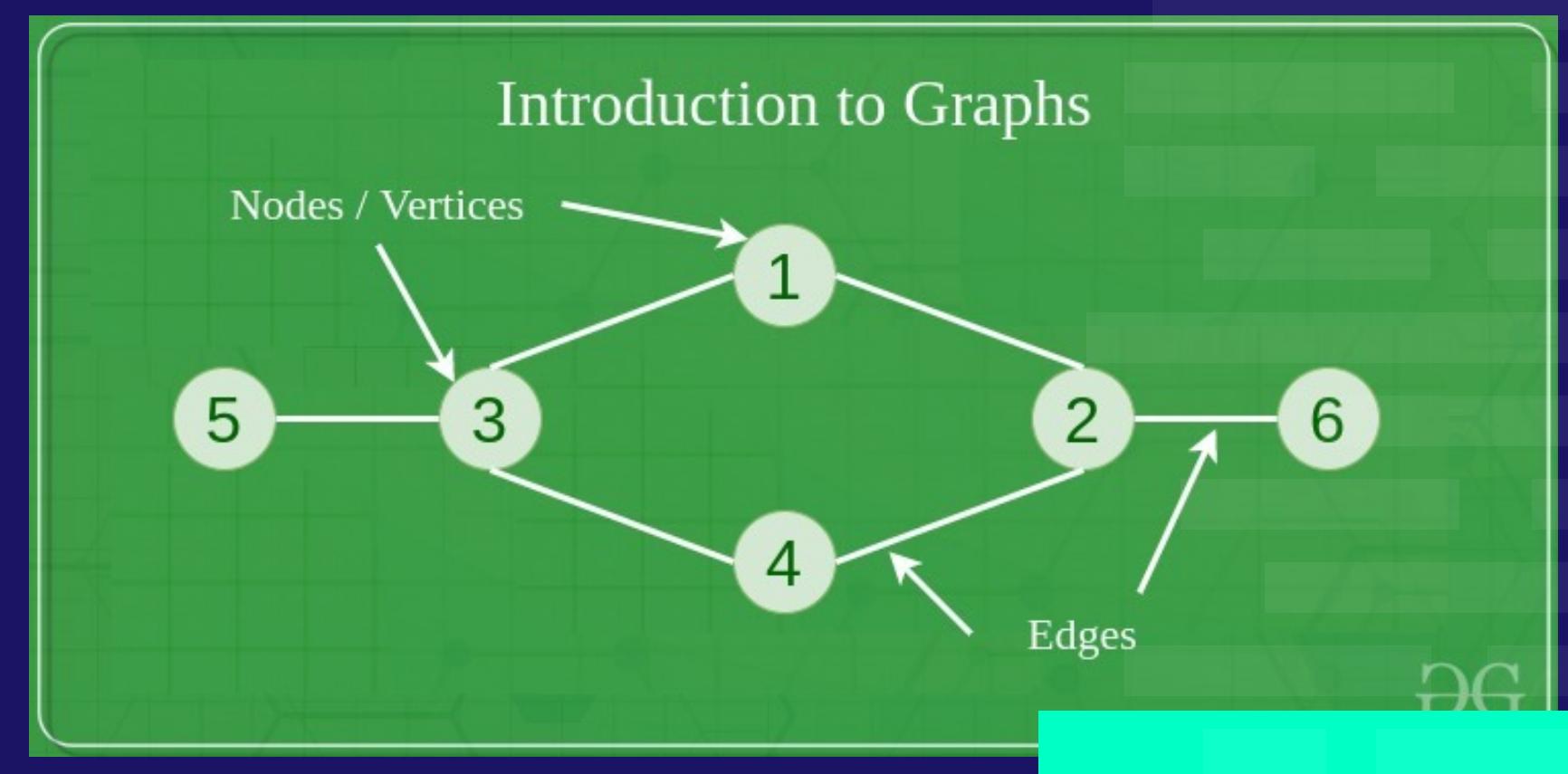
# 02 Konsep

# Graph

Graf adalah kumpulan titik (simpul) dan sisi yang menghubungkan pasangan titik.

## Vertices dan Edges

Vertices mewakili entitas, dan Edges menunjukkan hubungan antara entitas tersebut. Setiap sisi menghubungkan dua simpul dan mungkin memiliki arah (berarah) atau tidak berarah.

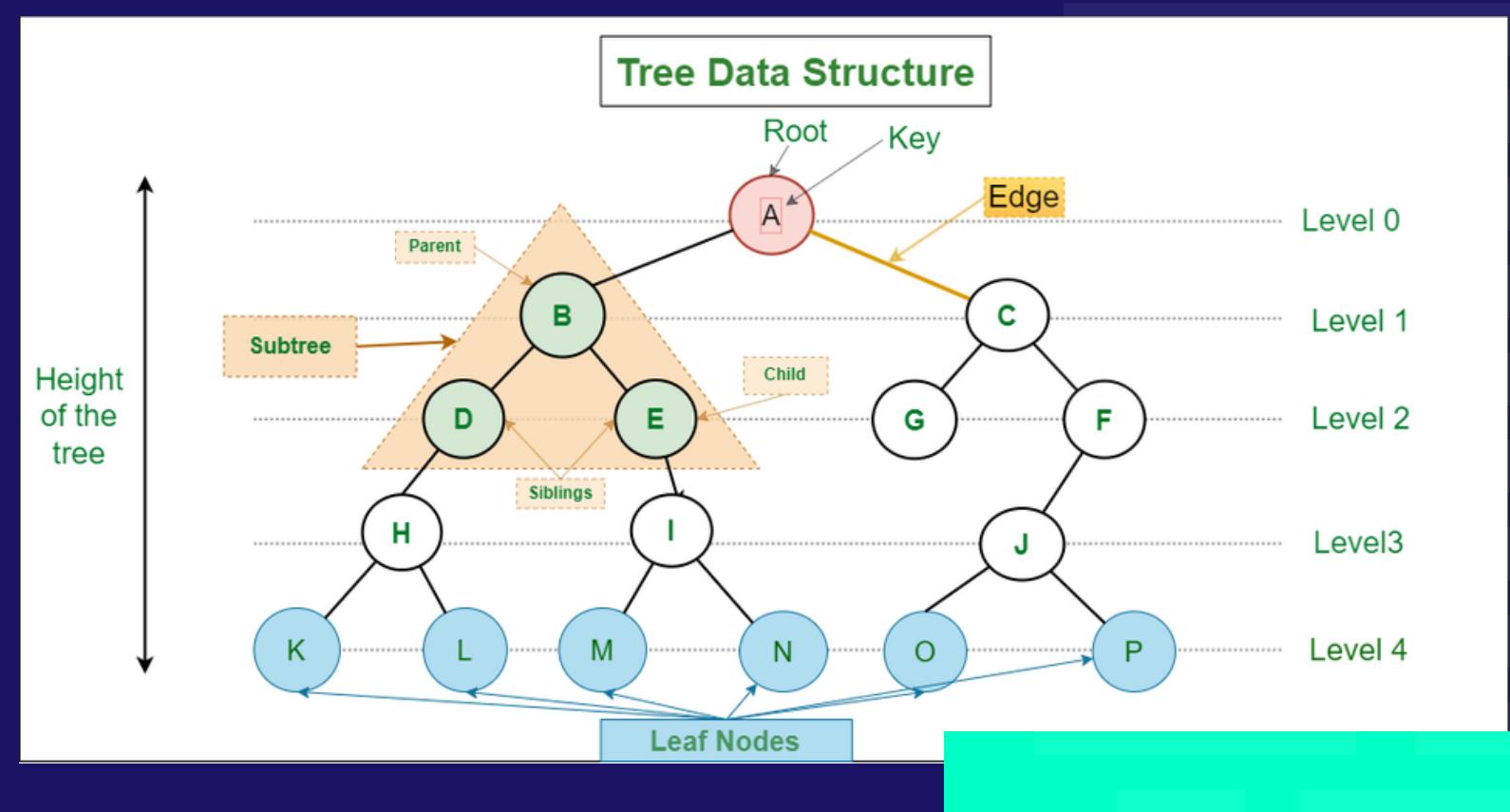


# Tree

Tree adalah struktur hierarki yang terdiri dari node yang dihubungkan oleh edges

## Nodes dan Branches

Node dalam Tree mewakili elemen, dan edges mewakili hubungan antara elemen-elemen tersebut. Node dalam Tree mempunyai hubungan induk-anak, dan setiap node, kecuali root, mempunyai tepat satu edge masuk (dari induknya).

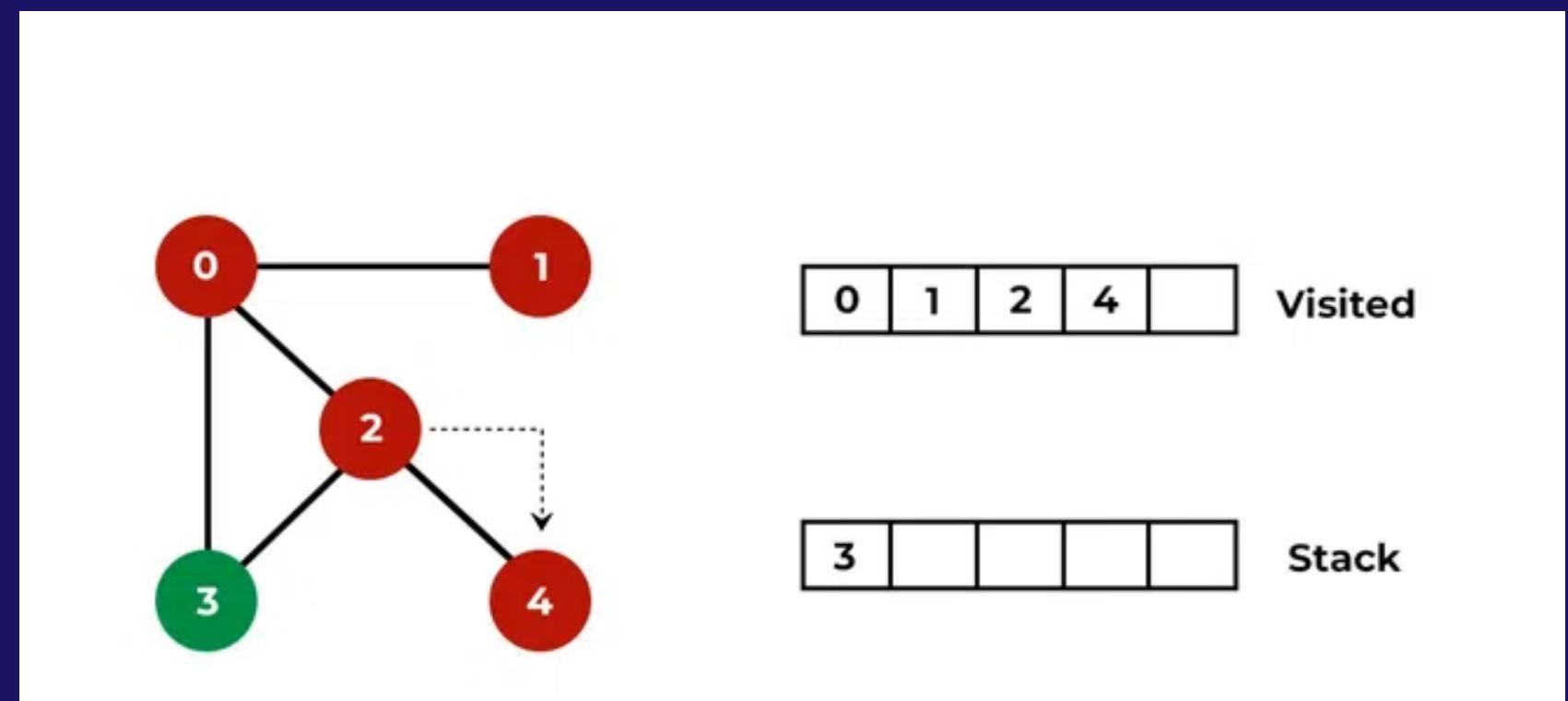


# Visited Nodes

Untuk menghindari perulangan tak terbatas pada grafik cycle, dalam algoritma traversal grafik seperti DFS, perlu dilakukan pelacakan node yang sudah dikunjungi. Hal ini penting untuk mencegah pengunjungan kembali ke node yang sama tanpa batas waktu.

# Tujuan

Hal ini memastikan bahwa algoritme tidak terjebak dalam siklus, sehingga menjamin penghentian dan kebenaran



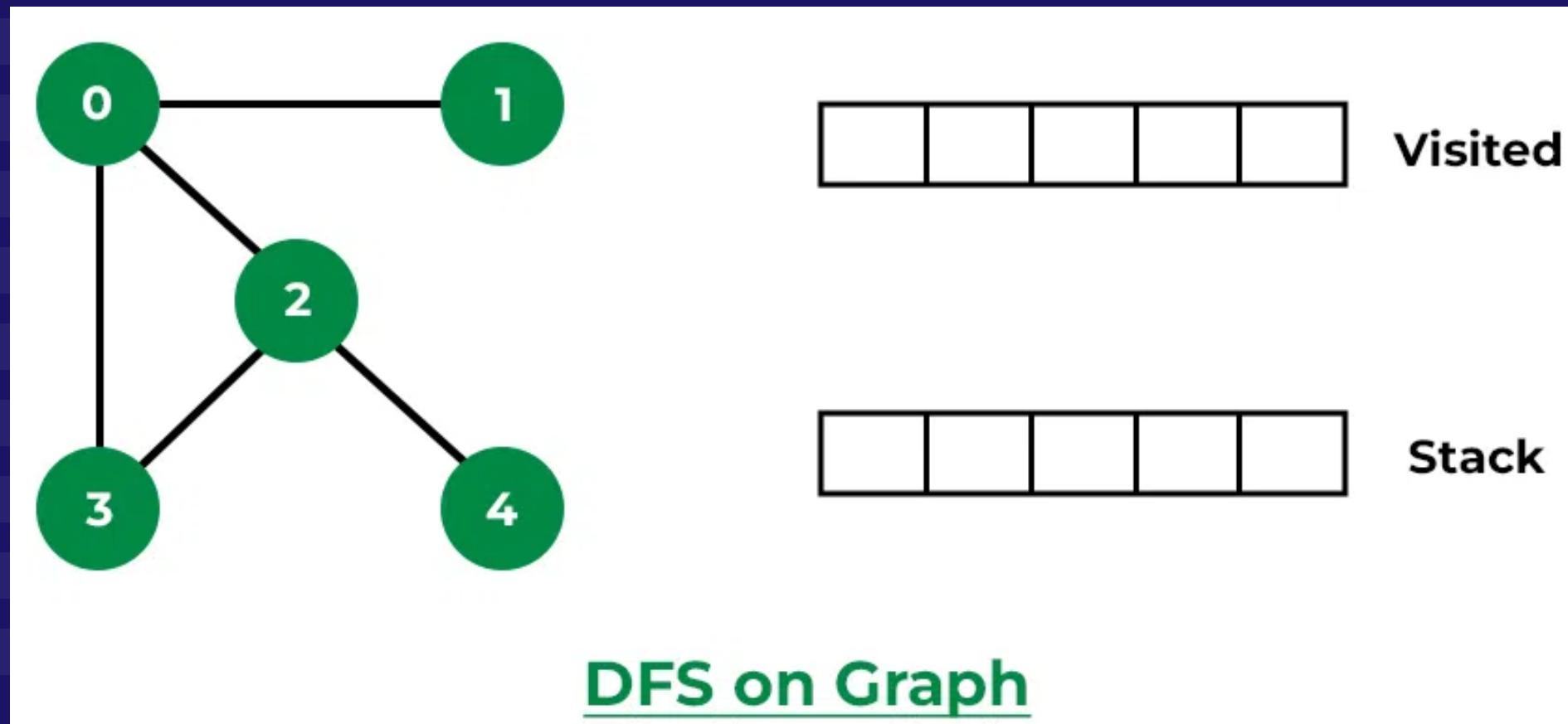
[DFS on Graph](#)

# Preventing Infinite Loops:

- Ketika sebuah node dikunjungi, node tersebut ditandai sebagai "dikunjungi".
- Selama traversal, sebelum menjelajahi node tetangga, algoritme memeriksa apakah node tersebut telah dikunjungi.
- Jika node telah dikunjungi, algoritme akan melewatkannya eksplorasi lebih lanjut untuk mencegah loop tak terbatas.

# 03 Cara Kerja

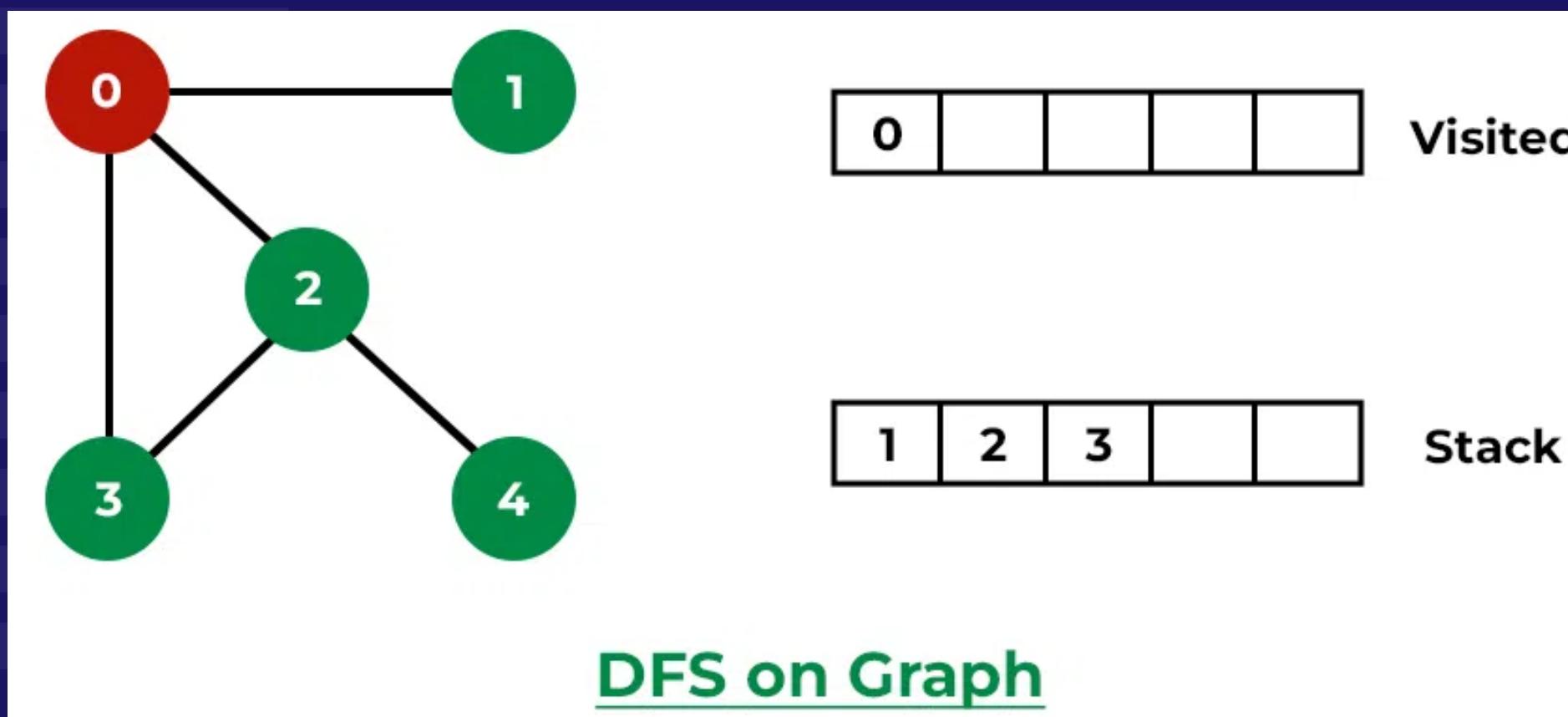
# Step 1



Awalnya Stack dan array yang dikunjungi kosong.



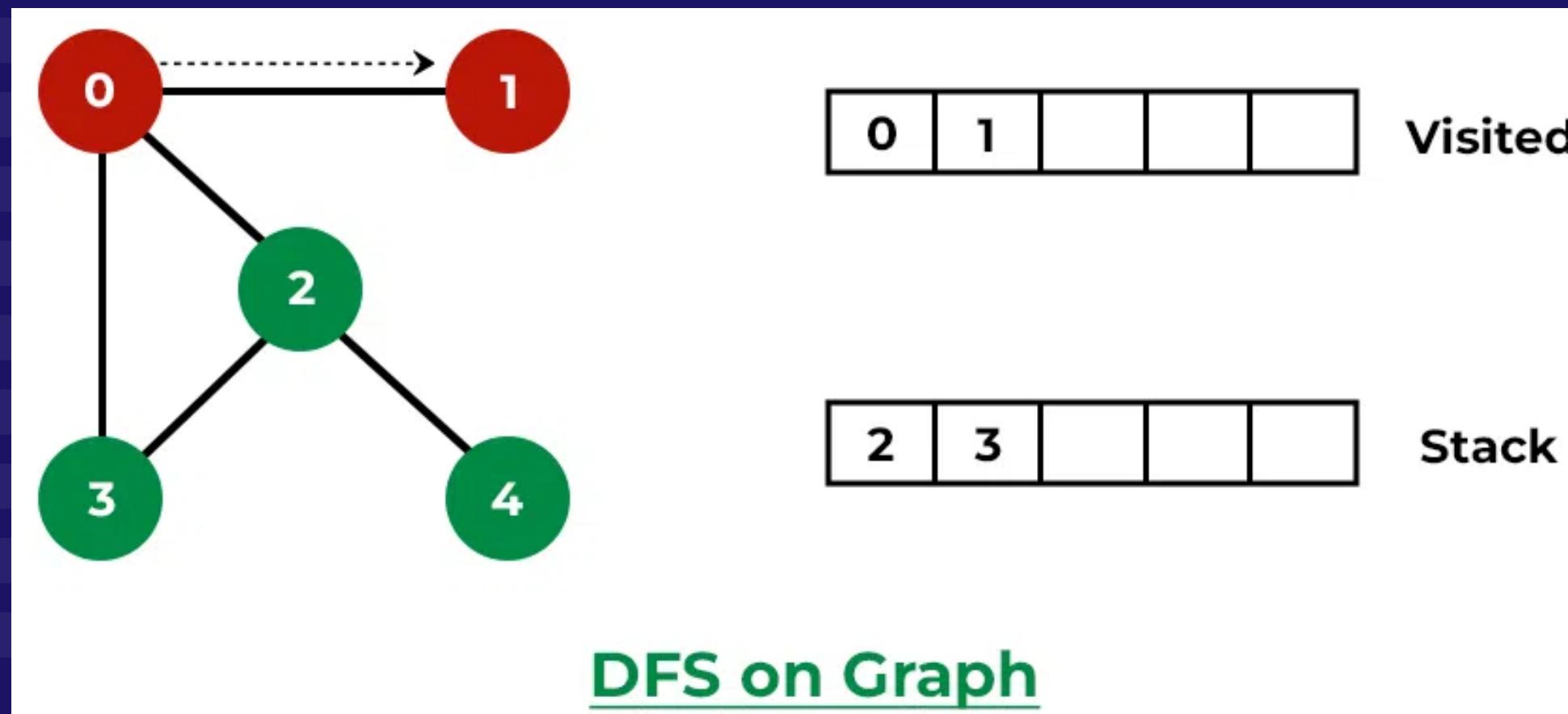
## Step 2



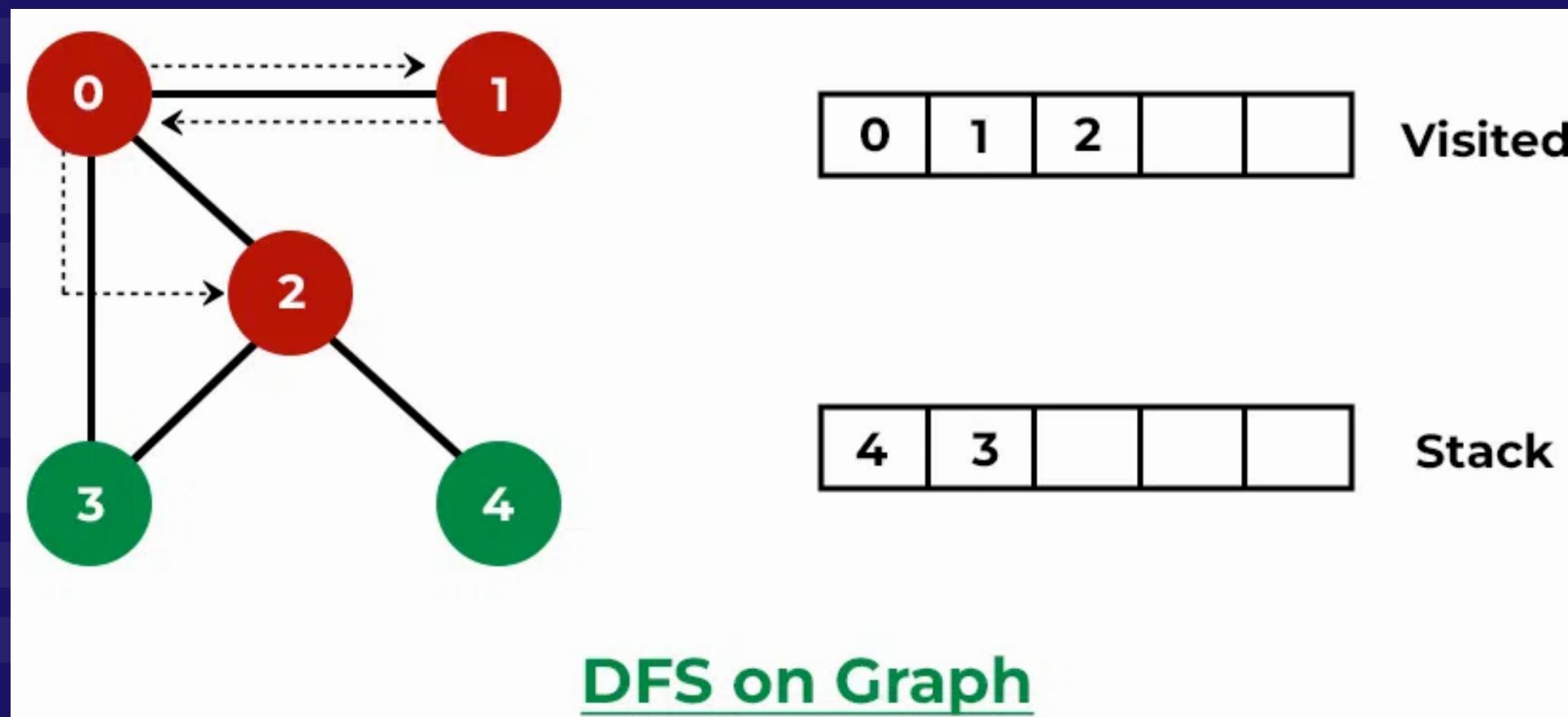
Kunjungi 0 dan masukkan node tetangganya yang belum dikunjungi ke dalam tumpukan.

## Step 3

Sekarang, Node 1 berada di bagian atas tumpukan, jadi kunjungi node 1 dan keluarkan dari tumpukan dan letakkan semua node berdekatan yang tidak dikunjungi ke dalam tumpukan.



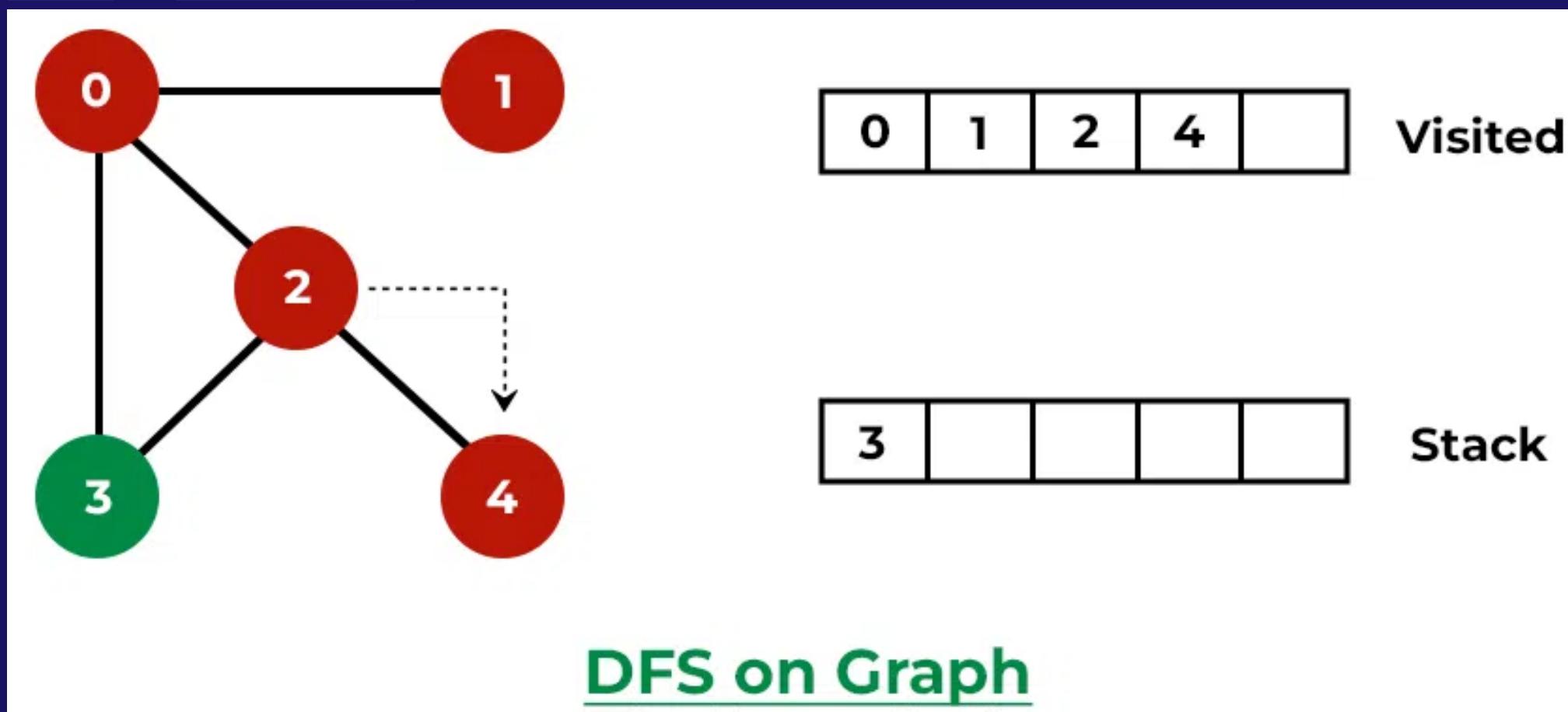
## Step 4



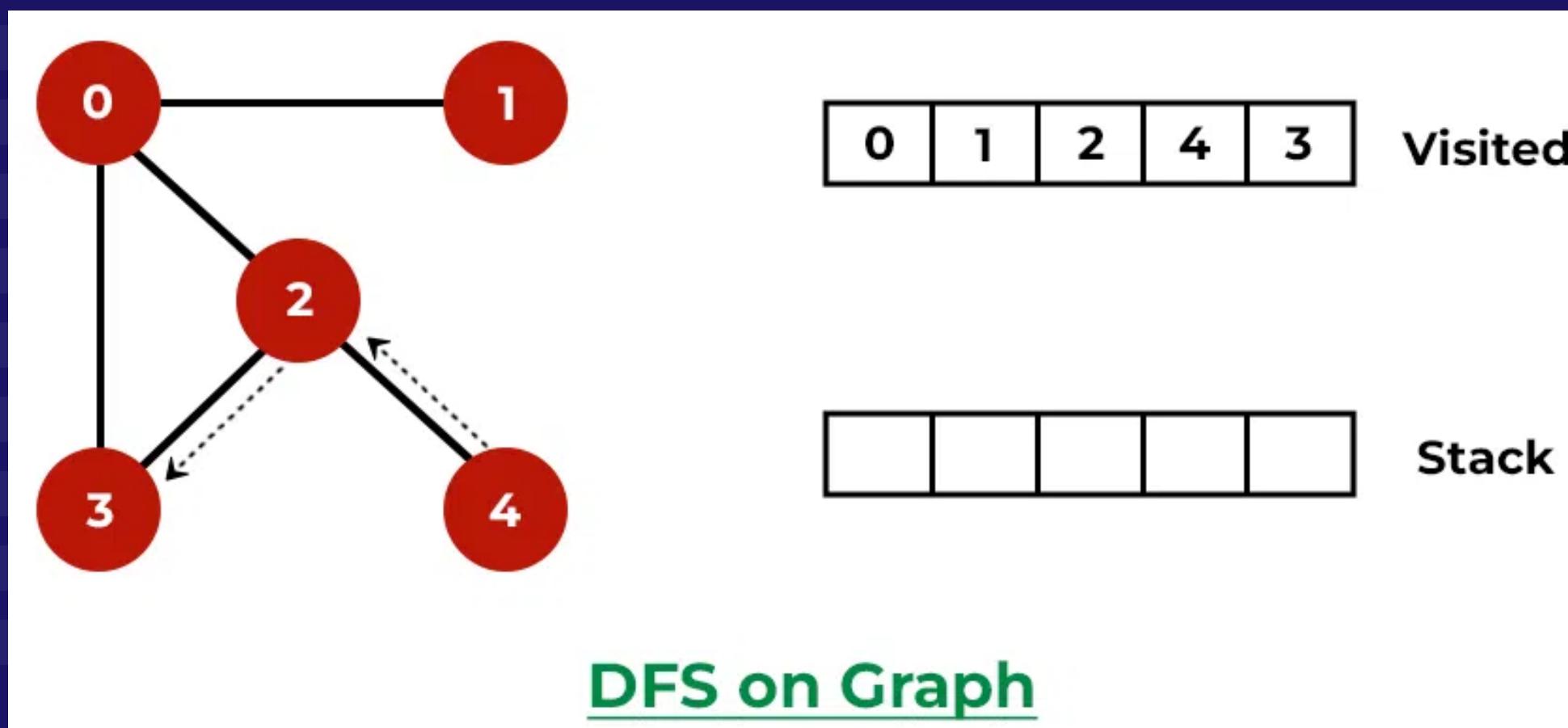
Sekarang, Node 2 berada di bagian atas tumpukan, jadi kunjungi node 2 dan keluarkan dari tumpukan dan letakkan semua node berdekatan yang tidak dikunjungi (yaitu, 3, 4) ke dalam tumpukan

## Step 5

Sekarang, Node 4 berada di bagian atas tumpukan, jadi kunjungi node 4 dan keluarkan dari tumpukan dan letakkan semua node berdekatan yang tidak dikunjungi ke dalam tumpukan.

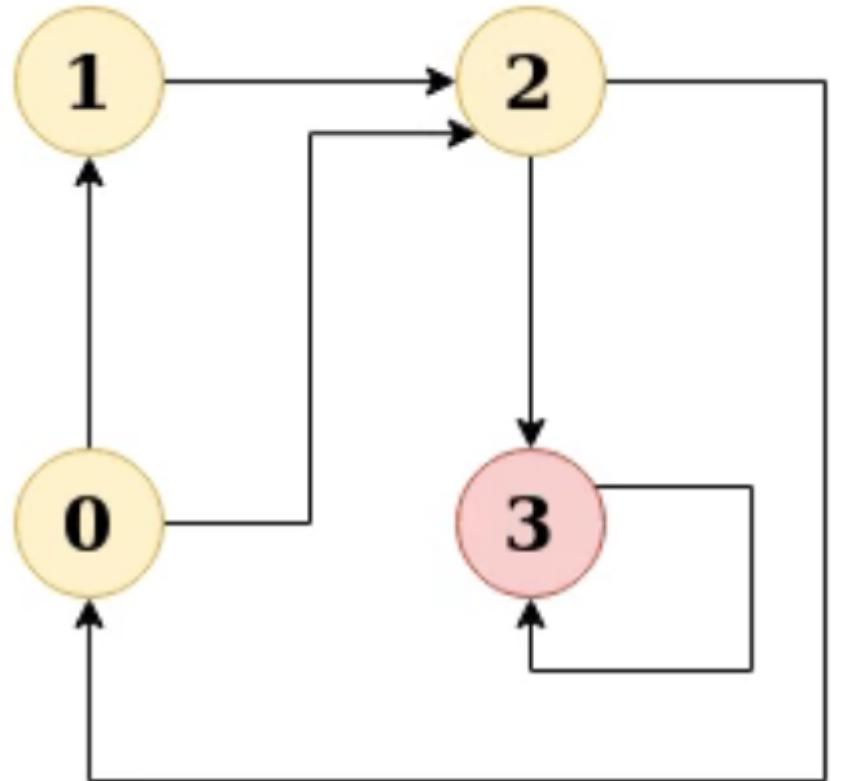


## Step 6



Sekarang, Node 3 berada di bagian atas tumpukan, jadi kunjungi node 3 dan keluarkan dari tumpukan dan masukkan semua node berdekatan yang tidak dikunjungi ke dalam tumpukan

# Contoh



Green is unvisited node.

Red is current node.

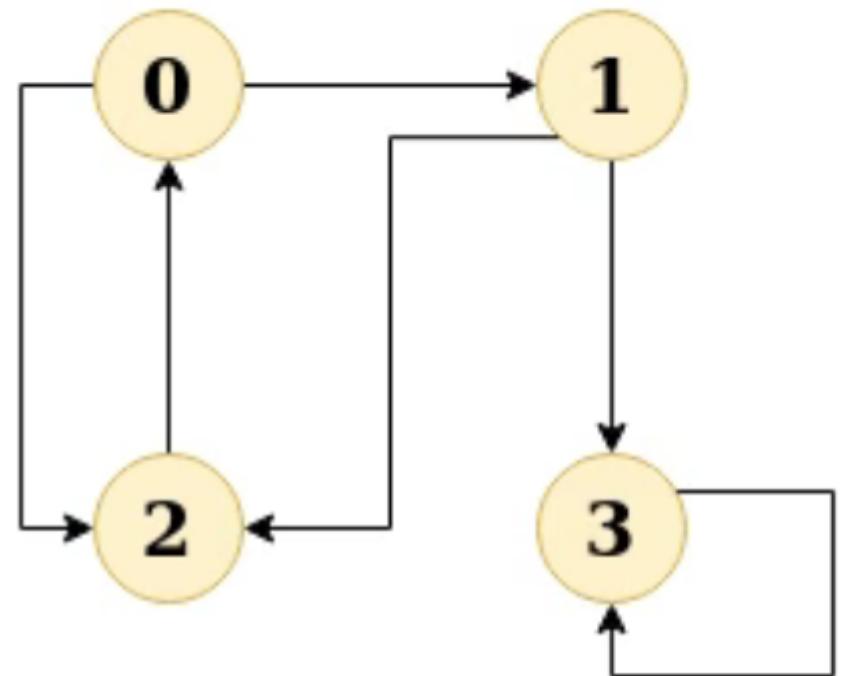
Orange is the nodes in the recursion stack.

Input:  $n = 4, e = 6$

$0 \rightarrow 1, 0 \rightarrow 2, 1 \rightarrow 2, 2 \rightarrow 0, 2 \rightarrow 3, 3 \rightarrow 3$

Output: DFS from vertex 1 : 1 2 0 3

# Contoh



Green is unvisited node.

Red is current node.

Orange is the nodes in the recursion stack.

Input:  $n = 4, e = 6$

$2 \rightarrow 0, 0 \rightarrow 2, 1 \rightarrow 2, 0 \rightarrow 1, 3 \rightarrow 3, 1 \rightarrow 3$

Output: DFS from vertex 2 : 2 0 1 3

# 04

# Time and Space Complexity

## Auxiliary Space

$O(V + E)$ , karena array ekstra yang dikunjungi dengan ukuran  $V$  diperlukan, Dan ukuran tumpukan untuk panggilan berulang ke fungsi DFS



## Time complexity

$O(V + E)$ , dengan  $V$  adalah jumlah simpul dan  $E$  adalah jumlah sisi pada grafik

# 05

# CODE

# Code - Bagian 1

## Representasi Graph

```
//Representasi weighted Edge
struct Edge
{
    int src, dest, weight;
};

//Graph menggunakan struct
struct Graph
{
    int V, E;
    struct Edge *edge;
};

//Fungsi untuk membuat Graph
struct Graph *createGraph(int V, int E)
{
    struct Graph *graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
```

# Code - Bagian 2

## Fungsi Visiting DFS

```
void visiting(struct Graph *graph, int aw, vector<bool>
&visited)
{
    cout << aw << " ";
    for(int i=0; i<graph->E; i++) {
        if(graph->edge[i].src == aw && !visited[graph-
>edge[i].dest]) {
            visited[graph->edge[i].dest] = true;
            visiting(graph, graph->edge[i].dest, visited);
        }
    }
}
```

# Code - Bagian 3

## Fungsi Utama DFS

```
void DFS(struct Graph *graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    vector<bool> visited(V);

    //declare that no vertex has been visited
    for(int i=0; i<V; i++) visited[i] = false;
    visited[src] = true;

    cout << "DFS(Depth-First Search)\nSource: " << src <<
endl;
    visiting(graph, src, visited);
    cout << endl << endl;
}
```

</>

# Bellman-Ford

Algoritma Shortest Path

Struktur data dan pemrograman berbasis object  
Kelompok 5

# TABLE OF CONTENTS

01

Introduction

02

Konsep & Cara Kerja

03

Pseudocode

04

Time Complexity

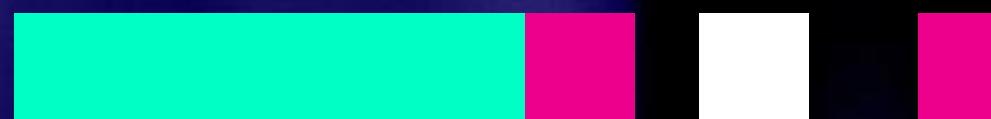
05

Aplikasi Algoritma Bellman Ford

06

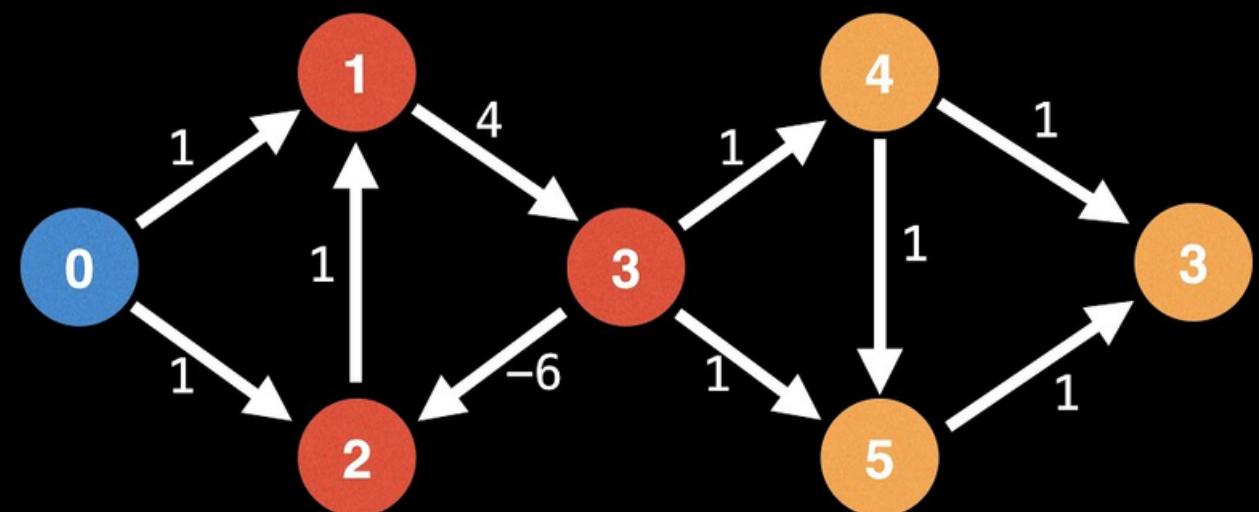
Penjelasan Kode

# APA ITU BELLMAN-FORD?



# BELLMAN-FORD

## Bellman-Ford Shortest Path Algorithm



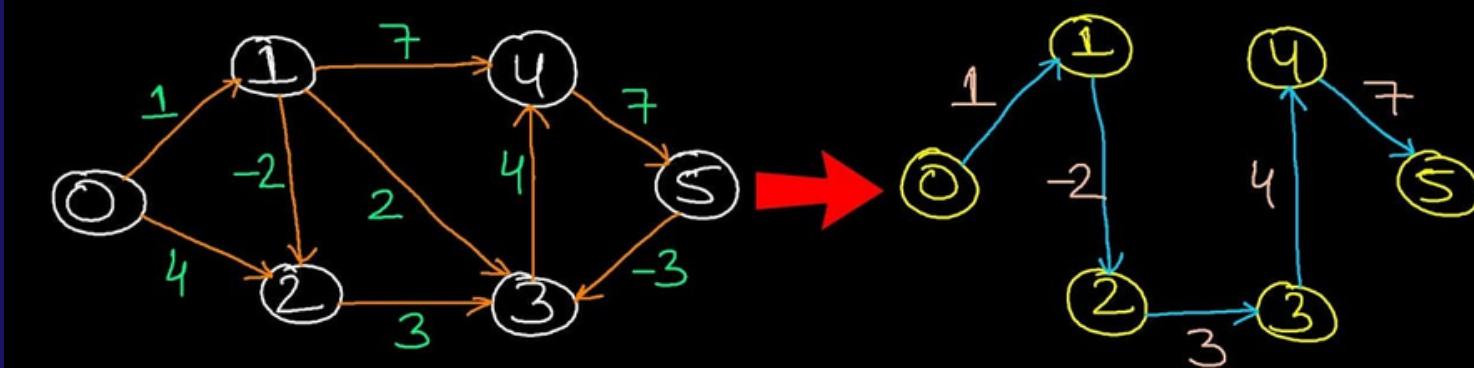
- Algoritma Bellman-Ford adalah **algoritma untuk mencari rute terpendek** dari **satu titik (vertex)** ke **semua titik lain** dalam sebuah graf berbobot.
- Algoritma ini ditemukan oleh Richard E. Bellman dan Lester R. Ford Jr. pada tahun 1958.

# KONSEP DAN CARA KERJA

# Konsep Bellman-Ford

- Algoritma ini **mencari jalur terpendek yang iteratif, mengupdate perkiraan jarak terpendek dari titik awal ke semua titik** dalam graf dengan melakukan iterasi pada semua sisi.
- Meskipun lebih lambat dari Dijkstra, algoritma ini **lebih serbaguna karena dapat menangani bobot negatif**.
- Dalam menghitung jarak terpendek, **Bellman-Ford berlaku untuk graf berbobot**, sementara Dijkstra lebih cepat asalkan tidak ada bobot negatif pada sisi.

## Bellman Ford Algorithm



# Mekanisme Kerja Bellman-Ford

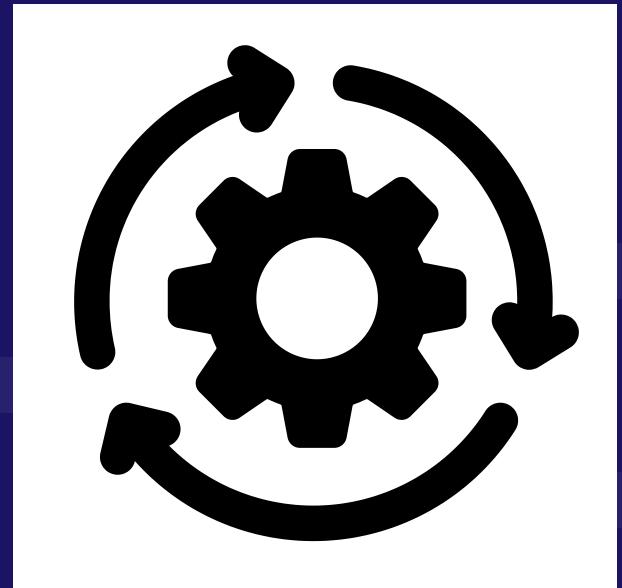
## Inisiasi

**Menetapkan perkiraan jarak terpendek** dari titik awal ke semua titik lain sebagai nilai infinity, kecuali titik awal yang diatur sebagai nol.



## Iterasi Relaksasi

**Memeriksa setiap sisi graf, memperbarui perkiraan jarak terpendek** dari titik awal ke tujuan jika menemukan jalur lebih pendek.



## Terminasi

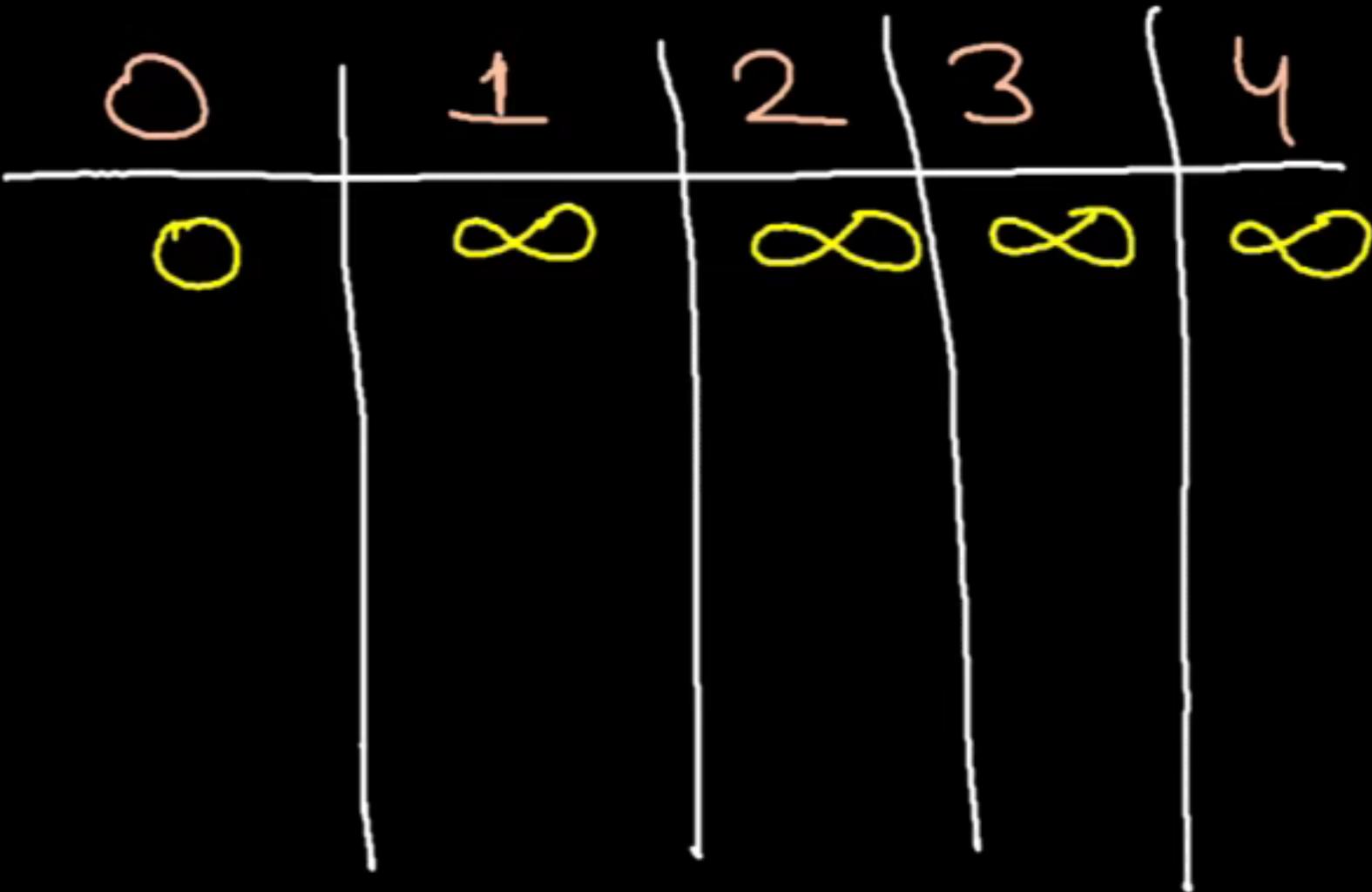
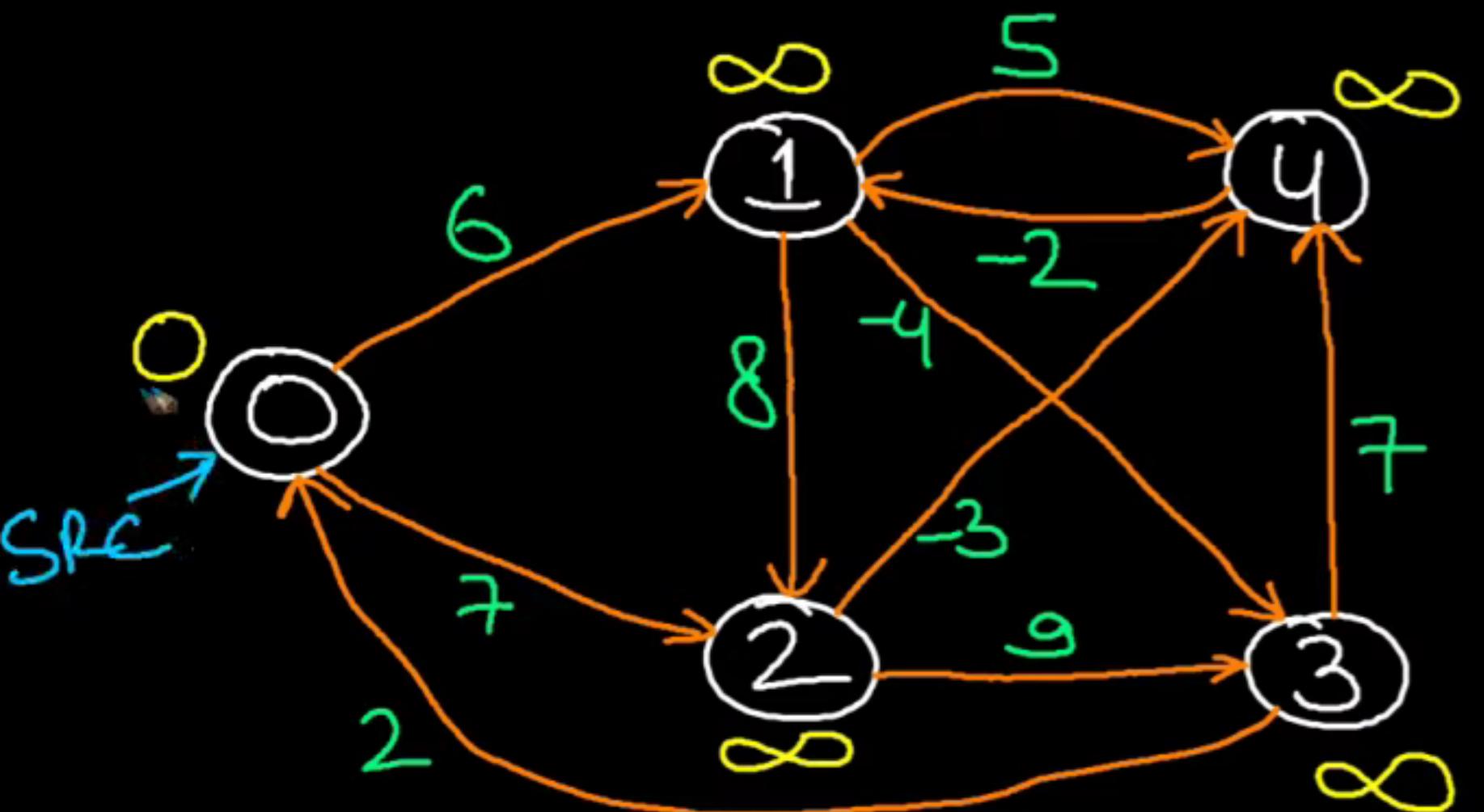
**Algoritma berhenti saat tidak ada lagi perkiraan jarak yang dapat diperbarui**, menandakan penemuan jalur terpendek untuk semua titik.



# Bellman Ford Pseudocode

# Konsep Bellman-Ford

1. Initialize all vertices at  $\text{dist} = \infty$  except  $\text{source}[0]$
2. Relax all  $E$  edges  $(V-1)$  times.  
If  $d[u] + \text{cost} < d[v] \rightarrow d[v] = d[u] + \text{cost}$   
else skip
3. Relax once more.  
If we find a new shortest path for any mode, then we have -VE edge cycle  
else we don't



Parent

0	1	2	3	4

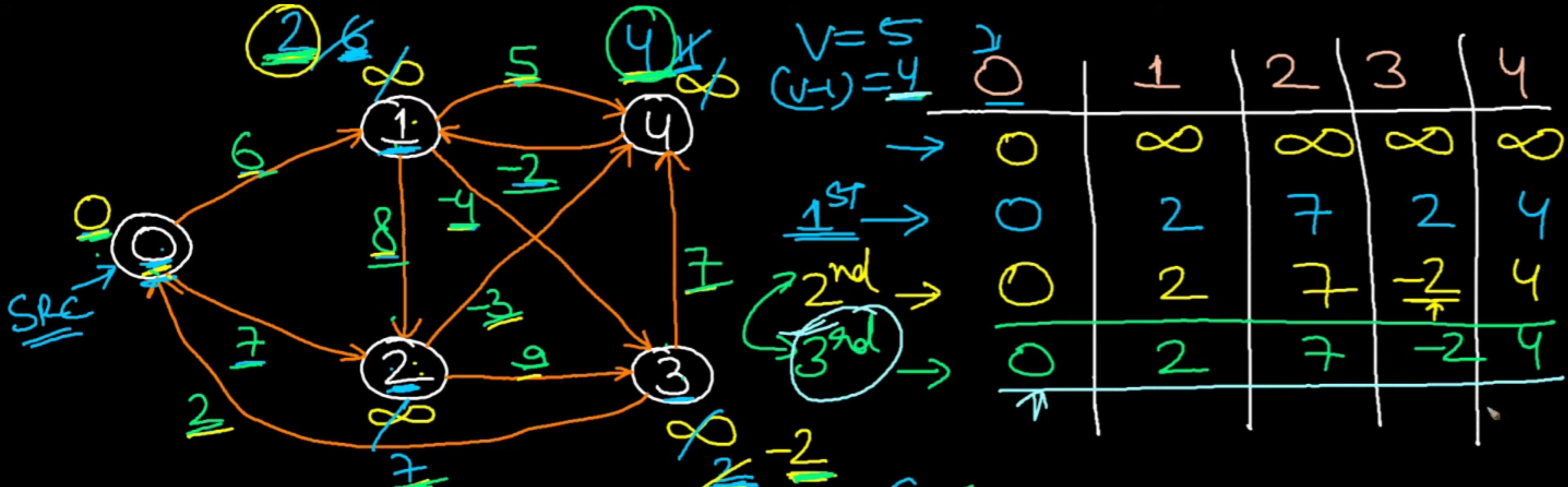
Cost

--	--	--	--	--

(0,1)  
(0,2)  
(1,2)  
(1,3)

(1,4)  
(2,3)  
(2,4)  
(3,4)

(3,0)  
(4,1)



Parent

	0	1	2	3	4
0	-1	∅	0	1	1/2

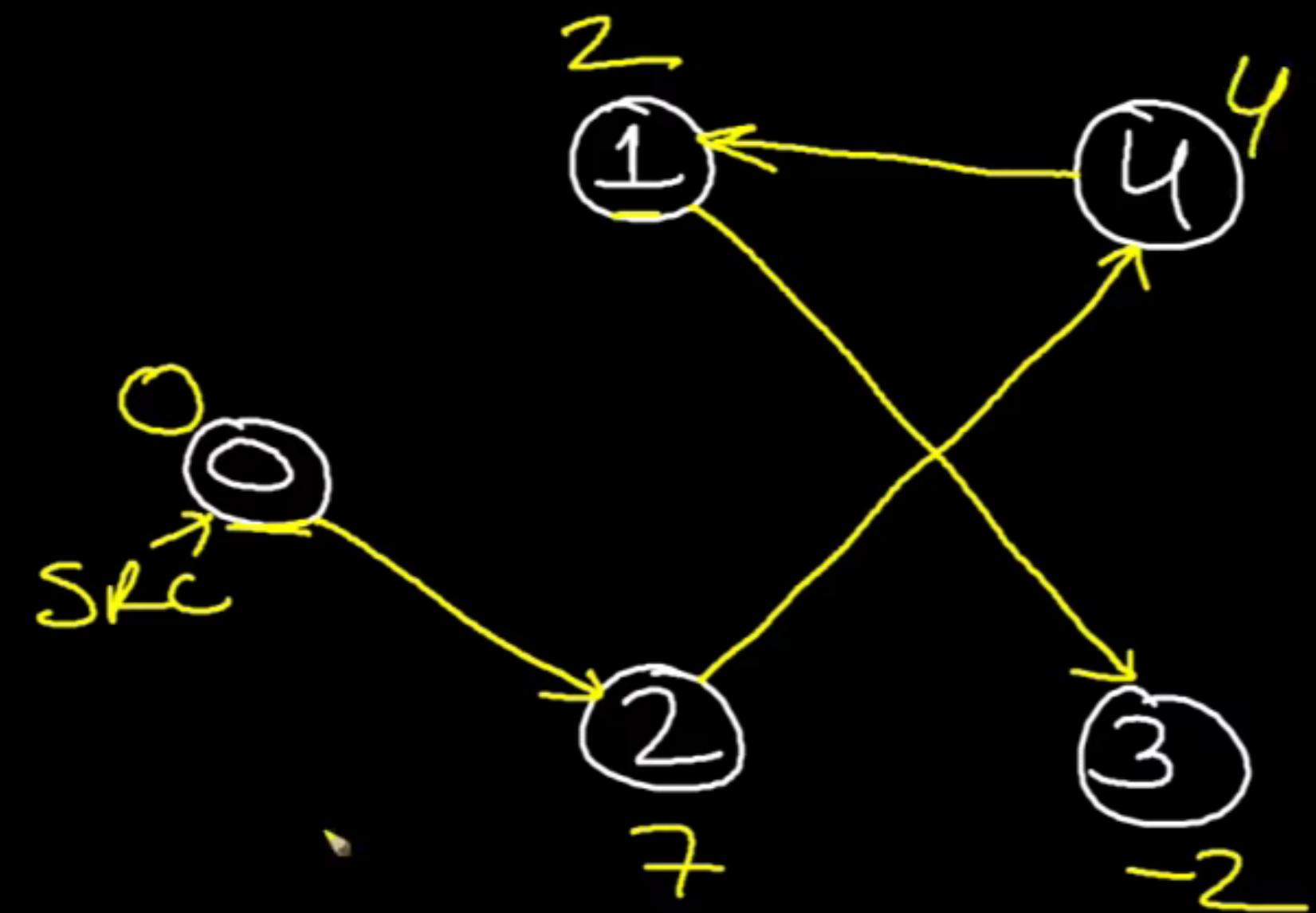
Cost

	0	1	2	3	4
0	0	6	7	2	8

- Set of states:
- (0, 1)
  - (0, 2)
  - (1, 2)
  - (1, 3)
  - (1, 4)
  - (2, 3)
  - (2, 4)
  - (3, 0)
  - (3, 1)
  - (3, 4)

# PRINT SINGLE SOURCE SHORTEST PATH GRAPH

0	1	2	3	4	
<u>Parent</u>	-1	4	0	1	2
<u>Cost</u>	0	2	7	-2	4



# Bellman Ford Pseudocode

```
Bellman-Ford( $G, w, s$ )  
Initialize-Single-Source( $G, s$ )  
for  $i = 1$  to  $|G.V| - 1$   
    for each  $edge(u, v) \in G.E$   
         $RELAX(u, v, w)$   
for each  $edge(u, v) \in G.E$   
    if  $v.d > u.d + w(u, v)$   
        return FALSE  
return TRUE
```

# Code - Bagian 1

## Inisiasi Jarak Terdekat Vertex sebagai INFINITE

```
void BellmanFord(struct Graph *graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other
    // vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
```

# Code - Bagian 2

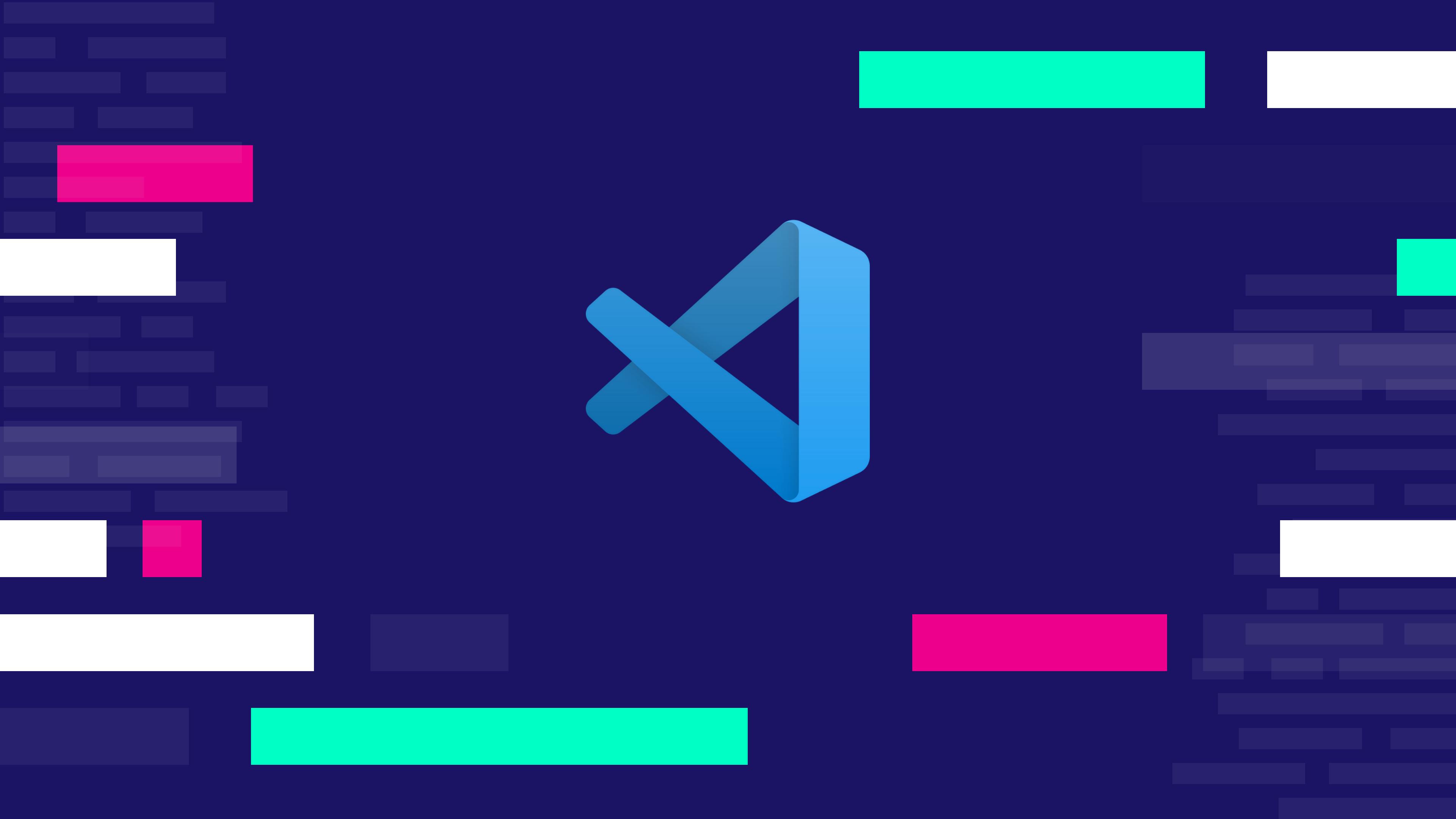
## Melakukan Relax pada setiap Edge

```
// Step 2: Relax all edges |V| - 1 times. A simple  
// shortest path from src to any other vertex can have  
// at-most |V| - 1 edges  
for (int i = 1; i <= V - 1; i++)  
{  
    for (int j = 0; j < E; j++)  
    {  
        int u = graph->edge[j].src;  
        int v = graph->edge[j].dest;  
        int weight = graph->edge[j].weight;  
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])  
            dist[v] = dist[u] + weight;  
    }  
}
```

# Code - Bagian 2

## Memeriksa cycles yang memiliki bobot negatif

```
// Step 3: check for negative-weight cycles. The above  
// step guarantees shortest distances if graph doesn't  
// contain negative weight cycle. If we get a shorter  
// path, then there is a cycle.  
for (int i = 0; i < E; i++)  
{  
    int u = graph->edge[i].src;  
    int v = graph->edge[i].dest;  
    int weight = graph->edge[i].weight;  
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])  
    {  
        printf("Graph contains negative weight cycle");  
        return; // If negative cycle is detected, simply  
               // return  
    }  
}
```



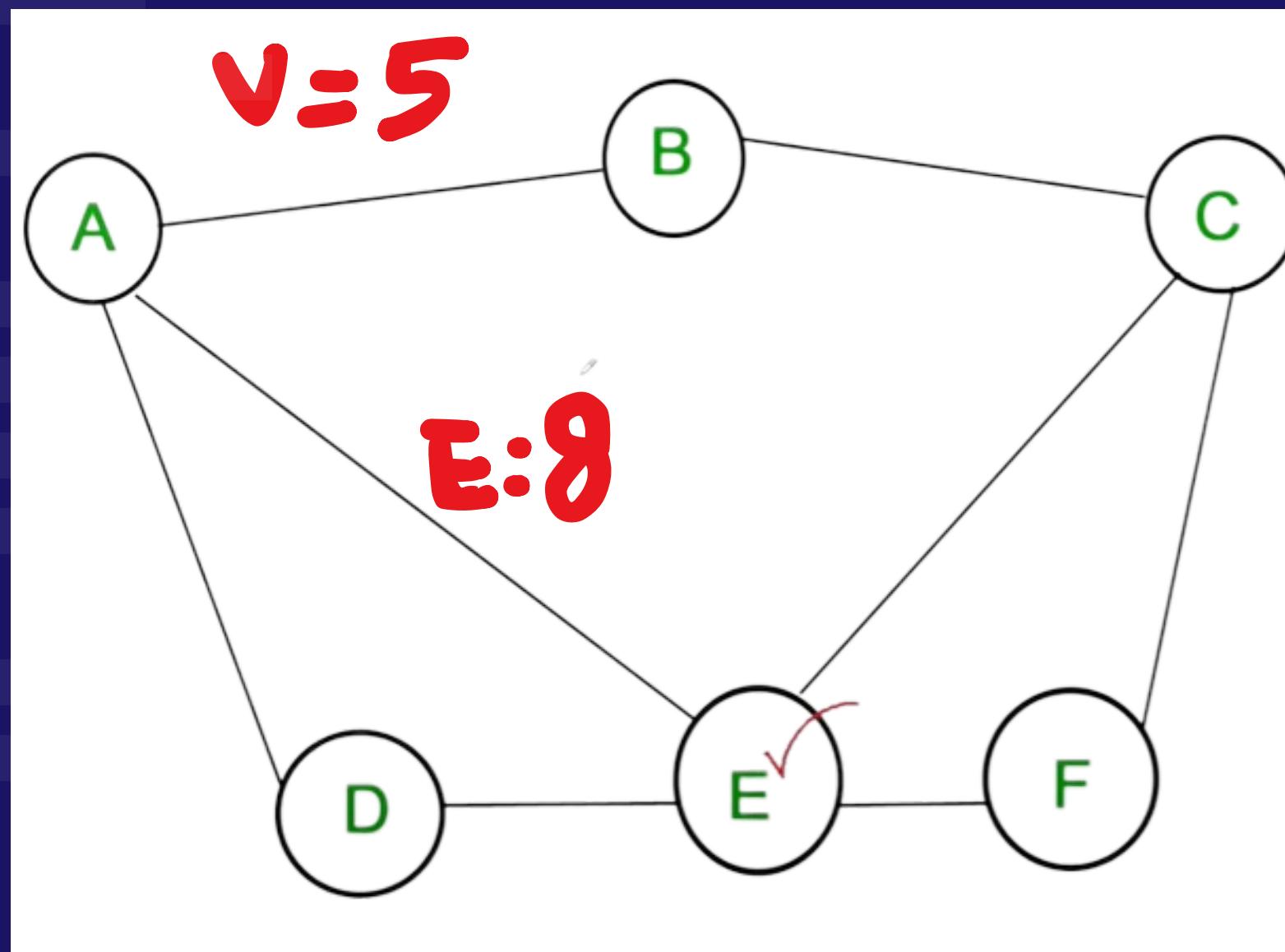
# FULL CODE

Dapat Diakses melalui :

<https://its.id/m/strukdat2023-t5-kel5>

# Time Complexity

# Kompleksitas Waktu (Simple Graph)



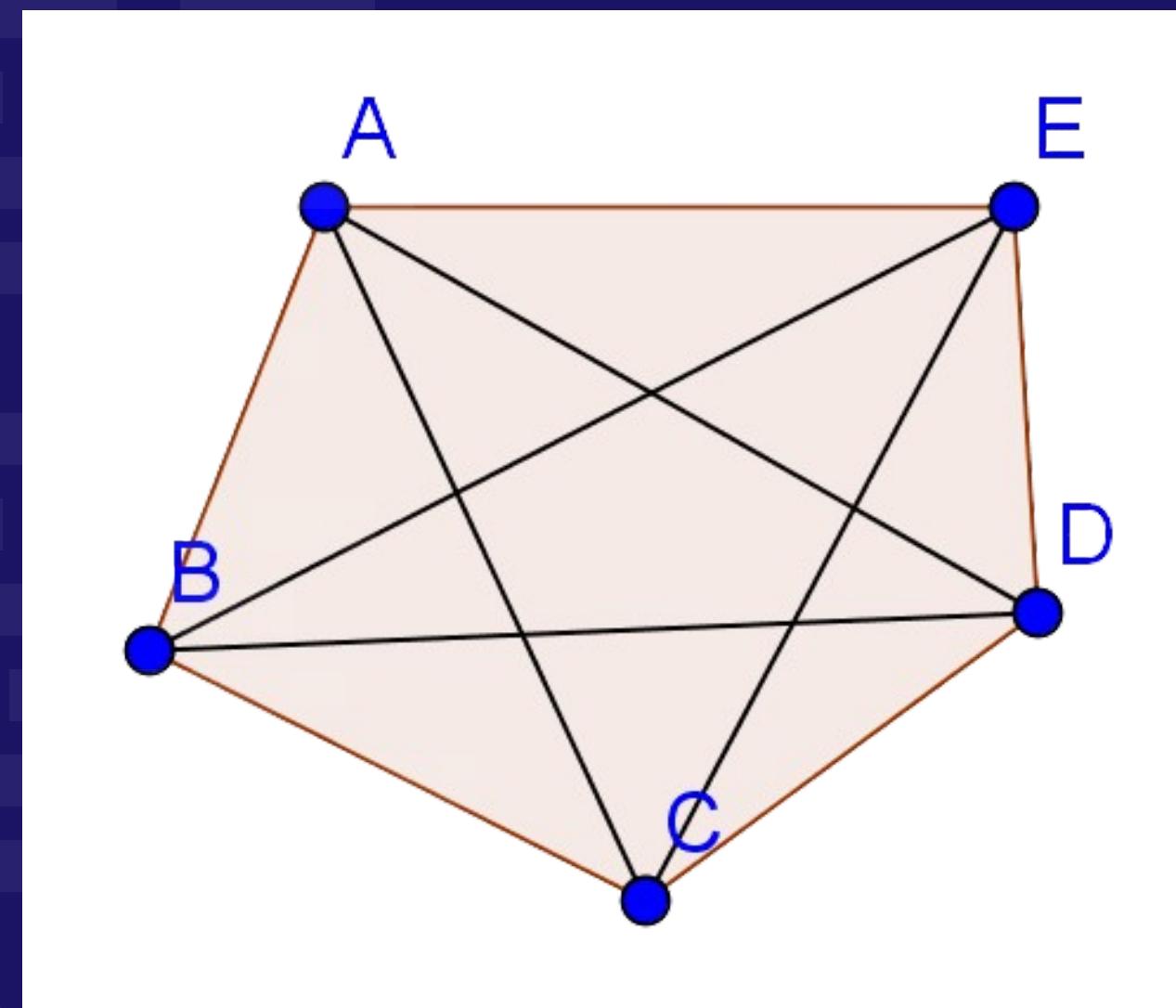
Kita perlu memperhitungkan seluruh relaksasi pada setia edge pada Graph. Sehingga :

$$\begin{aligned} O(|E| * |V| - 1) \\ O(|E| * |V|) \\ O(n * n) \\ O(n^2) \end{aligned}$$

[E] (Highlight Merah)  
Banyaknya edge pada graph

[V-1] (Highlight Biru)  
Banyaknya relaksasi pada satu Edge

# Kompleksitas Waktu (Complete Graph)



Pada complete graph terdapat edge untuk setiap pasan Vertex pada Graph, sehingga banyaknya Edge menjadi:

$$\begin{aligned} & O(|E| * |V| - 1) \\ & O(|E| * |V|) \\ & O(n(n-1)/2 * n) \\ & O(n^3) \end{aligned}$$

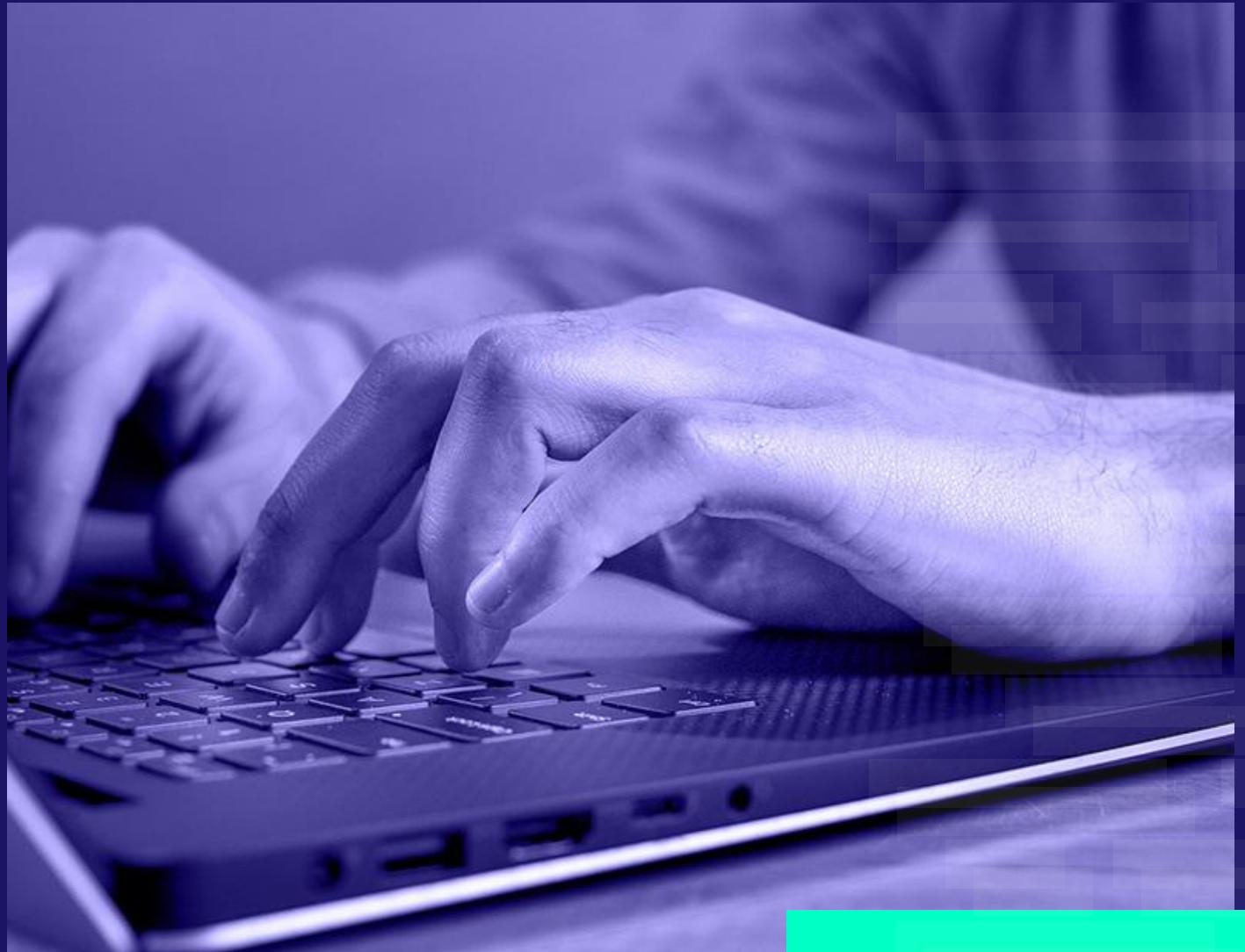
$$\begin{aligned} |E| &= n(n-1)/2 \\ |V| &= n \end{aligned}$$



# Aplikasi Algoritma Bellman Ford

# Aplikasi

- Navigasi GPS: Perangkat GPS menggunakan Bellman-Ford untuk menghitung rute terpendek atau tercepat antar lokasi, membantu aplikasi dan perangkat navigasi.
- Transportasi dan Logistik: Algoritma Bellman-Ford dapat diterapkan untuk menentukan jalur optimal kendaraan dalam transportasi dan logistik, meminimalkan konsumsi bahan bakar dan waktu perjalanan.
- Pengembangan Game: Bellman-Ford dapat digunakan untuk memodelkan pergerakan dan navigasi dalam dunia virtual dalam pengembangan game, di mana jalur yang berbeda mungkin memiliki biaya atau hambatan yang berbeda-beda.



**Thank you!**