

The background features abstract, translucent geometric shapes in shades of blue, purple, and green, floating against a dark blue gradient.

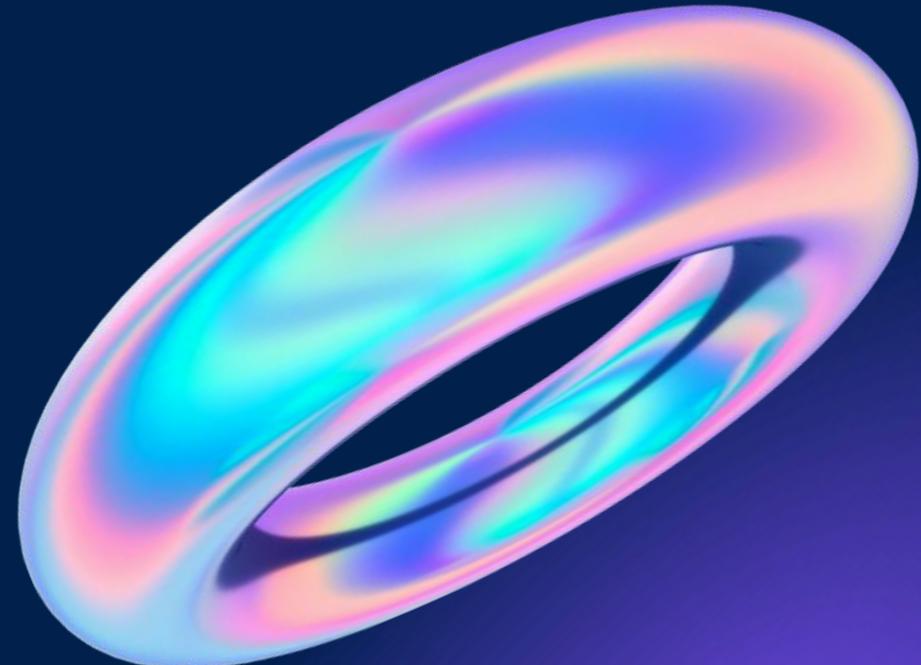
Laporan Praktikum Struktur Data Modul 3 dan 4

OLEH :
ANDI ALIYAH NUR INAYAH PATTOZA
5025221196



Praktikum

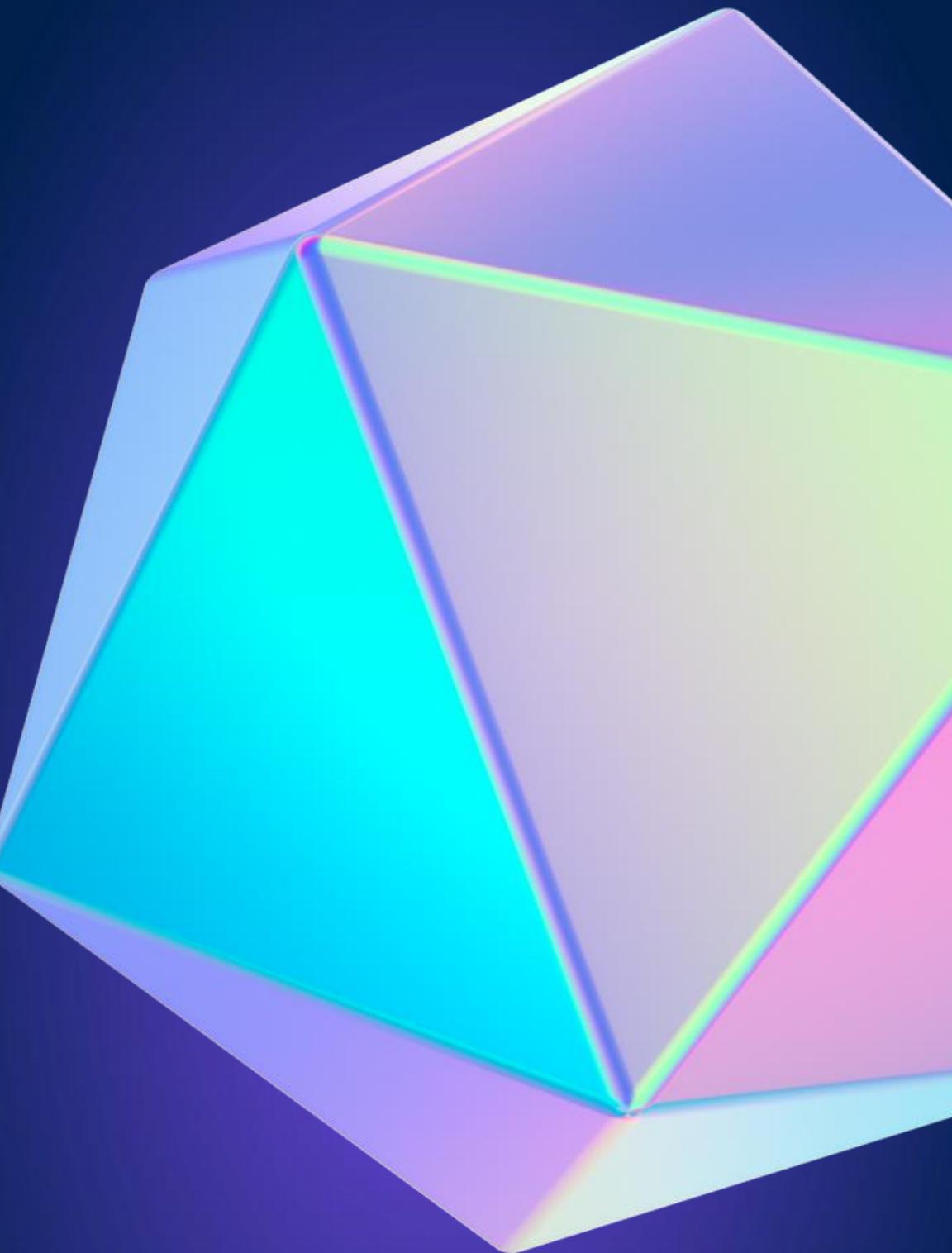
Modul 3



Miko Tukang Sihir (MTS)



Membantu Cika dalam mencari urutan ruangan dengan nomor unik tertentu dalam sebuah dimensi yang terbentuk melalui kekuatan sihir Miko. Dimensi tersebut berisi ruangan-ruangan yang saling terhubung membentuk sebuah Binary Search Tree (BST) dan selalu seimbang. Cika diminta untuk melakukan operasi "buat" untuk membuat ruangan baru dengan nomor unik tertentu, dan operasi "cari" untuk mencari urutan ruangan dengan nomor unik tertentu. Jika ruangan yang dicari tidak ditemukan, Cika harus memberikan respons yang sesuai.



Kode Program

- Deklarasikan struct dari AVL Tree

```
typedef struct AVLNode_t
{
    int val;
    struct AVLNode_t *left,*right;
    int height, index = 0;
}AVLNode;

typedef struct AVL_t
{
    AVLNode *_root;
    unsigned int _size;
}AVL;
```

- Fungsi avl_init digunakan untuk menginisialisasi sebuah AVL Tree.

```
void avl_init(AVL *avl)
{
    avl->_root = NULL;
    avl->_size = 0;
}
```

- Fungsi create digunakan untuk membuat sebuah AVLNode baru dengan nilai (val) tertentu.

```
AVLNode* create(int v) {
    AVLNode* newNode = new AVLNode();
    newNode->val = v;
    newNode->height = 1;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

- Fungsi height untuk mengembalikan tinggi (height) dari sebuah node dalam AVL Tree.

```
int height(AVLNode* node) {
    if(node == NULL)
        return 0;
    return node->height;
}
```

- Fungsi balanceFactor ini menghitung faktor keseimbangan dari sebuah node dalam AVL Tree.

```
int balanceFactor(AVLNode* node) {
    if(node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}
```

- Fungsi ini melakukan rotasi kanan pada AVL Tree dengan menggunakan node pivot yang diberikan. Rotasi kanan digunakan untuk memperbaiki ketidakseimbangan yang terjadi saat operasi penambahan (insertion) dilakukan pada subtree sebelah kiri yang menyebabkan LL case.

```
AVLNode* rightRotate(AVLNode* pivotNode) {

    AVLNode* newParrent=pivotNode->left;
    pivotNode->left=newParrent->right;
    newParrent->right=pivotNode;

    pivotNode->height=max(height(pivotNode->left), height(pivotNode->right))+1;
    newParrent->height=max(height(newParrent->left), height(newParrent->right))+1;

    return newParrent;
}
```

- Fungsi ini melakukan rotasi kiri pada AVL Tree dengan menggunakan node pivot yang diberikan. Rotasi kiri digunakan untuk memperbaiki ketidakseimbangan yang terjadi saat operasi penambahan (insertion) dilakukan pada subtree sebelah kanan yang menyebabkan RR case.

```
AVLNode* leftRotate(AVLNode* pivotNode) {

    AVLNode* newParrent = pivotNode->right;
    pivotNode->right = newParrent->left;
    newParrent->left = pivotNode;

    pivotNode->height = max(height(pivotNode->left), height(pivotNode->right))+1;
    newParrent->height = max(height(newParrent->left), height(newParrent->right))+1;

    return newParrent;
}
```

- Fungsi insert digunakan untuk menyisipkan sebuah node baru dengan nilai (val) tertentu ke dalam AVL Tree.

```
AVLNode* insert(AVL *avl, AVLNode* node, int val) {
    if(node == NULL) {
        return create(val);
    }

    if(val < node->val) {
        node->left = insert(avl, node->left, val);
    }
    else if(val > node->val) {
        node->right = insert(avl, node->right, val);
    }

    node->height = max(height(node->left), height(node->right)) + 1;
    int bf = balanceFactor(node);

    if(bf > 1 && val < node->left->val) { // LL
        return rightRotate(node);
    }
    if(bf < -1 && val > node->right->val) { // RR
        return leftRotate(node);
    }
    if(bf > 1 && val > node->left->val) { // LR
        node->left=leftRotate(node->left);
        return rightRotate(node);
    }
    else if(bf < -1 && val < node->right->val) { // RL
        node->right=rightRotate(node->right);
        return leftRotate(node);
    }
}

return node;
```

- Fungsi ini digunakan untuk memeriksa apakah sebuah nilai ada dalam AVL Tree.

```
bool avl_find(AVL *avl, int value) {
    AVLNode *temp = search(avl->_root, value);
    if (temp == NULL)
        return 0;
    if (temp->val == value)
        return 1;
    else
        return 0;
}
```

- Fungsi ini digunakan untuk menyisipkan nilai ke dalam AVL Tree.

```
void avl_insert(AVL *avl, int value)
{
    if (!avl_find(avl, value))
    {
        avl->_root = insert(avl, avl->_root, value);
        avl->_size++;
    }
}
```

- Fungsi ini digunakan untuk mencari node dengan nilai minimum

```
AVLNode *_findMinNode(AVLNode *node)
{
    AVLNode *currNode = node;
    while (currNode && currNode->left != NULL)
        currNode = currNode->left;
    return currNode;
}
```

- Fungsi ini digunakan untuk mencari node dengan nilai tertentu dalam AVL Tree. Fungsi ini melakukan pencarian secara iteratif mulai dari root dan berlanjut ke kiri atau kanan berdasarkan nilai yang dicari.

```
AVLNode* search(AVLNode *root, int value) {
    while (root != NULL) {
        if (value < root->val)
            root = root->left;
        else if (value > root->val)
            root = root->right;
        else
            return root;
    }
    return root;
}
```

- Fungsi ini melakukan pengindeksan secara preorder pada AVL Tree. Setiap node diberi nomor indeks (index) berdasarkan urutan pengindeksan. Dimulai dari root -> node kanan -> node kiri.

```
int preorder(AVLNode *root, int index)
{
    if (root)
    {
        root->index = index;
        index++;
        index = preorder(root->right, index);
        index = preorder(root->left, index);
    }
    return index;
}
```

```
int main() {
    AVL avl;
    avl_init(&avl);
    int T;
    cin >> T;

    while (T--) {
        string Q;
        int n;
        cin >> Q;
        cin >> n;

        if (Q == "buat") {
            avl_insert (&avl, n);
        } else if (Q == "cari") {
            preorder(avl._root, 1);
            AVLNode *node = search(avl._root, n);

            if (node)
                cout << "Ruangannya ada di urutan ke-" << node->index << endl;
            else
                cout << "Lah, ruangannya mana?\n";
        } else {
            cout << "Maksudnya gimana?\n";
        }
    }
}
```

Input Format

Pada baris pertama, terdapat bilangan T
T baris berikutnya:

- Apabila input adalah **buat n**, maka buatlah ruang baru ke dalam dimensi dengan nomor unik **n**.
- Apabila input adalah **cari Q**, maka cari urutan ruang dengan nomor unik **Q**.

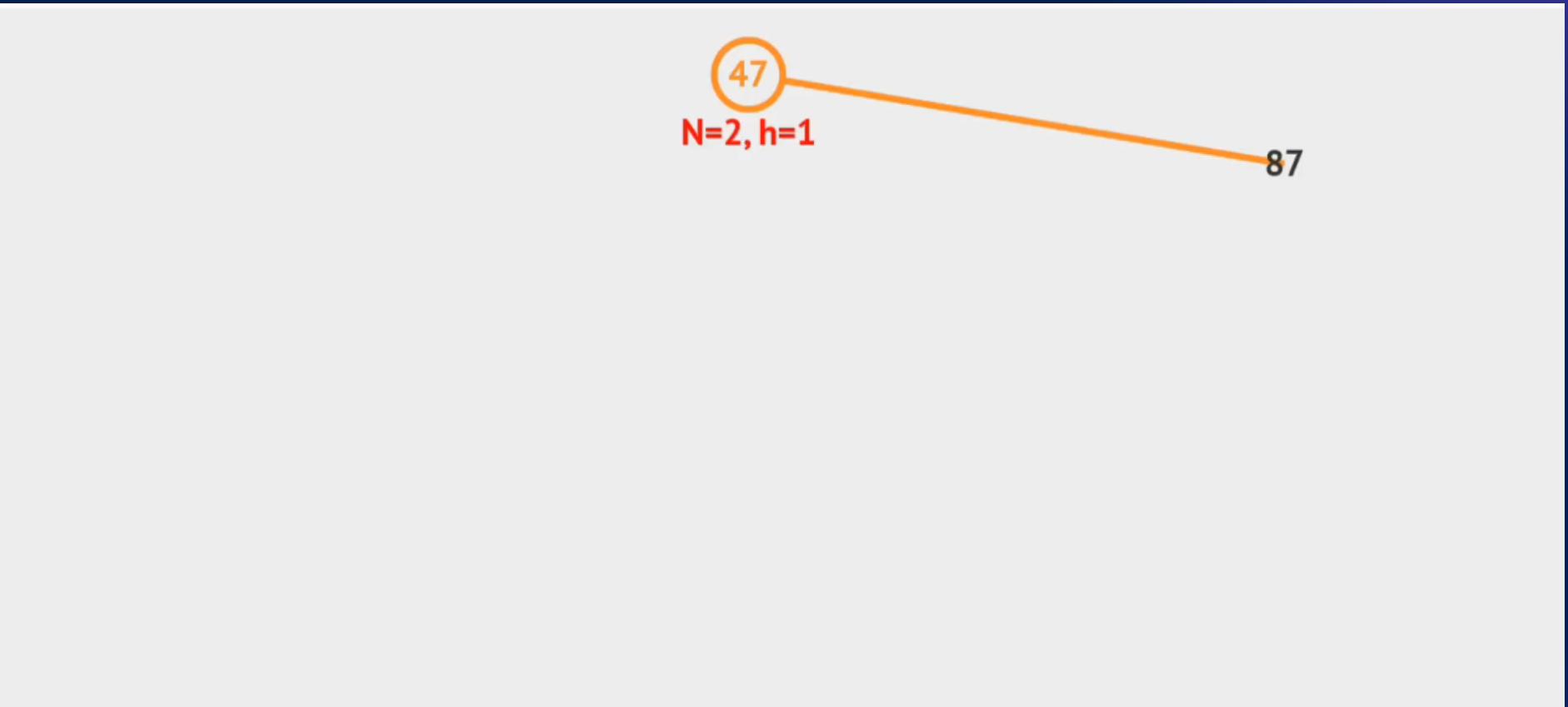
Output Format

Output dapat berupa:

- Apabila input adalah **cari Q**, maka cetak urutan ke-**n** dari ruang dengan nomor unik **Q** dengan format "Ruangannya ada di urutan ke-**n**" tanpa tanda kutip
- Apabila ruangan yang dicari tidak ditemukan, maka cetak "Lah, ruangannya mana?"
- Apabila ditemukan input selain buat dan cari maka cetak "Maksudnya gimana?" tanpa tanda kutip

Sample input dan Output

```
11
buat 47
buat 89
cari 89
Ruangannya ada di urutan ke-2
buat 64
buat 35
buat 14
buat 8
cari 8
Ruangannya ada di urutan ke-6
cari 47
Ruangannya ada di urutan ke-4
cari 17
Lah, ruangannya mana?
Cek 25
Maksudnya gimana?
```



Video di atas adalah ilustrasi avl dari sample inputnya. Jadi berdasarkan kode pada int main, saat diinputkan “cari” maka dia akan melakukan pemanggilan rekursi ke function preorder. Dimana program itu akan melakukan preorder yang dimulai dari root kemudian ke node kanan lalu ke node kiri. Lalu setiap diinputkan angka, maka program itu akan melakukan rotation apabila tree nya tidak balance.

Menyusun Piramida (MPD)

membuat program yang dapat mengecek apakah penambahan sebuah angka ke dalam sebuah AVL tree akan menjaga keseimbangan (balance) dari tree tersebut atau tidak. Dalam permainan Menyusun Piramida, setiap pemain akan memasukkan angka secara bergantian ke dalam piramida berbentuk AVL tree, dan pemain akan mendapatkan poin jika angka yang mereka masukkan dapat menjaga sifat balance dari tree yang sudah terbentuk. Program yang dibuat akan mengambil input berupa jumlah angka pada tree, nilai setiap node pada tree yang sudah terbentuk, dan angka yang akan ditambahkan. Output dari program akan memberikan informasi apakah penambahan angka tersebut akan menjaga keseimbangan tree atau tidak.



Kode Program

- Deklarasikan struct untuk merepresentasikan node dalam AVL tree

```
typedef struct Node {  
    int data;  
    Node *left, *right;  
}AVL;
```

- Fungsi insert digunakan untuk memasukkan sebuah nilai ke dalam AVL tree.

```
AVL* insert(AVL* root, int val) {  
    if (root == NULL) {  
        return create(val);  
    }  
    if (val < root->data) {  
        root->left = insert(root->left, val);  
    } else if (val > root->data){  
        root->right = insert(root->right, val);  
    }  
    return root;  
}
```

- Fungsi create digunakan untuk membuat sebuah node baru dengan nilai yang diberikan sebagai argumen.

```
AVL* create(int value) {  
    AVL* node = (AVL*)malloc(sizeof(AVL));  
    node->data = value;  
    node->left = NULL;  
    node->right = NULL;  
    return node;  
}
```

- Fungsi height digunakan untuk menghitung tinggi dari sebuah node dalam AVL tree.

```
int height(AVL* root) {  
    if (root == NULL) {  
        return -1;  
    }  
    int leftHeight = height(root->left);  
    int rightHeight = height(root->right);  
    if (leftHeight > rightHeight) {  
        return leftHeight + 1;  
    } else {  
        return rightHeight + 1;  
    }  
}
```

- Fungsi ini digunakan untuk memeriksa apakah sebuah AVL tree adalah balance atau tidak.

```
int isBalanced(AVL* root) {
    if (root == NULL) {
        return 1;
    }
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    if (abs(leftHeight - rightHeight) <= 1 && isBalanced(root->left) && isBalanced(root->right)) {
        return 1;
    }
    return 0;
}
```

```
int main() {
    int n, x;
    cin >> n;
    AVL* root = NULL;
    int value;
    for (int i = 0; i < n; i++) {
        cin >> value;
        root = insert(root, value);
    }
    cin >> x;
    root = insert(root, x);
    if (isBalanced(root)) {
        cout << "Tree tetap balance\n";
    } else {
        cout << "Tree tidak balance\n";
    }
    return 0;
}
```

Input Format

Baris pertama berisi bilangan bulat n, yaitu banyak angka pada tree

Baris kedua berisi n bilangan bulat yang merupakan nilai setiap node pada tree
(urutan sudah dipastikan membentuk tree yang balance)

Baris ketiga berisi bilangan bulat x, yaitu angka yang akan diinsert

Ouput Format

“Tree tetap balance”, jika angka yang diberikan (x) dapat menjaga balance dari tree.

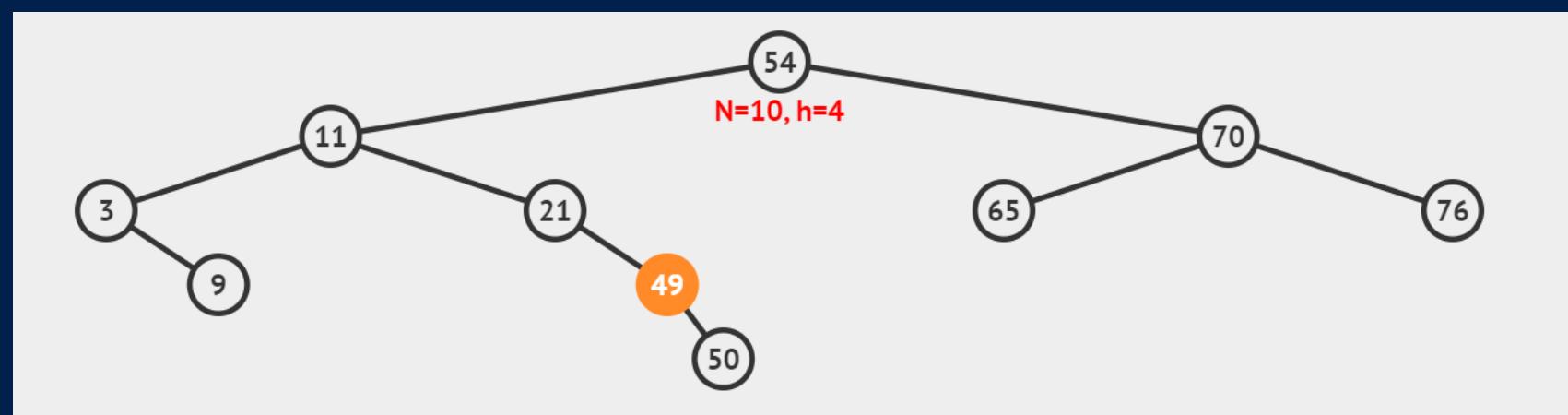
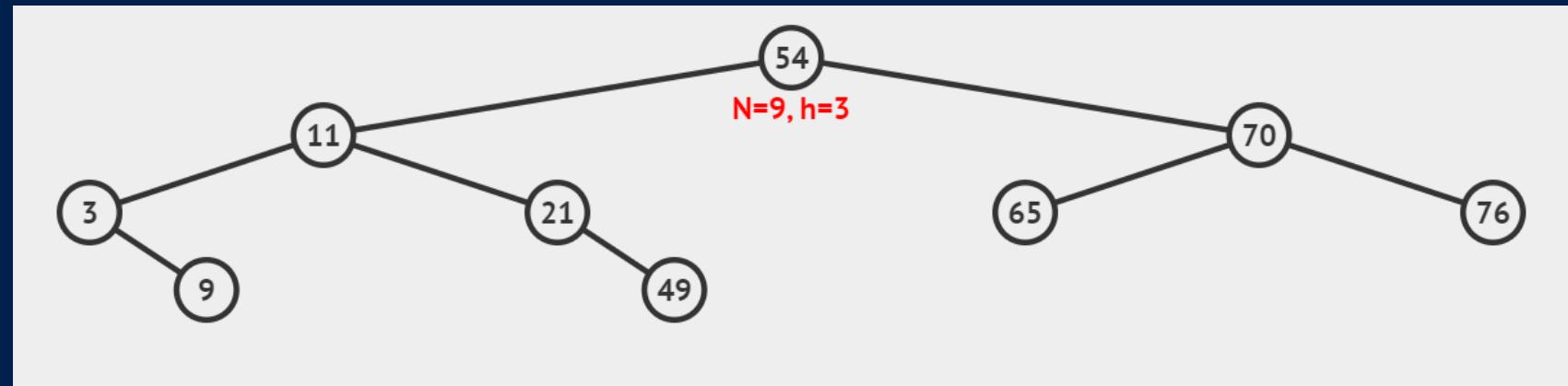
“Tree tidak balance”, jika angka yang diberikan (x) dapat membuat tree tidak balance.

Sample input dan Output

```
9  
54 11 70 3 21 65 76 9 49
```

```
50
```

```
Tree tidak balance
```

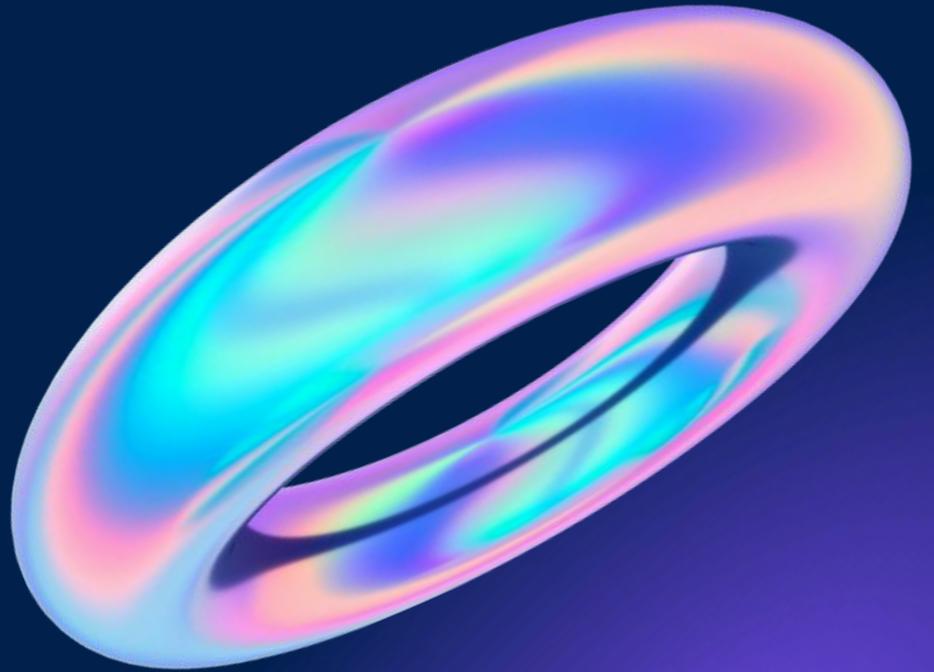


Tree dianggap tidak balance jika pada salah satu subtree (kiri atau kanan) terdapat ketidakseimbangan atau selisih tinggi lebih dari 1. Pada sample input tersebut, dapat dilihat pada gambar tengah (gambar 2) tree nya balance karena selisih tinggi node nya tidak ada yang lebih dari 1. Tapi saat di inputkan 50, tree nya akan tidak balance, hal ini karena nilai 50 akan terletak sebagai anak dari 49, dimana hal ini menyebabkan node 21 memiliki subtree kanan 2 dan subtree kiri nya 0 yang mengakibatkan selisih nya lebih dari 1 sehingga tree nya tidak balance



Praktikum

Modul 4



Pemerintah Buat Jembatan (PBJ)

Pada sebuah desa kepulauan ditunjuk seorang pemerintah baru. Program pertama yang ia akan jalankan adalah menghubungkan semua daerah kepulauan dengan jembatan. Masalahnya, sekarang sudah ada beberapa jembatan dibuat menghubungkan beberapa desa. Pemerintah itu pun kebingungan untuk membuat rancangan anggaran, karena ia belum tau ada berapa jembatan yang harus dibuat untuk menghubungkan semua pulau.

Akhirnya memiliki ide untuk menyewa seorang programmer untuk membuat program untuk membantu pemerintah! (Hint: Programmer tersebut dapat melihat rangkaian pulau tersebut seperti sebuah graf, dan diminta untuk mencari banyak edge yang diperlukan untuk menjadikan semua menjadi satu connected graph).



Kode Program

```
// Struktur data untuk merepresentasikan suatu edge dalam graf
typedef struct Edge {
    int sc; // source vertex
    int dest; // destination vertex
} Edge;

// Struktur data untuk merepresentasikan graf
typedef struct Graf {
    int numVertex; // jumlah vertex dalam graf
    int numEdges; // jumlah edge dalam graf
    Edge edges[MAX]; // array yang menyimpan edge-edge dalam graf
} Graph;

// Fungsi untuk mencari root dari suatu subset (digunakan dalam operasi union-find)
int findRoot(int parent[], int vertex) {
    while (parent[vertex] != vertex)
        vertex = parent[vertex];
    return vertex;
}

// Fungsi untuk menggabungkan dua subset (digunakan dalam operasi union-find)
void unionSet(int parent[], int size[], int vertex1, int vertex2) {
    int root1 = findRoot(parent, vertex1);
    int root2 = findRoot(parent, vertex2);

    if (root1 != root2) {
        // Menggabungkan subset dengan menggunakan rank/size
        if (size[root1] < size[root2]) {
            parent[root1] = root2;
            size[root2] += size[root1];
        } else {
            parent[root2] = root1;
            size[root1] += size[root2];
        }
    }
}
```

```
// Fungsi untuk menghitung jumlah subset dalam graf
int countEdges(Graph* graph) {
    int parent[MAX], size[MAX];

    // Menginisialisasi setiap vertex sebagai subset terpisah
    for (int i = 0; i < graph->numVertex; i++) {
        parent[i] = i;
        size[i] = 1;
    }

    // Menggabungkan setiap edge dalam graf menggunakan operasi union-find
    for (int i = 0; i < graph->numEdges; i++) {
        int first = graph->edges[i].sc;
        int second = graph->edges[i].dest;
        unionSet(parent, size, first, second);
    }

    int count = 0;
    // Menghitung jumlah subset dalam graf
    for (int i = 0; i < graph->numVertex; i++) {
        if (parent[i] == i)
            count++;
    }
    // Mengembalikan jumlah subset dikurangi 1, karena subset terakhir tidak memiliki edge
    return count - 1;
}
```

```
int main() {
    int V, N;
    cin >> V >> N;

    Graph graf;
    graf.numVertex = V;
    graf.numEdges = N;

    // Membaca input edge-edge dalam graf
    for (int i = 0; i < N; i++) {
        int A, B;
        cin >> A >> B;
        graf.edges[i].sc = A;
        graf.edges[i].dest = B;
    }

    // Menghitung dan mencetak jumlah minimum edge yang diperlukan untuk menghubungkan semua vertex
    int minEdge = countEdges(&graf);
    cout << minEdge << endl;

    return 0;
}
```

Input Format

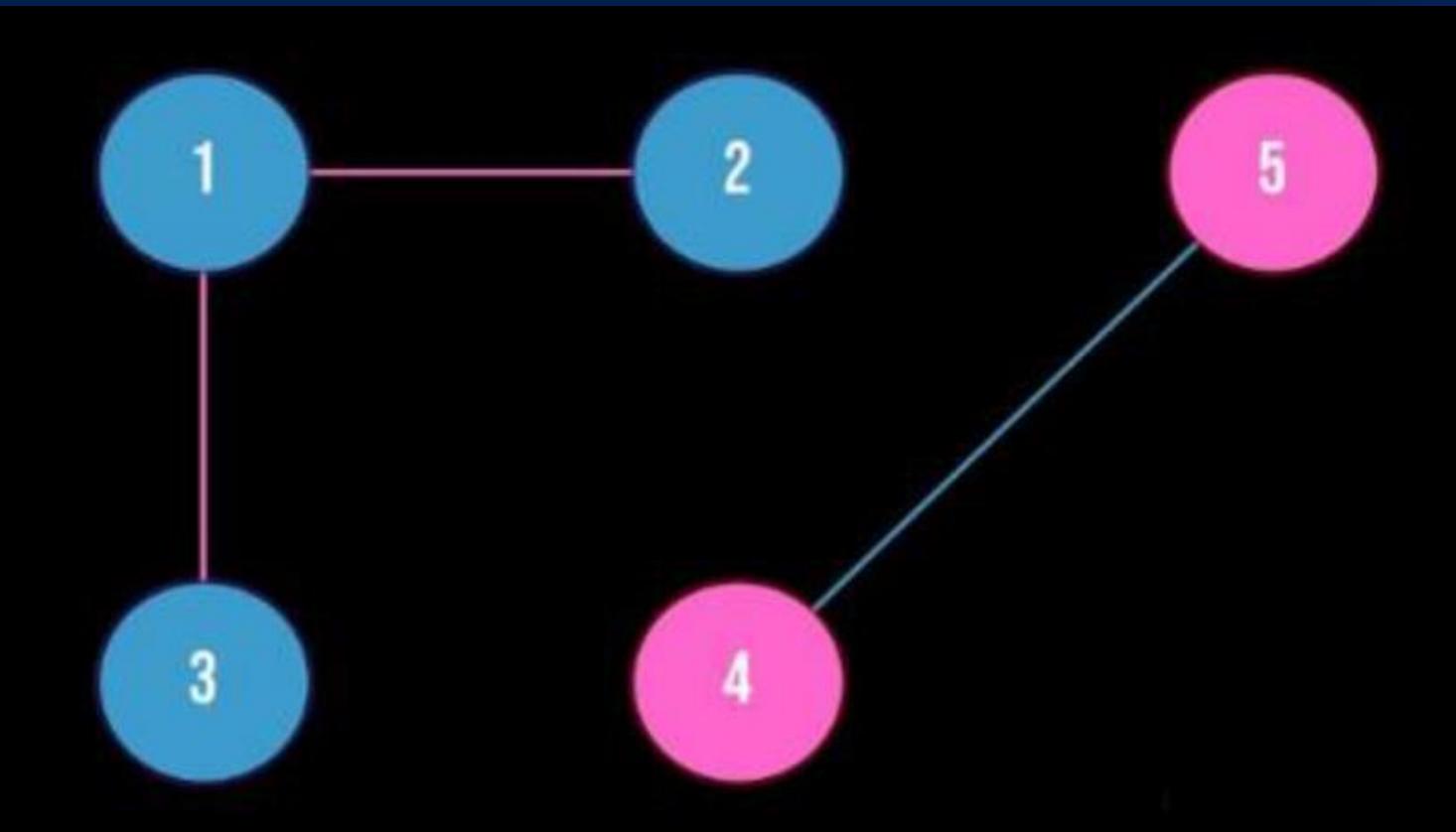
Baris pertama adalah V dan N yang merupakan jumlah vertex dan jumlah pasangan vertex. Sebanyak N baris adalah A dan B yang merupakan pasangan vertex yang saling terhubung.

Output Format

Keluarkan berapa banyak edge yang dibutuhkan agar semua vertex dapat terhubung menjadi satu connected graf.

Sample input dan Output

```
5 4
1 2
2 3
3 1
4 5
1
```

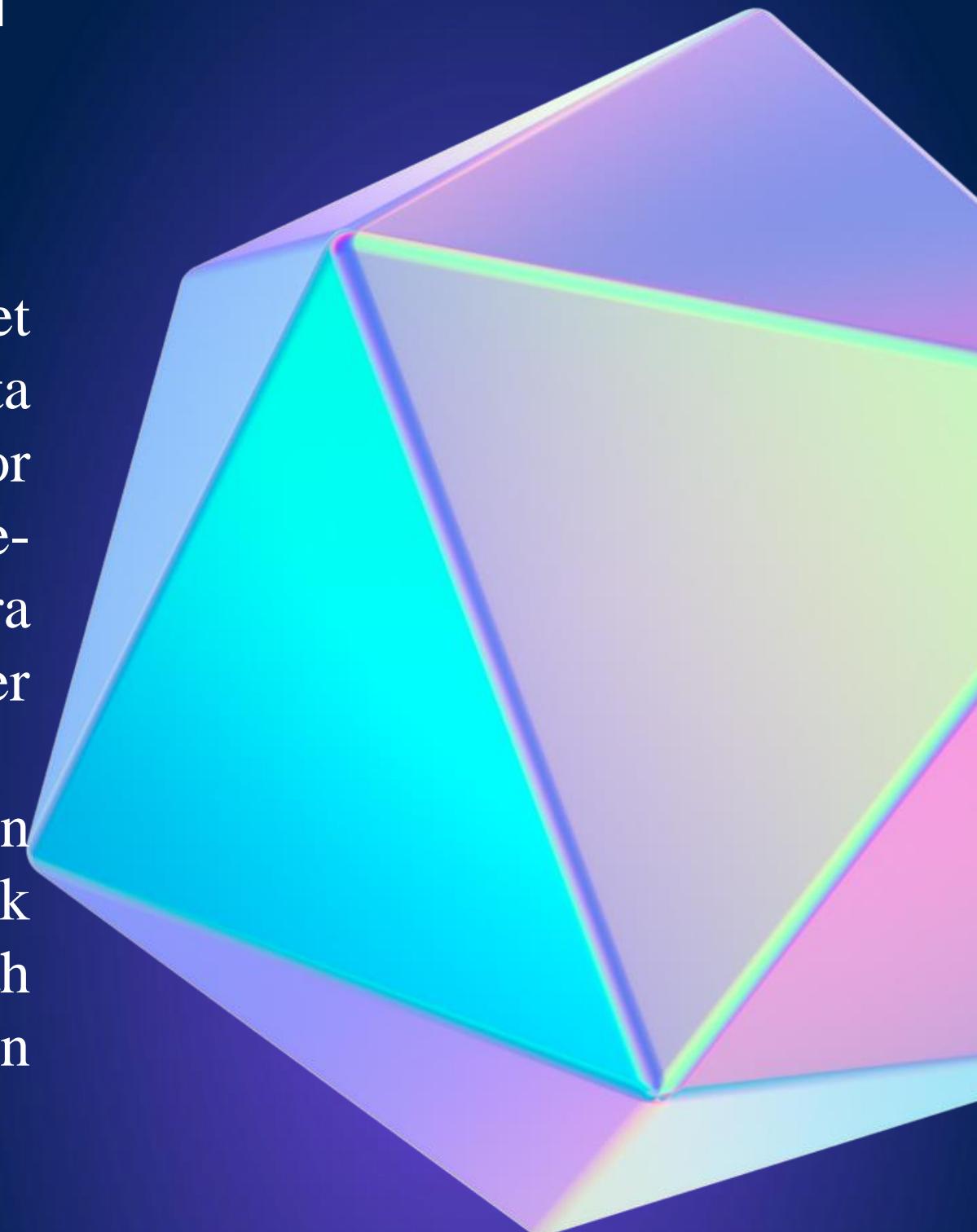


Karena diantara 1, 2 dan 3 tidak ada edge yang menghubungkannya dengan 4 ataupun 5, maka terbentuklah 2 graf dan untuk menghubungkannya membentuk satu connected graph dibutuhkan 1 buah edge yang menjadi penghubung antara 1, 2, 3 dengan 4, 5.

A Hunt For Shocker (AHOCK)

Mencari anggota Shocker yang paling cocok dijadikan target berdasarkan jumlah neighbor yang paling sedikit, dan anggota Shocker yang paling berbahaya berdasarkan jumlah neighbor yang paling banyak. Menggambarkan sebuah graf dengan node-node yang mewakili anggota Shocker, di mana setiap edge antara node-node tersebut mewakili persekutuan antara anggota Shocker yang terhubung.

Dalam graf ini, semakin sedikit neighbor sebuah node, semakin rendah pangkat anggota Shocker tersebut dan semakin cocok dijadikan target. Sebaliknya, semakin banyak neighbor sebuah node, semakin tinggi pangkat anggota Shocker tersebut dan semakin berbahaya.

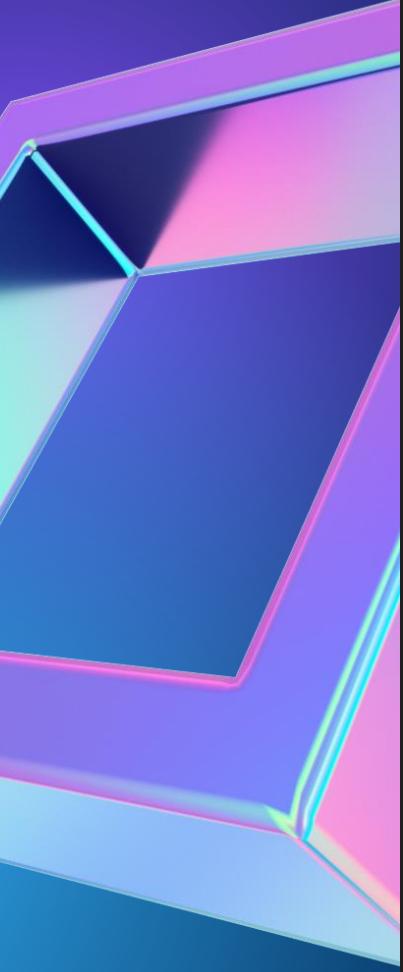


Kode Program

```
int main() {
    // Inputkan jumlah Anggota
    int N;
    cin >> N;
    // Buat batasan N nya dari 3 sampai 1000
    if (N < 3 || N > 1000) {
        return 0;
    }

    // Membuat vektor untuk menyimpan graf, dan peta (map) untuk menghitung jumlah sekutu setiap anggota
    vector<vector<int>> graf(N + 1); // Graf dengan N+1 node
    map<int, int> sekutu; // Peta untuk menghitung jumlah sekutu setiap anggota

    // Baris-baris persekutuan antar anggota
    int a, b;
    string input;
    while (true) {
        cin >> input;
        if (input == "Yee!") // Programnya akan berhenti jika diinputkan "Yee!"
            break;
        a = stoi(input); // Mengubah input menjadi tipe data integer
        cin >> b;
        graf[a].push_back(b); // Menambahkan node b ke tetangga node a
        graf[b].push_back(a); // Menambahkan node a ke tetangga node b
        sekutu[a]++; // Meningkatkan jumlah sekutu anggota a
        sekutu[b]++; // Meningkatkan jumlah sekutu anggota b
```



```
int target = -1;
int bahaya = -1;
int minNeighbor = N + 1; // Inisialisasi nilai tetangga minimum dengan nilai maksimum yang mungkin
int maxNeighbor = -1; // Inisialisasi nilai tetangga maksimum dengan nilai minimum yang mungkin

// Melakukan iterasi untuk setiap anggota Shocker dari 1 hingga N
for (int node = 1; node <= N; ++node) {
    int sekutu_n = sekutu[node]; // Jumlah sekutu dari anggota node saat ini

    // Memperbarui anggota target jika jumlah tetangga lebih sedikit
    if (sekutu_n < minNeighbor) {
        minNeighbor = sekutu_n;
        target = node;
    }
    // Memperbarui anggota target jika jumlah tetangga sama, tapi nilai node lebih besar
    else if (sekutu_n == minNeighbor && node > target) {
        target = node;
    }

    // Memperbarui anggota berbahaya jika jumlah tetangga lebih banyak
    if (sekutu_n > maxNeighbor) {
        maxNeighbor = sekutu_n;
        bahaya = node;
    }
    // Memperbarui anggota berbahaya jika jumlah tetangga sama, tapi nilai node lebih kecil
    else if (sekutu_n == maxNeighbor && node < bahaya) {
        bahaya = node;
    }
}

// Outputkan hasil
cout << target << " Targetnya" << endl;
cout << bahaya << " Paling Bahaya" << endl;

return 0;
```

Input Format

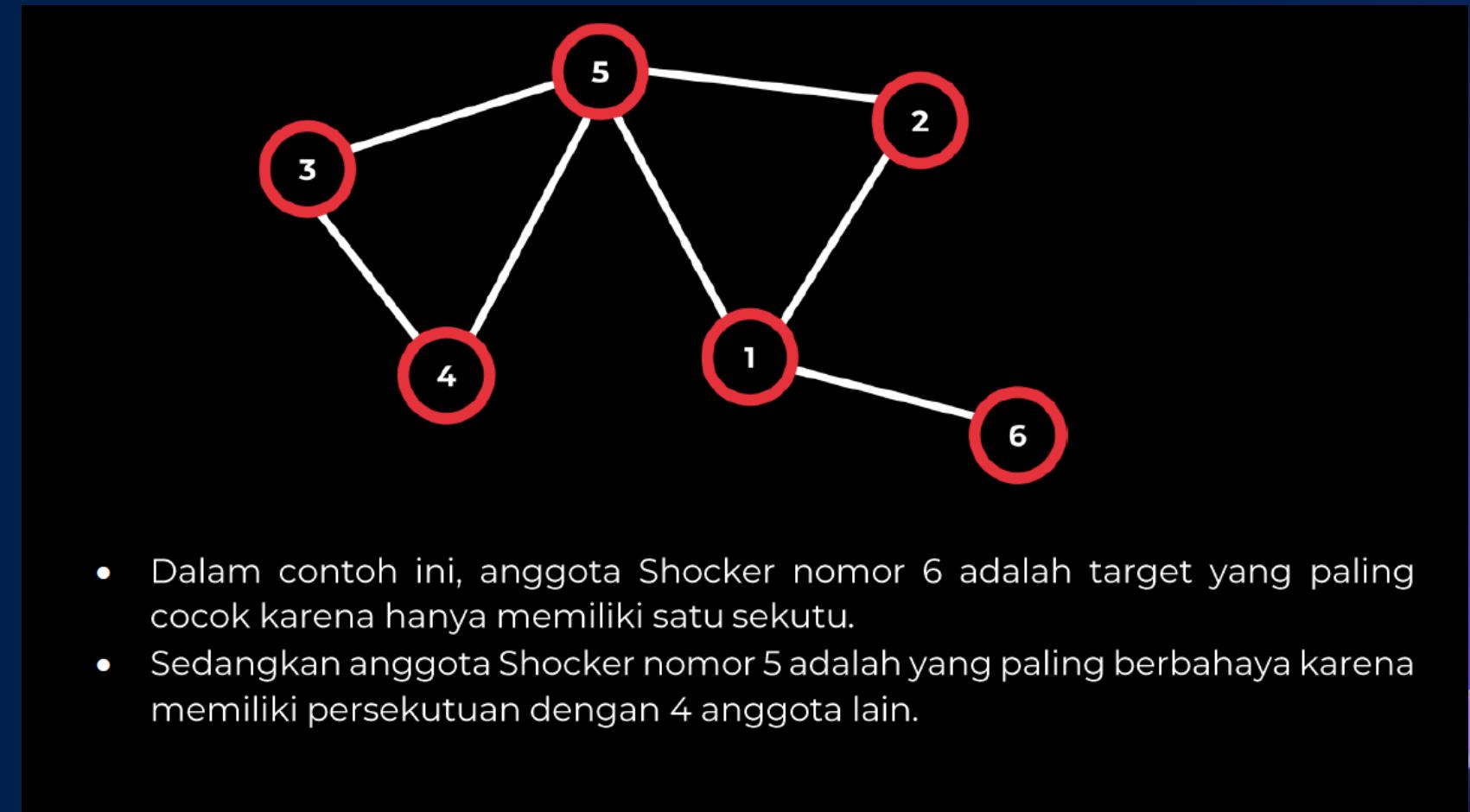
- Baris pertama adalah integer N, jumlah anggota
- Baris-Baris berikutnya adalah persekutuan-persekutuan yang terjadi antar anggota
- Input akan berakhir dengan kata “Yee!”

Output Format

- Baris pertama, output-kan anggota yang paling cocok dijadikan target
- Baris kedua, output-kan anggota yang berbahaya

Sample input dan Output

```
6
1 2
1 5
1 6
2 5
3 4
3 5
4 5
Yee!
6 Targetnya
5 Paling Bahaya
```



Connected Number (CN)



Bilangan bulat a dan b terhubung jika $(a \text{ xor } b)$ merupakan kelipatan dari bilangan bulat c . Setiap bilangan yang terhubung akan berada dikelompok yang sama. Diberikan suatu list bilangan bulat dan sebuah bilangan c , tentukan banyak kelompok yang akan terbentuk dari kumpulan bilangan tersebut.

INPUT FORMAT

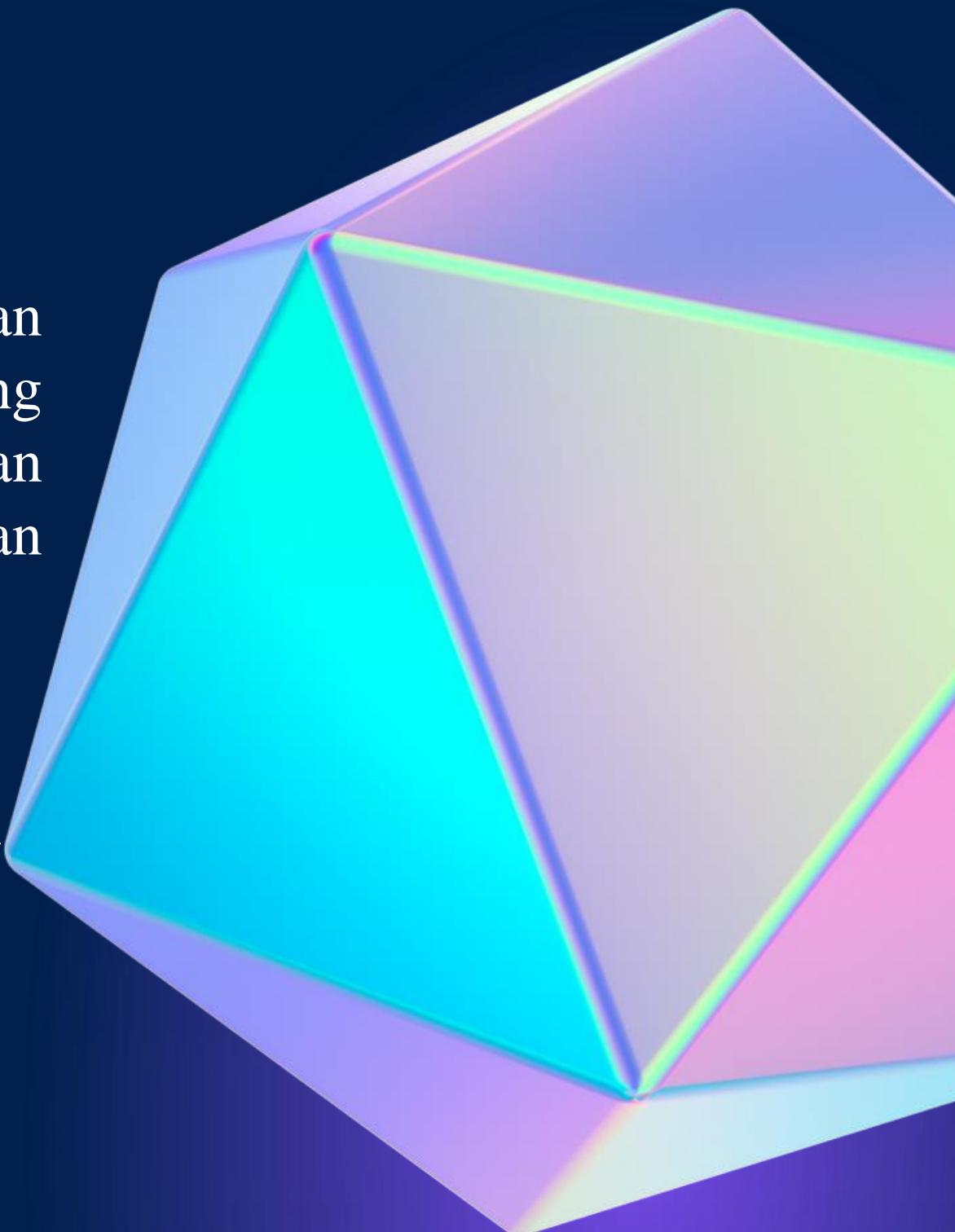
Baris pertama berisi bilangan bulat n , yaitu banyak bilangan pada list.

Baris kedua berisi n bilangan bulat x_i , yaitu bilangan yang ada pada list.

Baris ketiga berisi bilangan bulat c

OUTPUT FORMAT

1 bilangan bulat yang merupakan jumlah kelompok yang akan terbentuk.



Kode Program

```
int main() {
    int n;
    // Menginputkan banyak bilangan pada list (n)
    cin >> n;
    // Memeriksa batasan n
    if (n < 1 || n > 100) {
        return 0;
    }
    /* membaca n bilangan bulat yang ada dalam list
    dan memasukkannya ke dalam vektor x */
    vector<int> x(n);
    for (int i = 0; i < n; i++) {
        cin >> x[i];
        // Memeriksa batasan nilai x[i]
        if (x[i] < 1 || x[i] > 10000) {
            return 0;
        }
    }
    // Menginputkan nilai bilangan bulat
    int c;
    cin >> c;

    // Memeriksa batasan nilai c
    if (c < 1 || c > 10000) {
        return 0;
    }
}
```

```
// Membuat unordered map 'parent' untuk menyimpan parent dari setiap bilangan
// Membuat unordered map 'group' untuk menyimpan kelompok bilangan yang terhubung
unordered_map<int, int> parent;
unordered_map<int, vector<int>> group;

// Inisialisasi setiap bilangan menjadi parent dari dirinya sendiri
// Setiap bilangan juga menjadi anggota dari kelompoknya sendiri
for (int num : x) {
    parent[num] = num;
    group[num].push_back(num);
}

// Melakukan iterasi untuk setiap pasangan bilangan dalam list
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int xorRes = x[i] ^ x[j];

        // Jika hasil XOR merupakan kelipatan dari c
        if (xorRes % c == 0) {
            int parentX = parent[x[i]];
            int parentY = parent[x[j]];

            // Jika kedua bilangan memiliki parent yang berbeda
            if (parentX != parentY) {
                // Menggabungkan kelompok dengan memperbarui parent dan anggota kelompok
                for (int num : group[parentY]) {
                    parent[num] = parentX;
                    group[parentX].push_back(num);
                }
                group[parentY].clear();
            }
        }
    }
}

int Kelompok = 0;

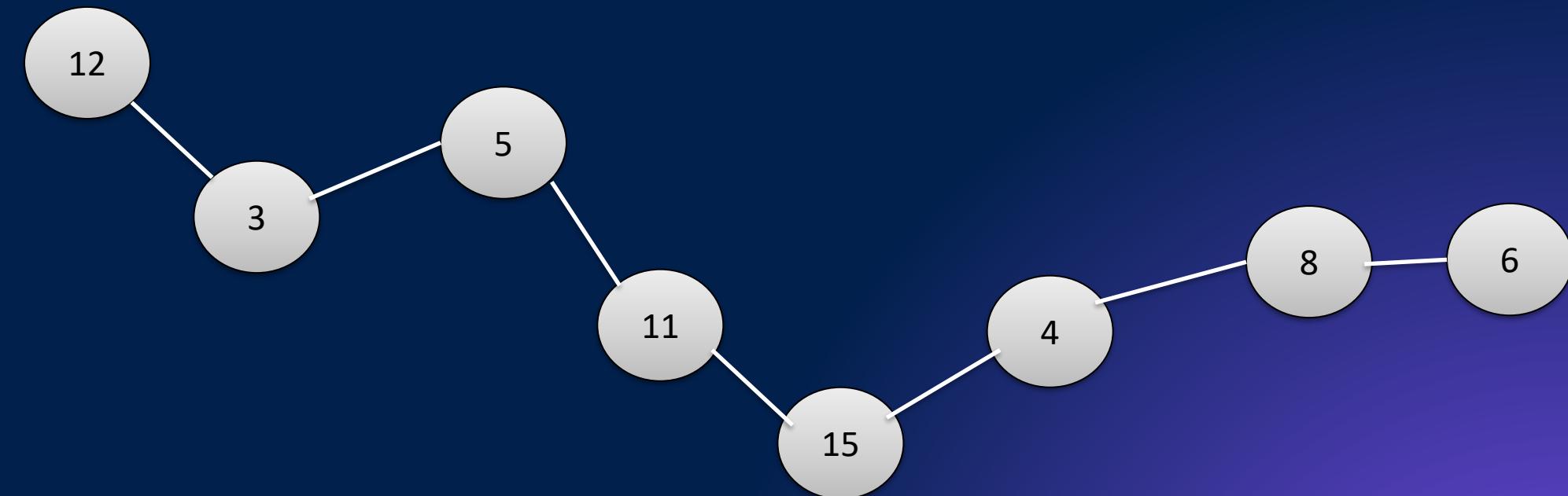
// Menghitung jumlah kelompok yang terbentuk
for (auto it = group.begin(); it != group.end(); ++it) {
    if (!it->second.empty())
        Kelompok++;
}

// Cetak jumlah kelompok yang terbentuk
cout << Kelompok << endl;

return 0;
```

Sample Input dan Output

```
8  
12 3 5 11 15 4 8 6  
7  
3
```



Bilangan yang terhubung:

- $12 \text{ xor } 11 = 7$, 7 merupakan kelipatan 7 sehingga '42' dan '35' berada dikelompok yang sama
- $3 \text{ xor } 4 = 7$, 7 merupakan kelipatan 7 sehingga '3' dan '4' berada dikelompok yang sama
- $5 \text{ xor } 11 = 14$, 14 merupakan kelipatan 7 sehingga '5' dan '11' berada dikelompok yang sama
- $15 \text{ xor } 8 = 7$, 7 merupakan kelipatan 7 sehingga '15' dan '8' berada dikelompok yang sama
- $8 \text{ xor } 6 = 14$, 14 merupakan kelipatan 7 sehingga '8' dan '6' berada di kelompok yang sama

Dari daftar bilangan yang terhubung tersebut, maka terdapat 3 kelompok bilangan yang terbentuk, yaitu:

1. Kelompok dengan anggota bilangan 12, 11, 5
2. Kelompok dengan anggota bilangan 3, 4
3. Kelompok dengan anggota bilangan 6, 8, 15

Penjelasan Lanjutan

Untuk menemukan kelompok-kelompok bilangan yang terhubung, hal yang harus dilakukan adalah mencari XOR (eXclusive OR) dari tiap dua nilai. Kemudian, nilai XOR nya akan diperiksa apakah hasilnya merupakan kelipatan dari ‘c’. Jika iya maka kedua nilai itu akan digabungkan kedalam satu kelompok. Pada awalnya, setiap bilangan memiliki dirinya sendiri sebagai orang tua. Contoh perhitungannya:

- Ubah dulu nilai desimal tersebut ke dalam biner :

12 binernya 1100

3 binernya 0011

11 binernya 1011

15 binernya 1111

4 binernya 0100

8 binernya 1000

6 binernya 0110

Untuk Kelipatan 7 memiliki nilai biner untuk:

7 binernya 0111

14 binernya 1110

A handwritten binary addition problem on a light gray background. It shows the binary numbers 1100 and 1011 aligned vertically under a plus sign. A horizontal line with a plus sign above it separates the numbers from the result. The result, 0111, is written below the line. All digits are circled except for the first digit of each number.

$$\begin{array}{r} 1100 \\ 1011 \\ \hline 0111 \end{array}$$

Diatas adalah contoh xor dari 12 dan 11 yang menghasilkan nilai 7. Sehingga 11 dan 12 berada Dalam kelompok yang sama



Thank You

Link Video Demo

<https://youtu.be/kOmUuRcXx4Y>