

# KIỂM THỬ PHẦN MỀM

## KIỂM THỬ HỘP TRẮNG

ThS. Dương Hữu Thành  
Khoa CNTT, Đại học Mở Tp.HCM  
[thanh.dh@ou.edu.vn](mailto:thanh.dh@ou.edu.vn)

Software  
testing





# Nội dung chính

- 1. Tổng quan kiểm thử hộp trắng**
- 2. Đồ thị luồng**
- 3. Đường dẫn độc lập**
- 4. Độ phức tạp Cyclomatic**
- 5. Kiểm thử đường dẫn cơ sở**
- 6. Kiểm thử cấu trúc điều khiển**

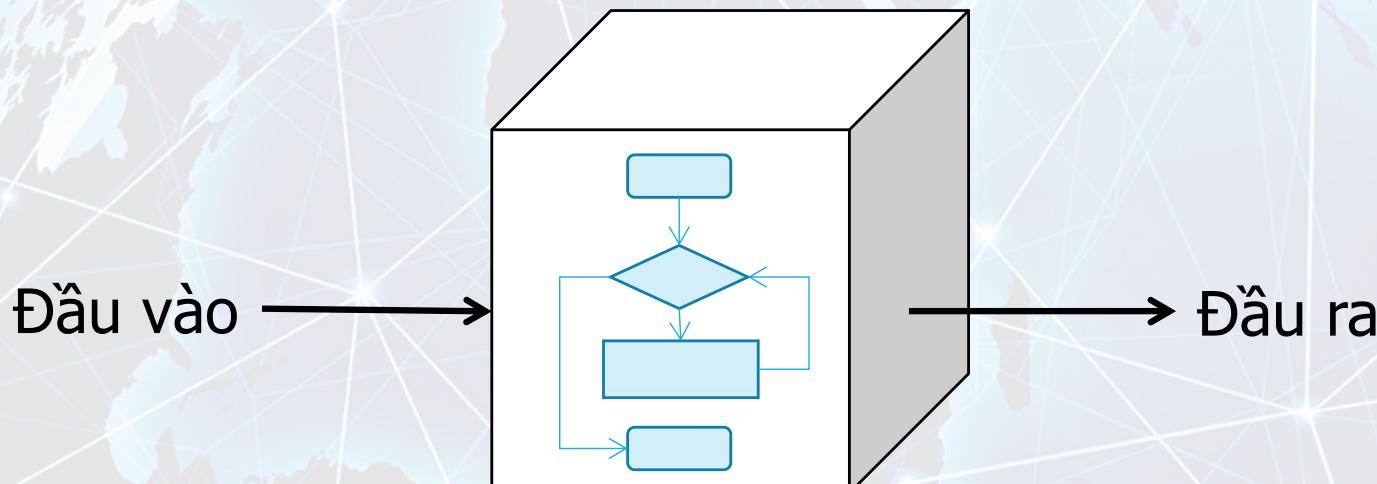


# Tổng quan kiểm thử hộp trắng



Kỹ thuật này còn gọi là structural testing, hoặc glass testing, hoặc open-box testing.

Tester cần biết cách thức (**HOW**) thực thi bên trong của phần mềm.





# Tổng quan kiểm thử hộp trắng



## Ưu điểm

**Dễ dàng xác định loại dữ liệu** để kiểm tra, nên việc kiểm tra hiệu quả hơn.

Nhờ biết mã nguồn, nên tester **có thể phủ tối đa** khi viết kịch bản kiểm thử.

## Khuyết điểm

**Chi phí tăng** vì tester cần có kỹ năng đọc và hiểu mã nguồn.

Khó xét hết các ngõ ngách trong mã nguồn nên **có thể sót đường dẫn** không được test.



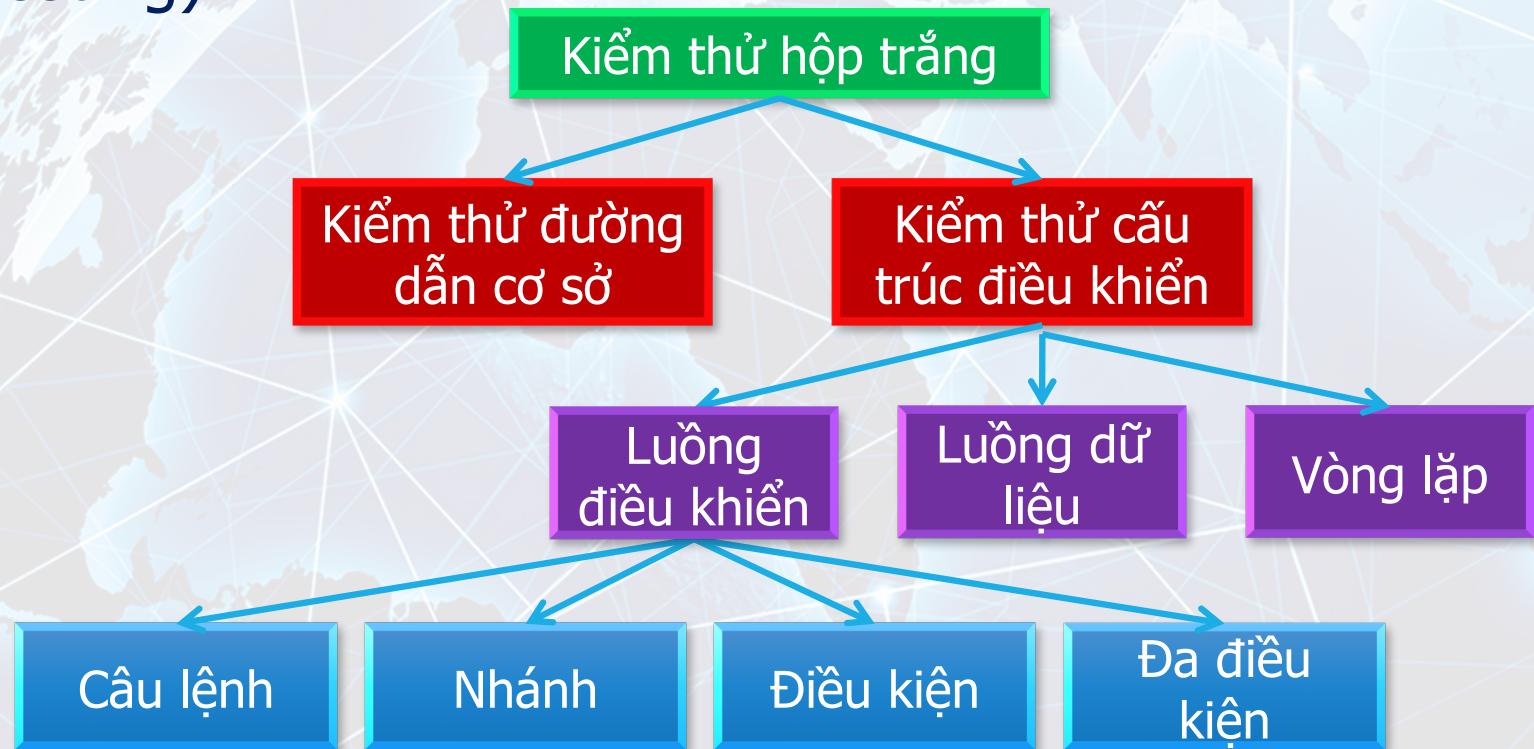
# Tổng quan kiểm thử hộp trắng



## Tiếp cận kiểm thử hộp trắng

Kiểm thử đường dẫn cơ sở (basis path testing)

Kiểm thử cấu trúc điều khiển (control structural testing)





# Đồ thị luồng



Đồ thị luồng (flow graph) là đồ thị có hướng, dùng mô tả **luồng điều khiển logic**.

Đồ thị luồng gồm hai thành phần cơ bản: **các đỉnh** tương ứng với các lệnh/nhóm câu lệnh **và** **các cạnh kết nối** tương ứng với dòng điều khiển giữa các câu lệnh/nhóm câu lệnh.

Các nút trong đồ thị luồng



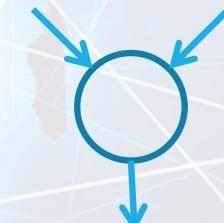
Nút bắt đầu



Khối xử lý



Nút vị từ hay  
nút quyết định



Nút nối

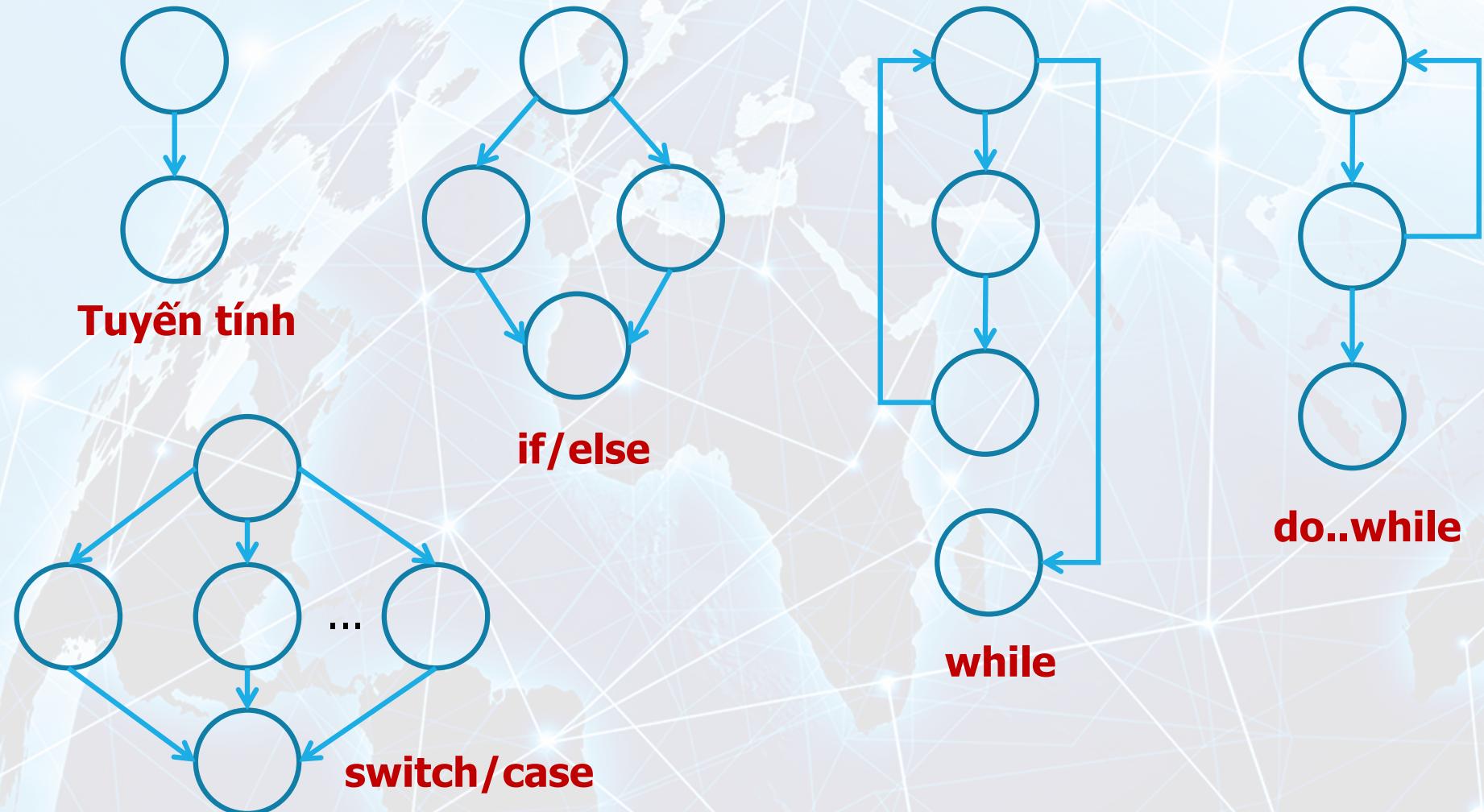


Nút kết thúc



# Đồ thị luồng

## Các đồ thị luồng cơ bản

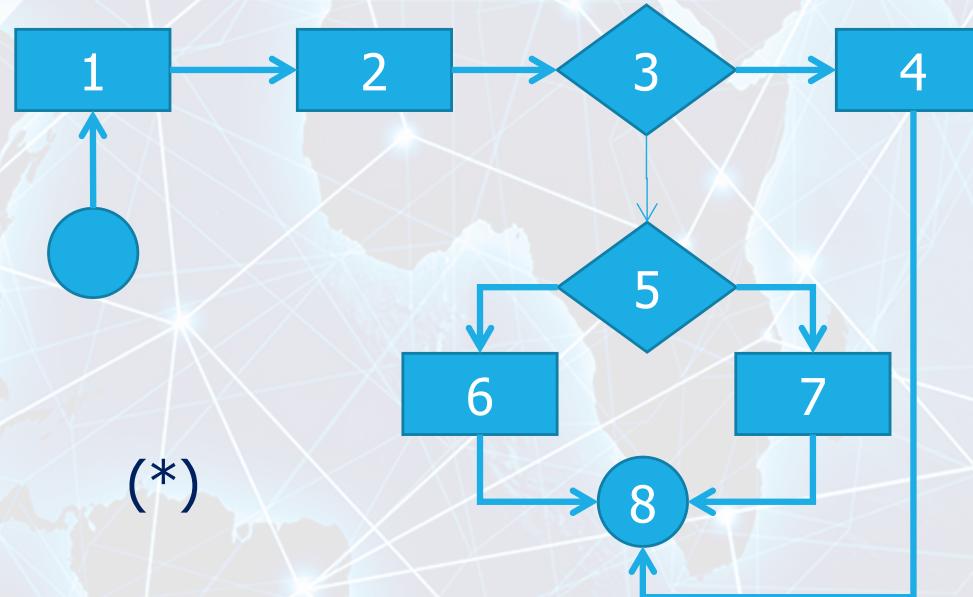




# Sơ đồ luồng → đồ thị luồng

**Đồ thị luồng** được xây dựng dựa trên **sơ đồ luồng** điều khiển (flow chart).

Sơ đồ luồng điều khiển mô tả cấu trúc điều khiển của chương trình.





# Sơ đồ luồng → đồ thị luồng



Các **hình chữ nhật** liên tiếp và **một hình thoi** (điều kiện) liên tiếp nhau trong sơ đồ luồng sẽ tạo thành một nút (node) trong đồ thị luồng.

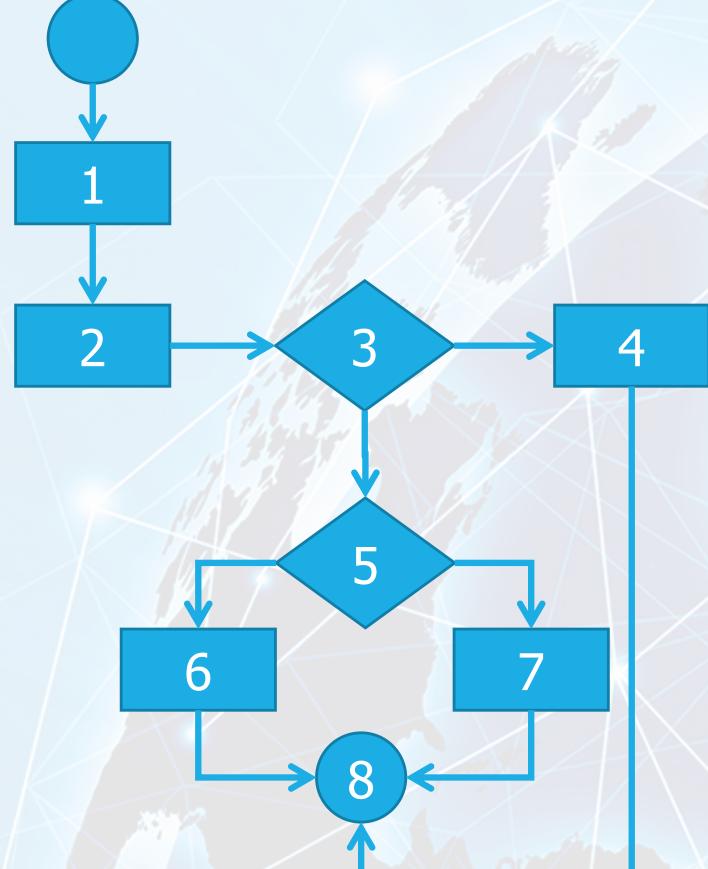
Từ điều kiện rẽ thành các cạnh (edge) nối tới các node khác.

Các cạnh và nút có thể hợp lại thành những **vùng khép kín** (region). Phạm vi bên ngoài đồ thị luồng cũng xem là một vùng.

Nút chứa biểu thức điều kiện gọi là **nút vị từ** (predicate) hay **nút quyết định** (node).



# Sơ đồ luồng → đồ thị luồng



Nút (Node)

4

→

1, 2, 3

5

6

8

Cạnh (Edge)

R<sub>2</sub>

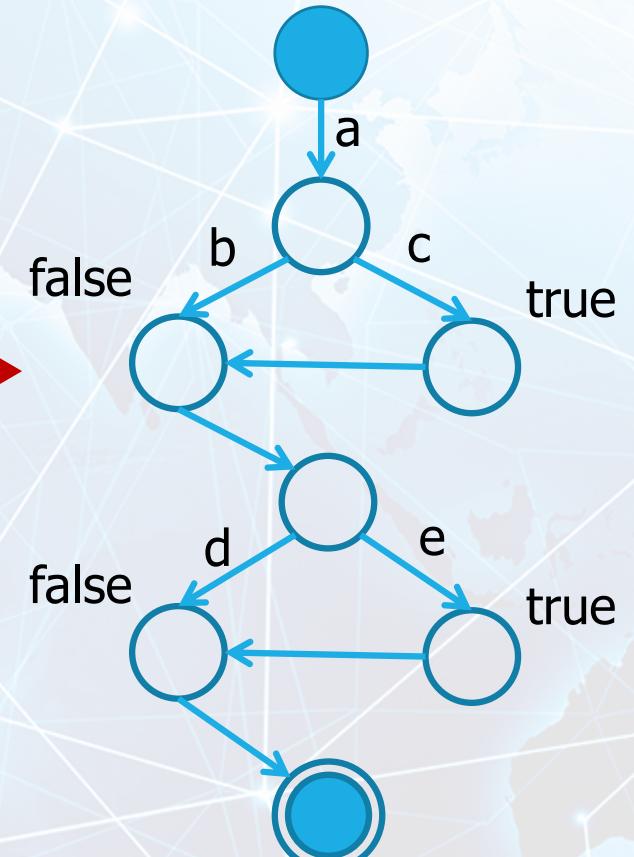
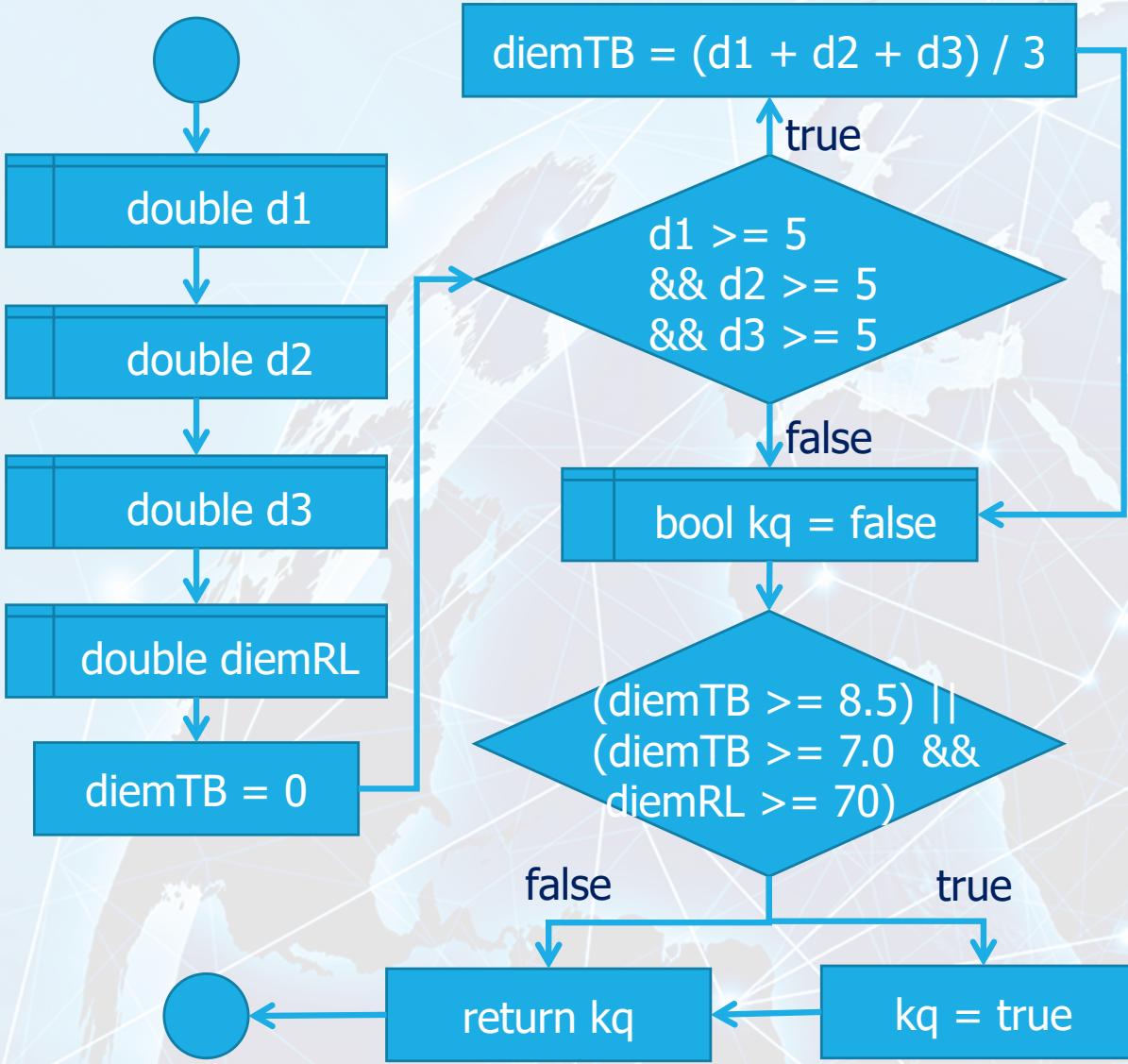
R<sub>1</sub>

R<sub>3</sub>

Vùng (region)



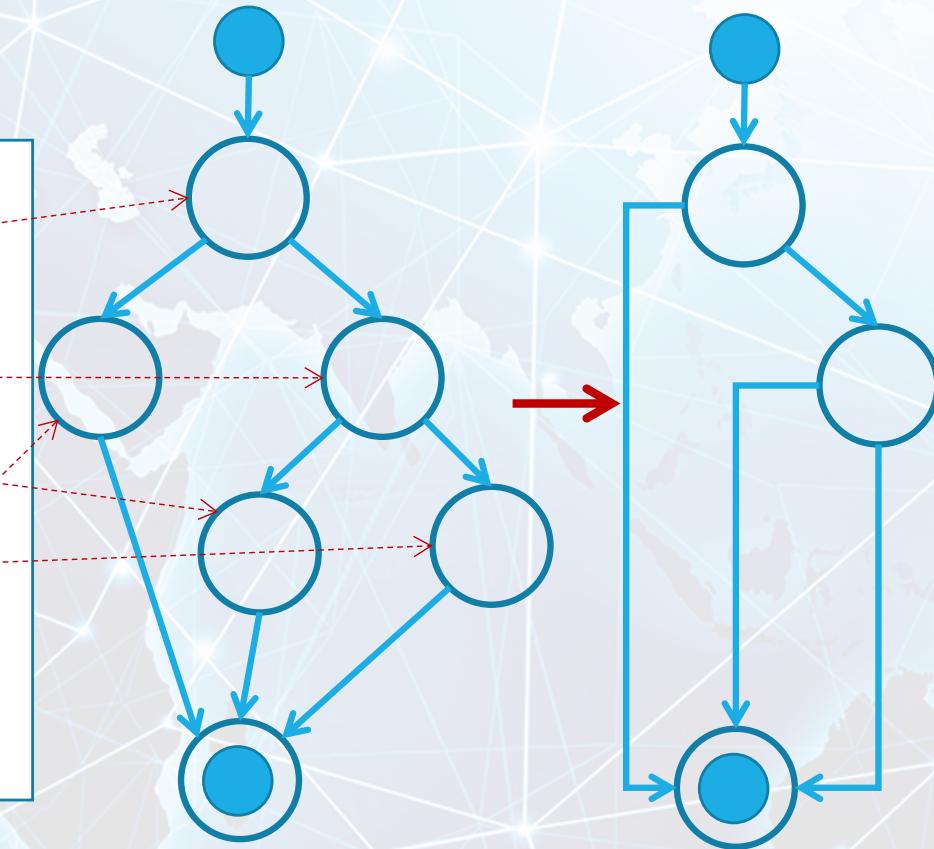
# Sơ đồ luồng → đồ thị luồng





# Sơ đồ luồng → đồ thị luồng

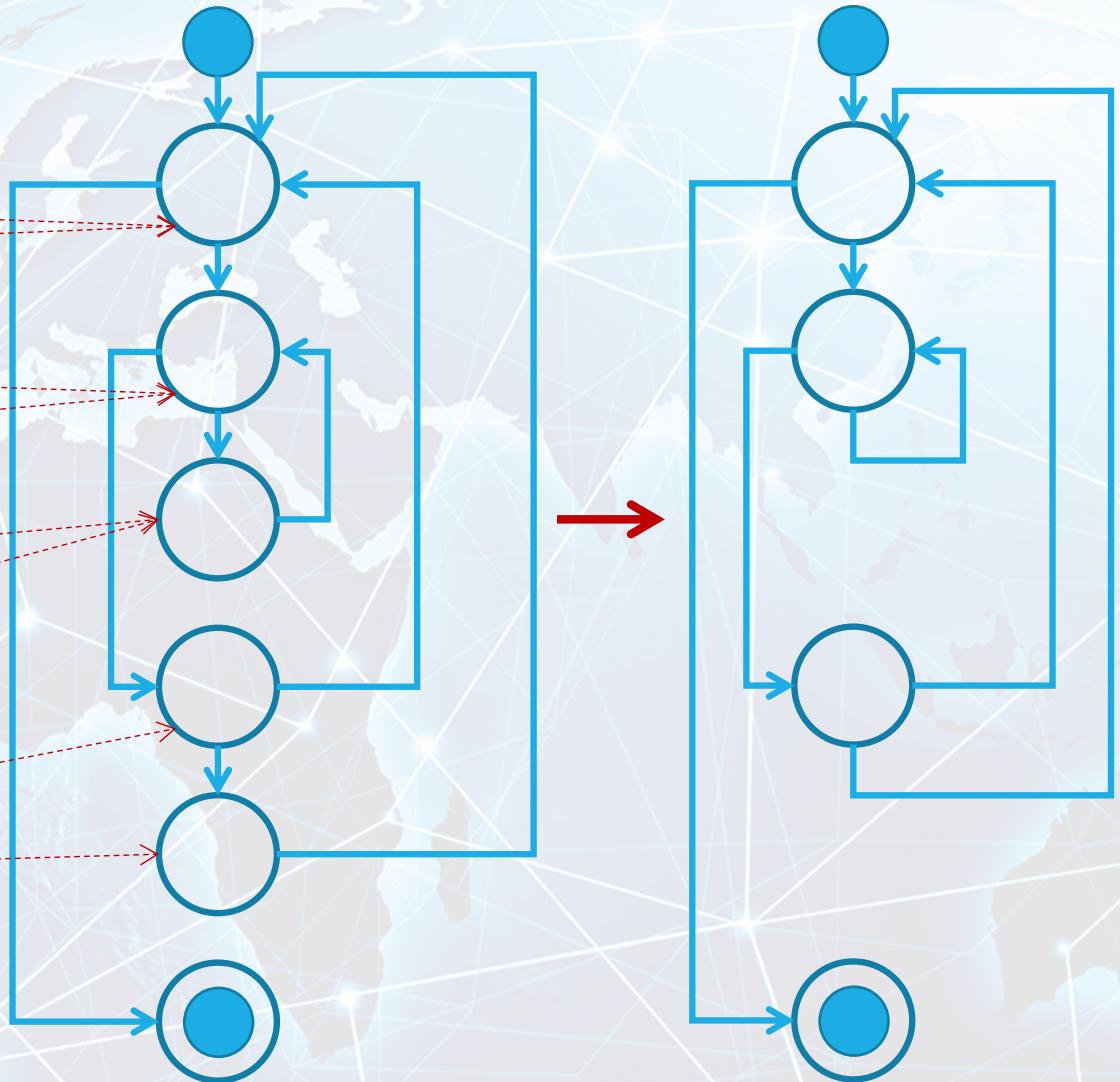
```
void giaiPTBacNhat(double a, double b)
{
    if (a == 0)
        if (b == 0)
            cout << "PT VSN\n";
        else
            cout << "PT VN\n";
    else
        cout << "x = " << -b / a;
}
```





# Sơ đồ luồng → đồ thị luồng

```
int tinhTong(int n)
{
    int tong = 0;
    while (n)
    {
        tong = 0;
        while (n)
        {
            tong += n % 10;
            n = n / 10;
        }
        if (tong > 10)
            n = tong;
    }
    return tong;
}
```





# Bài tập

Vẽ đồ thị luồng  
và xác định độ  
phức tạp  
Cyclomatic của  
đoạn chương  
trình sau:

```
void InsertionSort(int a[], int n)
{
    int x, idx;
    for (int i = 0; i < n - 1; i++)
    {
        if (a[i] > a[i + 1])
        {
            x = a[i + 1];
            idx = i + 1;
            do
            {
                a[idx] = a[idx - 1];
                idx--;
            } while (idx > 0 && a[idx - 1] > x);
            a[idx] = x;
        }
    }
}
```



# Đường dẫn độc lập

Đường dẫn độc lập là đường thông qua chương trình có **ít nhất** một tập các câu lệnh xử lý mới hoặc điều kiện mới.

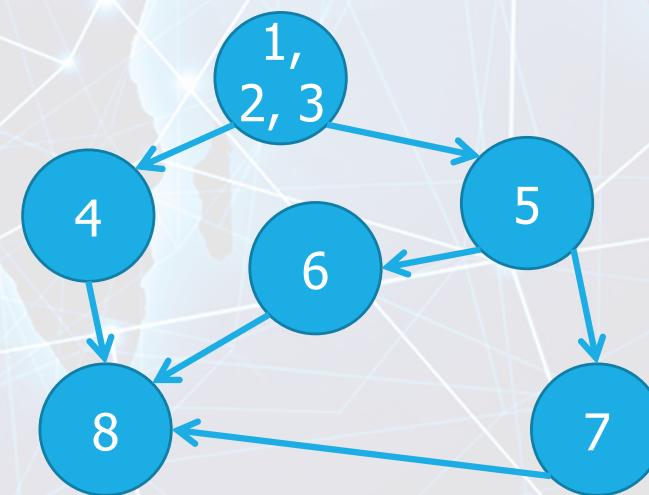
Trong đồ thị luồng, đường dẫn mới sẽ có cạnh mới.

Ví dụ: các đường dẫn độc lập cơ sở của đồ thị luồng bên dưới

1, 2, 3, 4, 8

1, 2, 3, 5, 6, 8

1, 2, 3, 5, 7, 8





# Đường dẫn độc lập



Nếu thiết kế test case thực thi được các đường dẫn độc lập, thì đảm bảo được:

Mỗi **câu lệnh** được thực thi **ít nhất một lần**.

Mỗi **biểu thức điều kiện** (kể cả trường hợp true và false) được thực thi **ít nhất một lần**.

Làm sao biết  
có bao nhiêu  
đường dẫn  
độc lập?

Độ phức tạp  
**Cyclomatic**





# Độ phức tạp Cyclomatic



Độ phức tạp Cyclomatic là một độ đo phần mềm cung cấp thước đo **định lượng độ phức tạp về mặt logic** của chương trình.

Trong ngữ cảnh của kiểm thử đường thì độ đo này là **số lượng các đường độc lập** trong tập cơ sở của chương trình và cung cấp **chặn trên** (upper bound) số lượng các test case được thực thi **đảm bảo phủ đường dẫn cơ sở** của chương trình.



# Độ phức tạp Cyclomatic



Độ đo này được tính bằng một trong ba cách sau (ký hiệu  $V(G)$  là giá trị độ phức tạp Cyclomatic của đồ thị luồng  $G$ ):

Số lượng các vùng (region) của  $G$

$$V(G) = E - N + 2$$

$E$  là số cạnh của  $G$

$N$  là số nút của  $G$

$$V(G) = P + 1$$

$P$  là số các nút vị từ của  $G$



# Độ phức tạp Cyclomatic

Ví dụ: đồ thị luồng như hình sau

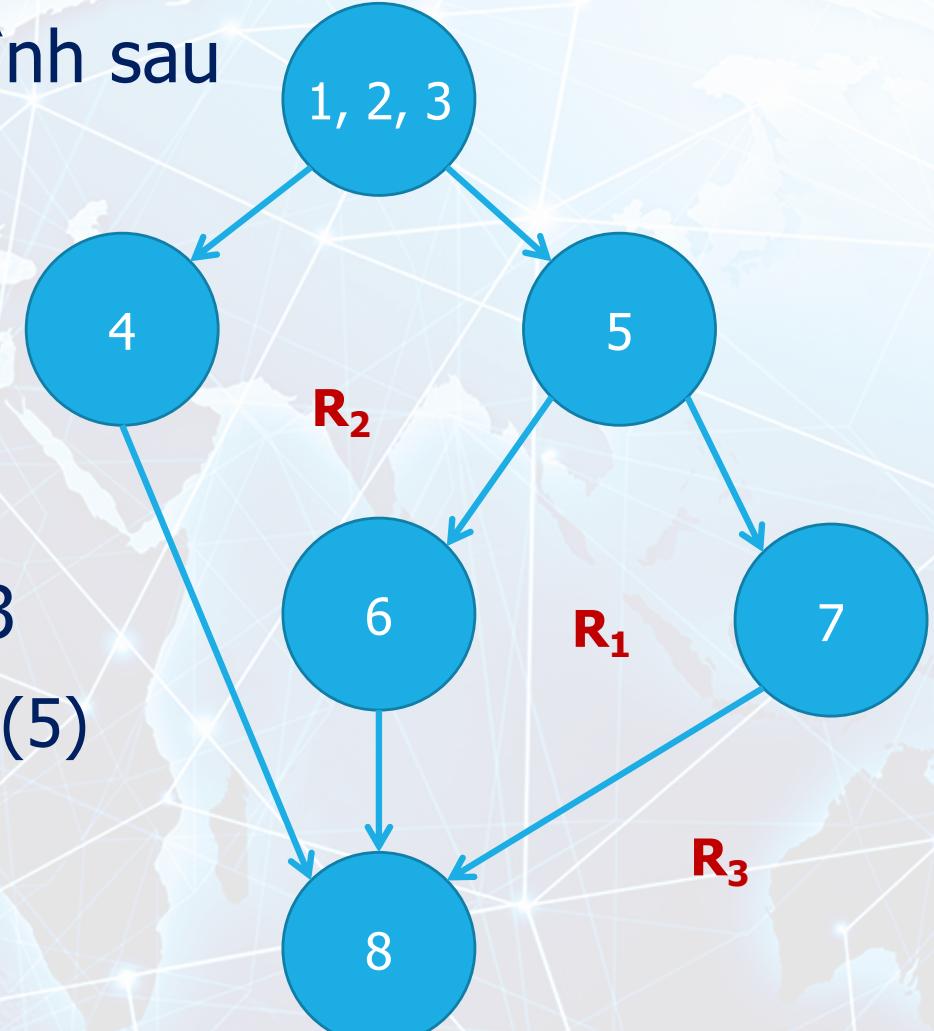
Có 3 region

$$V(G) = E - N + 2$$

$$= 7 - 6 + 2 = 3$$

$$V(G) = P + 1 = 2 + 1 = 3$$

2 nút vị từ là (1, 2, 3) và (5)



# Kiểm thử phần mềm

## KIỂM THỬ ĐƯỜNG DẪN CƠ CỎ



# Kiểm thử đường dẫn cơ sở



Kiểm thử đường dẫn cơ sở đảm bảo các **đường dẫn độc lập** được kiểm thử qua ít nhất một lần.

Các bước thực hiện:

Vẽ đồ thị luồng G.

Tính độ phức tạp Cyclomatic  $V(G)$ .

Xác định tập cơ sở các đường dẫn độc lập.

Viết test case thực thi mỗi đường dẫn trong tập cơ sở.



# Ví dụ 1

Viết các test case kiểm thử chương trình giải và biện luận phương trình bậc nhất  $ax + b = 0$ , trong đó  $a, b$  là các số thực nhập từ bàn phím.

Chương trình như sau:

```
void giaiPTBacNhat(double a, double b)
{
    if (a == 0)
        if (b == 0)
            cout << "PT vo so nghiem\n";
        else
            cout << "PT vo nghiem\n";
    else
        cout << "Nghiem x = " << -b / a << endl;
}
```



*Chương trình minh họa bằng C++*



# Ví dụ 1

## Xác định độ phức tạp Cyclomatic

Số lượng các vùng của G: 3

$$V(G) = E - N + 2 = 5 - 4 + 2 = 3$$

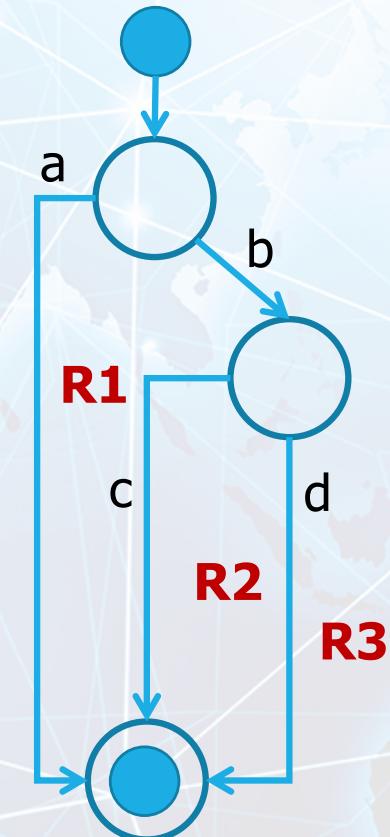
$$V(G) = P + 1 = 2 + 1 = 3$$

## Xác định tập đường dẫn cơ sở

a

b, c

b, d





# Ví dụ 1



Thiết kế test case cho từng đường dẫn cơ sở

Đường dẫn	Đầu vào		Đầu ra mong muốn
	a	b	
a	5	-10	Nghiem $x = 2$
b, c	0	0	PT vo so ngleim
b, d	0	5	PT vo ngleim



## Ví dụ 2

Viết các test case để kiểm thử chương trình tính tổng các chữ số của số nguyên dương n cho đến khi tổng nhỏ hơn 10.

Ví dụ:

$$\begin{aligned}123456 &\rightarrow 1 + 2 + 3 + 4 + 5 + 6 = 21 \rightarrow 2 + 1 \\&\rightarrow 3 < 10\end{aligned}$$

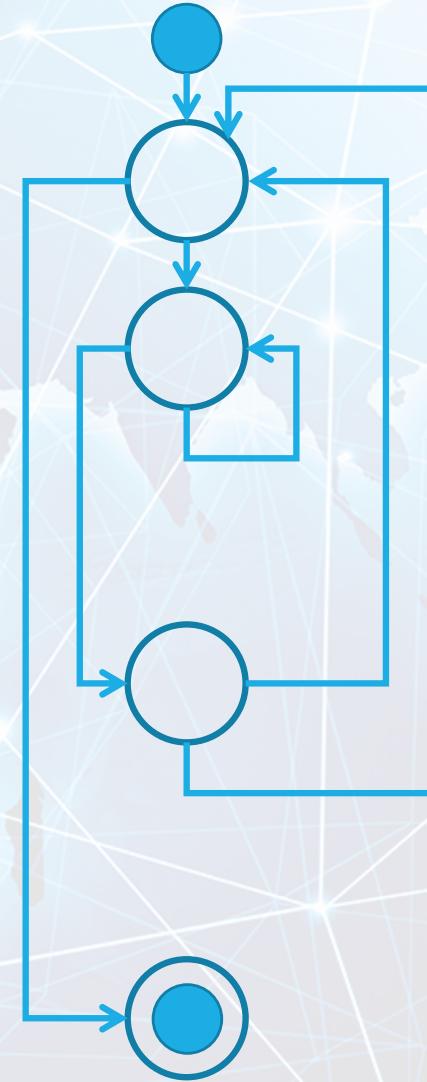
$$857 \rightarrow 8 + 5 + 7 = 20 \rightarrow 2 + 0 = 2 < 10$$



## Ví dụ 2

```
int tinhTong(int n)
{
    int tong = 0;
    while (n)
    {
        tong = 0;
        while (n)
        {
            tong += n % 10;
            n = n / 10;
        }

        if (tong > 10)
            n = tong;
    }
    return tong;
}
```



*Chương trình minh họa bằng C++*



## Ví dụ 2

Xác định độ phức tạp Cyclomatic

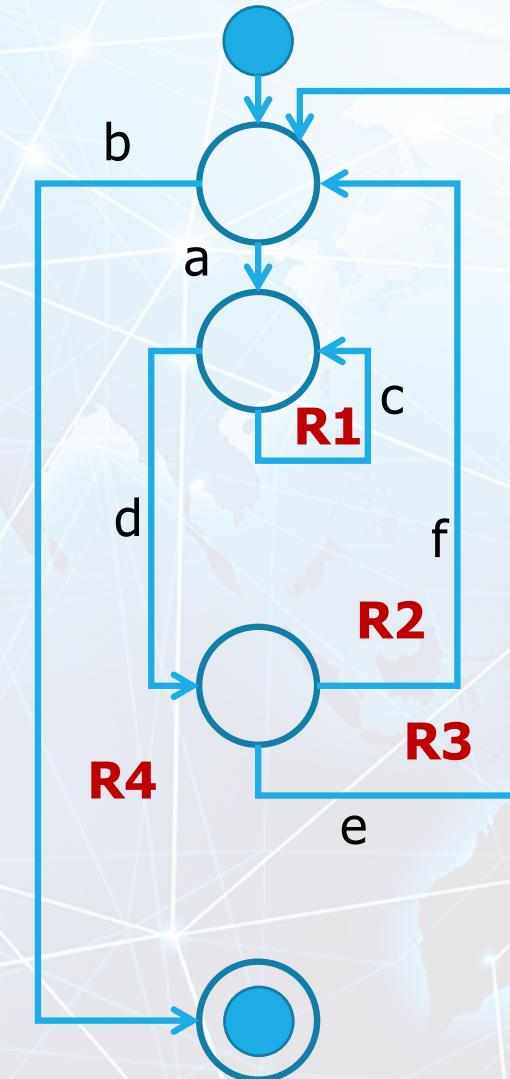
Số lượng các vùng của G: 4

$$V(G) = E - N + 2 = 7 - 5 + 2 = 4$$

$$V(G) = P + 1 = 3 + 1 = 4$$

Chọn tập đường dẫn cơ sở

a, c, c, d, e, a, c, c, d, f, b





## Ví dụ 2

Thiết kế test case cho từng đường dẫn cơ sở

Đường dẫn	Đầu vào	Đầu ra mong muốn
	n	
a, c, c, d, e, a, c, c, d, f, b	87	Tổng là 6



# Bài tập 1

Thiết kế các test case để phủ đường dẫn cơ sở của đoạn chương trình sau:

```
bool xetHocBong(double diemMH[], int soMH, int diemRL)
{
    double tongDiem = 0;
    for (int i = 0; i < soMH; i++)
        if (diemMH[i] < 5)
            return false;
        else
            tongDiem += diemMH[i];
    double diemTB = tongDiem / soMH;
    if (diemTB >= 9 || (diemTB >= 7 && diemRL >= 80))
        return true;
    return false;
}
```



## Bài tập 2

```
int BinarySearch(int a[], int n, int x)
{
    int middle, left = 0, right = n - 1, idx = -1;
    while (left <= right)
    {
        middle = (left + right) / 2;
        if (a[middle] == x)
        {
            idx = middle;
            break;
        }
        else if (a[middle] > x)
            right = middle - 1;
        else
            left = middle + 1;
    }
    return idx;
}
```





# Bài tập 3

```
1 double tinhDiem(double diem[], int n)
2 {
3     if (n < 3)
4         return 0;
5
6     double tb = 0;
7     for (int i = 0; i < n; i++)
8         tb += diem[i];
9     tb /= n;
10
11    double kq = 0;
12    int dem = 0;
13    for (int i = 0; i < n; i++)
14        if (abs(diem[i] - tb) <= 20)
15        {
16            kq += diem[i];
17            dem++;
18        }
19
20    return kq/dem;
21 }
```



```
bool check(int a[], int n, int k)
{
    if (k <= 1 || k > n)
        return false;

    int dem = 1;
    for (int i = 0; i < n - 1 && dem != k; i++)
        if (a[i] > a[i+1])
            dem = 1;
        else
            dem++;

    if (dem == k)
        return true;

    return false;
}
```

# Kiểm thử phần mềm

KIỂM THỬ  
CẤU TRÚC ĐIỀU KHIỂN



# Kiểm thử cấu trúc điều khiển



Kiểm thử luồng điều khiển (Control Flow Testing)  
hoặc bao phủ (Coverage Testing)

Kiểm thử luồng dữ liệu (Data Flow Testing)

Kiểm thử vòng lặp (Loop Testing)



# Kiểm thử bao phủ



Kiểm thử bao phủ dùng kiểm tra mức độ phủ (coverage) của các test case.

Các loại kiểm thử bao phủ:

Phủ **câu lệnh** (statement coverage)

Phủ **nhánh** (branch coverage)

Phủ **đường** (path coverage)

Phủ **điều kiện** (condition coverage)

Phủ **đa điều kiện** (multi-conditions coverage)



# Phủ câu lệnh

Phủ câu lệnh (statement coverage): **mỗi câu lệnh** được thực thi ít nhất một lần.

Ví dụ: hàm xét học bổng như sau

```
bool xetHocBong(double d1, double d2, double d3, double diemRL)
{
    double diemTB = 0.0;
    if (d1 >= 5 && d2 >= 5 && d3 >= 5)
        diemTB = (d1 + d2 + d3) / 3;

    bool kq = false;
    if ((diemTB >= 8.5) || (diemTB >= 7.0 && diemRL >= 70))
        kq = true;

    return kq;
}
```

*Chương trình minh họa bằng C++*



# Phủ câu lệnh



Để kiểm thử phủ câu lệnh trong hàm trên chỉ cần test case sau:

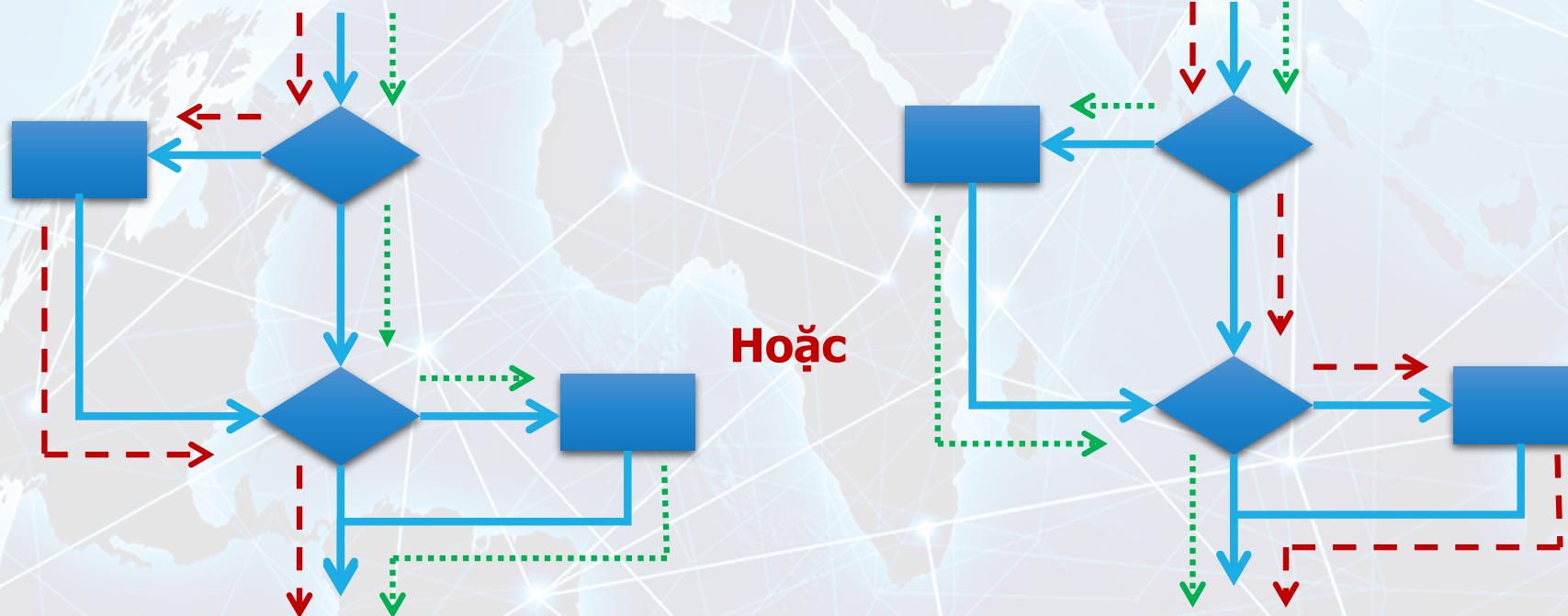
Đầu vào				Đầu ra mong muốn
d1	d2	d3	diemRL	
7	7	7	70	Hàm trả về kết quả true



# Phủ nhánh

Phủ nhánh (branch coverage): **mỗi nhánh** phải được thực hiện ít nhất một lần.

Phủ nhánh **đảm bảo** phủ câu lệnh.





# Phủ nhánh



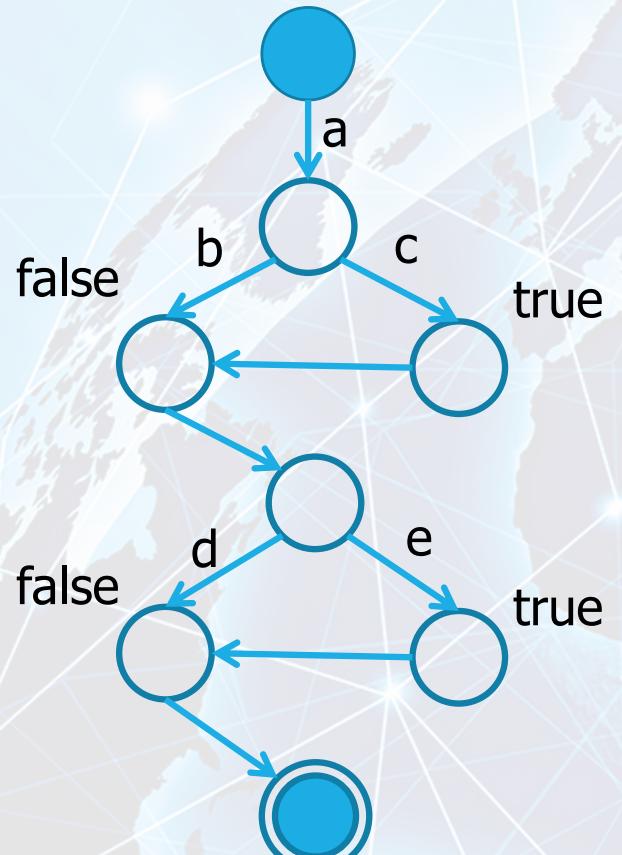
Nhận xét: trong **đồ thị luồng**

Phủ nhánh có nghĩa là **các cạnh được đi qua ít nhất một lần**.

Để phủ nhánh phải thiết kế dữ liệu kiểm thử sao cho **mỗi nút vị từ** (predicate) xảy ra tất cả các kết quả (true/false) có thể của nó, nên phủ nhánh còn gọi là **phủ quyết định** (decision coverage).



# Phủ nhánh



Để phủ các nhánh:

- Hoặc **abd** (FF) và **ace** (TT)
- Hoặc **abe** (FT) và **acd** (TF)



# Phủ nhánh

Chọn abd và ace kiểm thử phủ nhánh, thiết kế các test case như sau:

Nhánh	Đầu vào				Đầu ra mong muốn
	d1	d2	d3	diemRL	
abd	4	5	5	70	Hàm trả về kết quả false
ace	7	7	7	70	Hàm trả về kết quả true

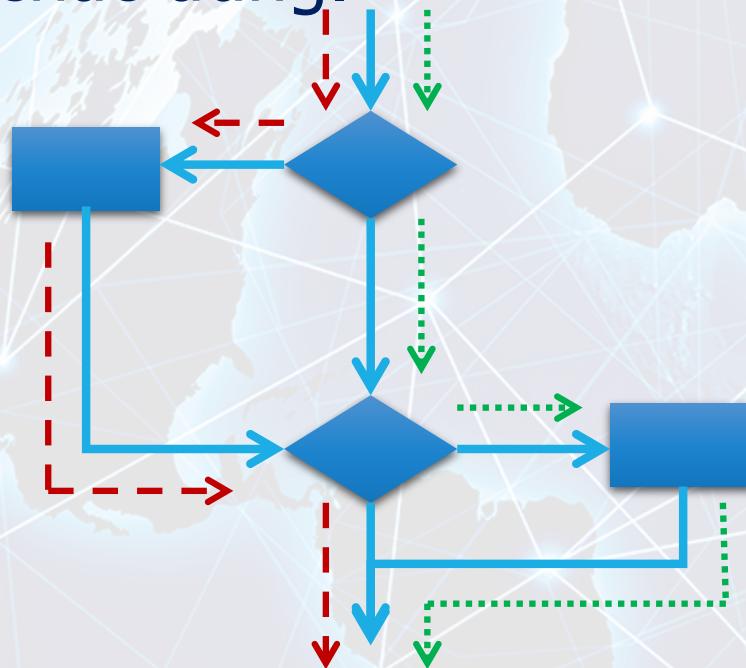


# Phủ đường dẫn

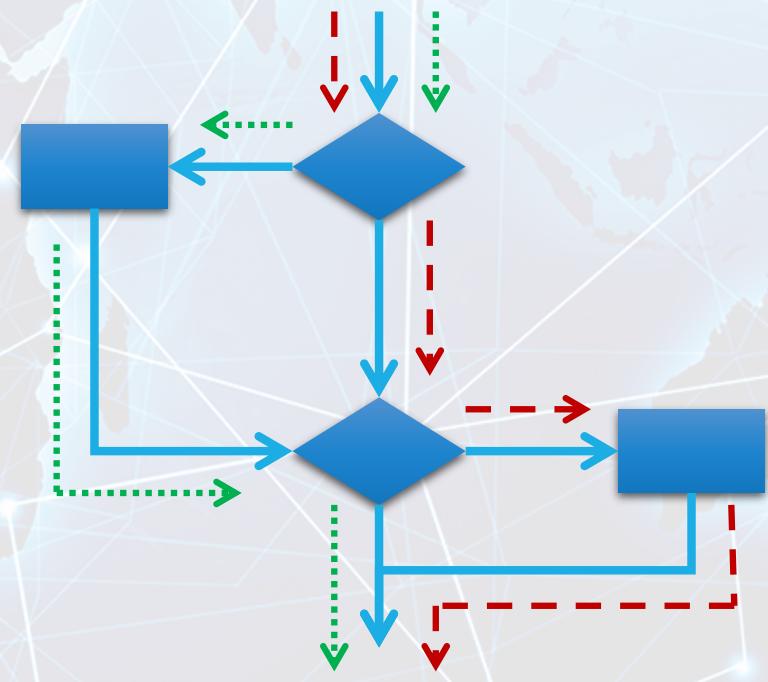


Phủ đường dẫn (path coverage): mỗi đường dẫn qua ít nhất một lần.

Đường là một tập các nhánh, nên **phủ đường chắc chắn phủ nhánh**, nhưng ngược lại chưa chắc đúng.



Và



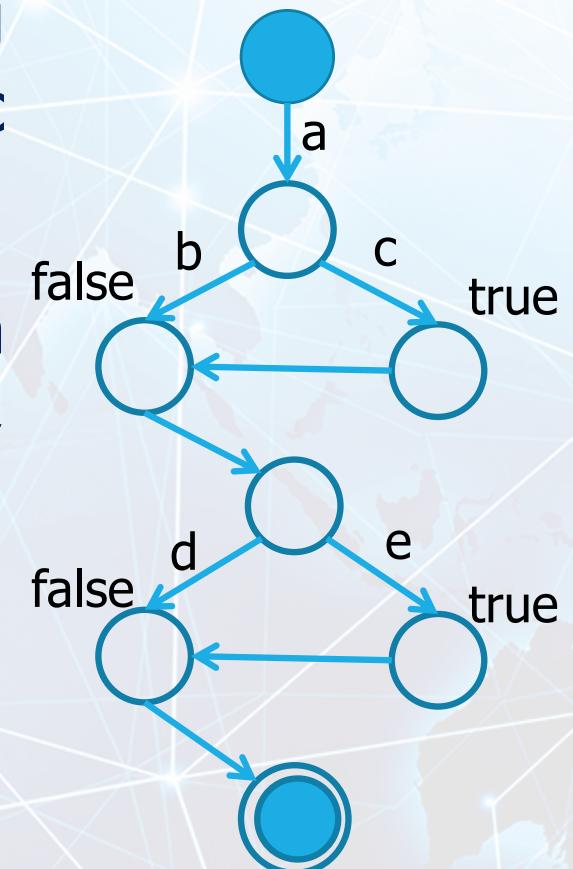


# Phủ đường dẫn

Ví dụ: quay lại ví dụ xét học bổng, phủ đường dẫn trong đồ thị luồng bằng 4 test case sau đi qua các đường abd, ace, acd, abe.

Chú ý: không có test case nào đi qua đường abe (đường bất khả thi). Do đó, để phủ đường chỉ cần 3 test case sau

Đầu vào				Đầu ra mong muốn
d1	d2	d3	diemRL	
4	5	5	70	Hàm trả về kết quả false
7	7	7	70	Hàm trả về kết quả true
7	7	7	60	Hàm trả về kết quả false





# Phủ điều kiện



Thông thường vị từ quyết định nhánh sẽ được thực thi là **tổ hợp nhiều điều kiện**.

Ví dụ:

```
if (d1 >= 5 && d2 >= 5 && d3 >= 5) ...  
if ((nam % 400 == 0) ||  
(nam % 4 == 0 && nam % 100 != 0)) ...
```

Phủ điều kiện (condition coverage): **mỗi điều kiện trong các vị từ** được thực hiện ít nhất một lần cho cả trường hợp true và false (không bắt buộc các kết hợp giữa chúng).



# Bài tập



Số test case tối thiểu để phủ câu lệnh?

Số test case tối đa phủ đường dẫn cơ sở?

Số test case tối thiểu phủ điều kiện? Xác định các test case phủ điều kiện?

Thiết kế test case phủ đường dẫn cơ sở?

```
int reverse(int n)
{
    bool isNegative = false;
    if (n < 0)
    {
        isNegative = true;
        n = -n;
    }

    int result = 0;
    while (n > 0)
    {
        result = result * 10 + n % 10;
        n /= 10;
    }
    if (isNegative == true)
        result = - result;
    return result;
}
```



# Phủ nhánh và điều kiện

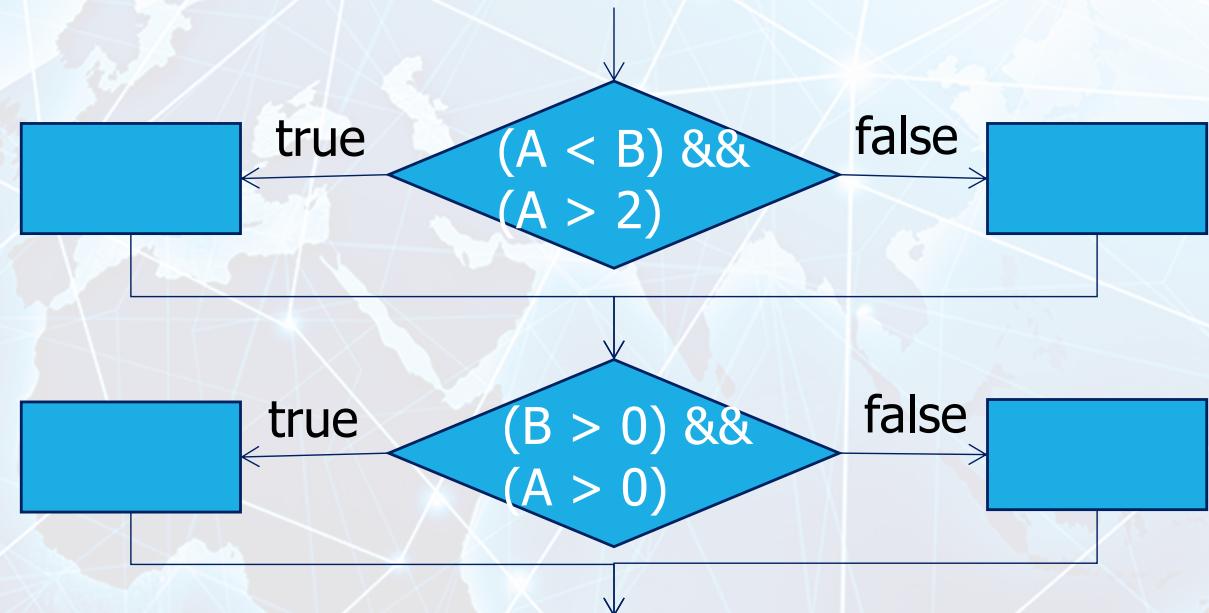
Ví dụ: phủ điều kiện trong sơ đồ luồng sau với các test case

A = 3, B = 4

A = -3, B = 4

A = -3, B = -4

Làm sao xác định các test case đó?





# Phủ nhánh và điều kiện

$\rightarrow (A < B) \&\& (A > 2)$

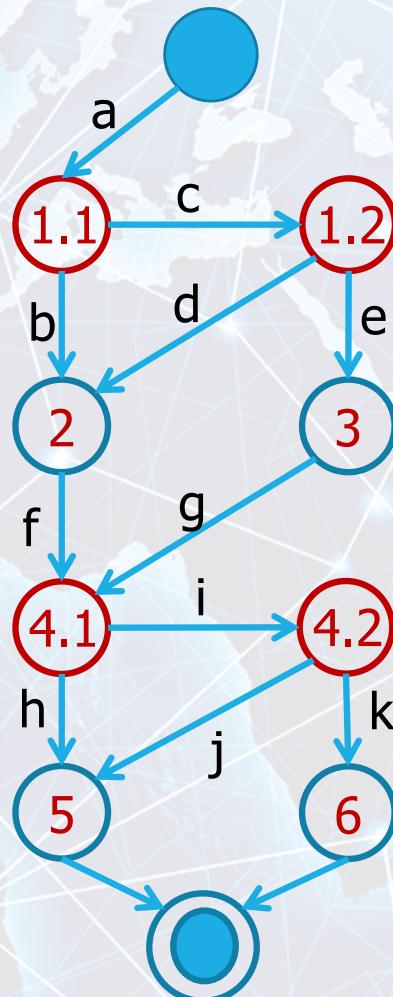
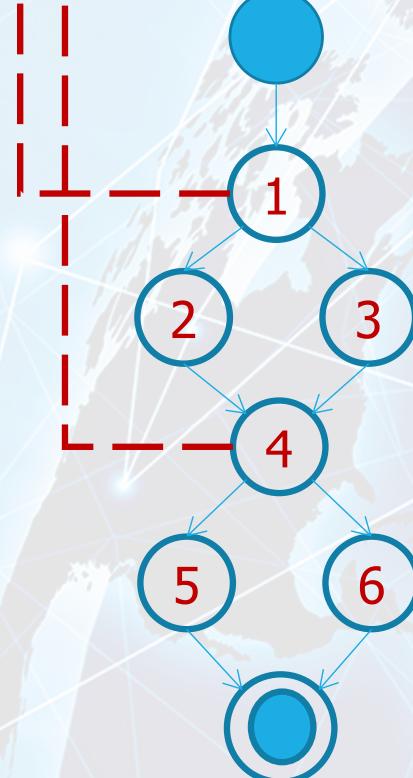
1.1

1.2

$\rightarrow (A > 0) \&\& (B > 0)$

4.1

4.2



Ta chọn các **đường  
dẫn phủ đường dẫn  
cơ sở** của đồ thị này:

- a, c, e, g, i, k
- a, c, d, f, i, j
- a, b, f, h

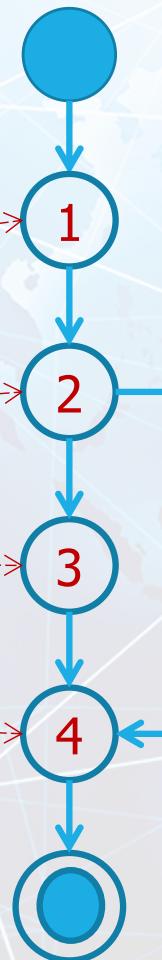


# Phủ nhánh và điều kiện

Ví dụ: thiết kế các test case phủ nhánh và điều kiện của hàm kiểm tra năm nhuận.

```
bool ktNamNhuan(int nam)
{
    bool kq = false;
    if ((nam % 400 == 0) ||
        (nam % 4 == 0 && nam % 100 != 0))
        kq = true;

    return kq;
}
```





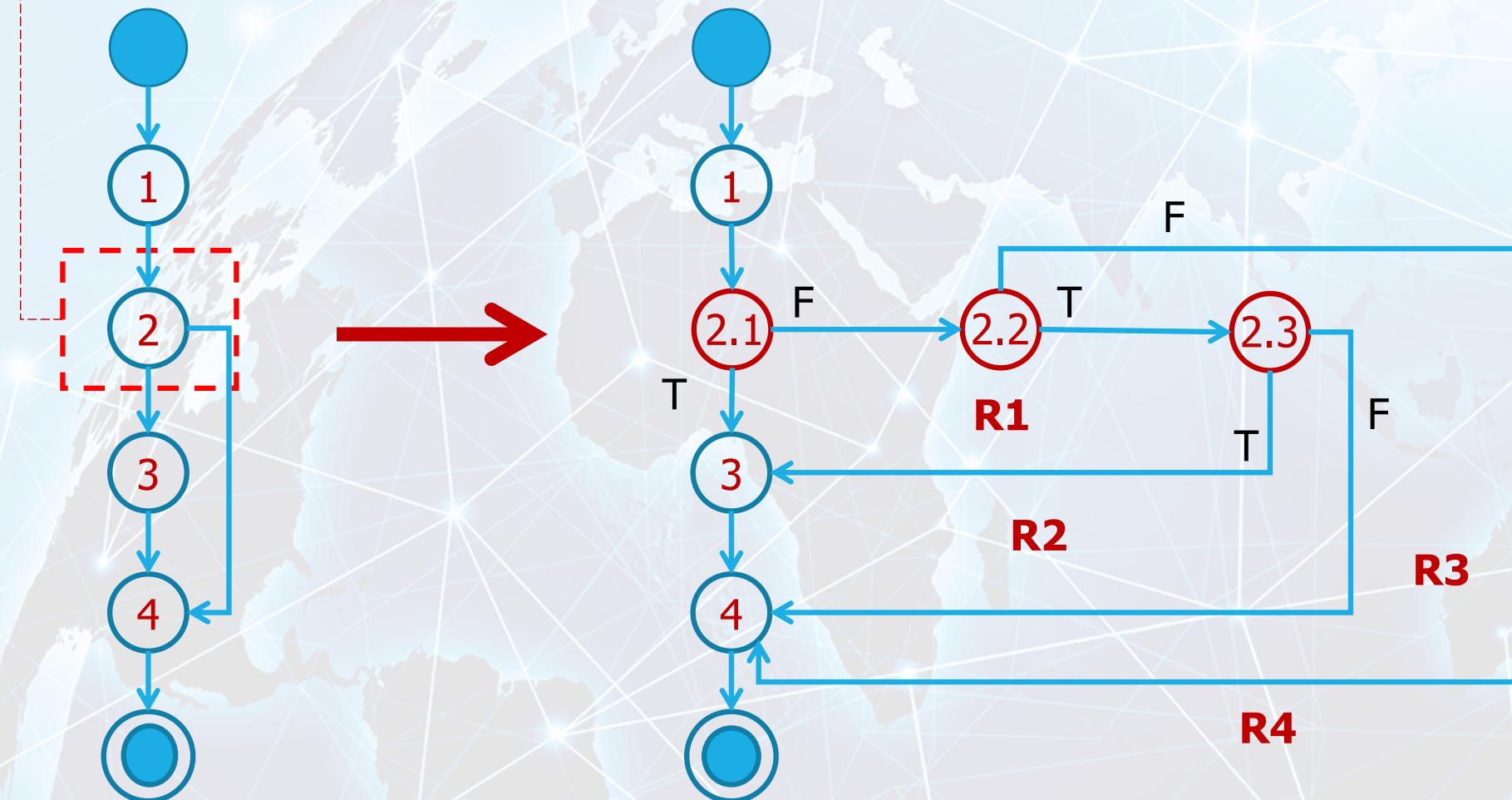
# Phủ nhánh và điều kiện

→  $(\text{nam \% } 400 == 0) \text{ || } (\text{nam \% } 4 == 0 \text{ && nam \% } 100 != 0)$

2.1

2.2

2.3





# Phủ nhánh và điều kiện

Độ phức tạp Cyclomatic:

Số lượng các vùng của G: 4

$$V(G) = E - N + 2 = 10 - 8 + 2 = 4$$

$$V(G) = P + 1 = 3 + 1 = 4$$

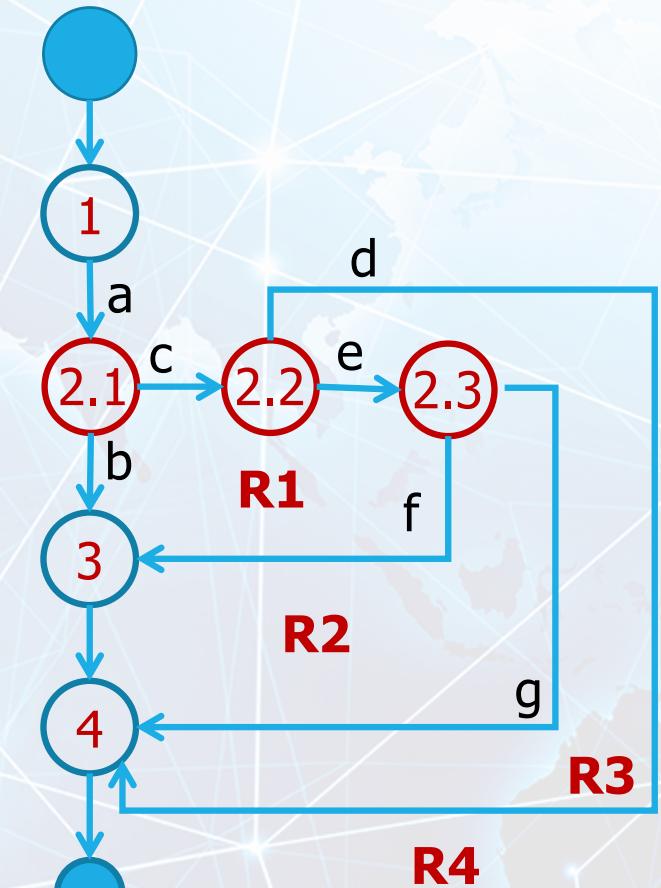
Các đường dẫn sau đảm bảo phủ nhánh và điều kiện:

a, c, e, f

a, c, e, g

a, c, d

a, b





# Phủ nhánh và điều kiện

Để phủ nhánh và điều kiện cần tối thiểu các test case minh họa:

<b>Đường dẫn</b>	<b>Đầu vào</b>	<b>Đầu ra mong muốn</b>
	<b>Năm</b>	
a, c, e, f	2016	Năm nhuận
a, c, e, g	1900	Không phải năm nhuận
a, c, d	2017	Không phải năm nhuận
a, b	1600	Năm nhuận



# Bài tập 1



Viết các test case phủ nhánh và điều kiện hàm:

```
bool xetHocBong(double diemMH[], int soMH, int diemRL)
{
    if (soMH > 0)
    {
        int i;
        double tongDiem = 0;
        for (i = 0; i < soMH; i++)
            tongDiem = tongDiem + diemMH[i];

        double diemTB = tongDiem / soMH;
        if (diemTB >= 8 || (diemTB >= 7 && diemRL >= 80))
            return true;
    }

    return false;
}
```



# Bài tập 2

Viết các test case phủ nhánh và điều kiện hàm:

```
bool checkLottery(string result, string number)
{
    bool k = false;
    if (result.Length == 8 && number.Length == 8)
    {
        int count = 0;
        for (int i = 0; i < result.Length && count < 6; i++)
            if (result[i] == number[i])
                count++;
            else
                count = 0;

        k = count == 6;
    }
    return k;
}
```

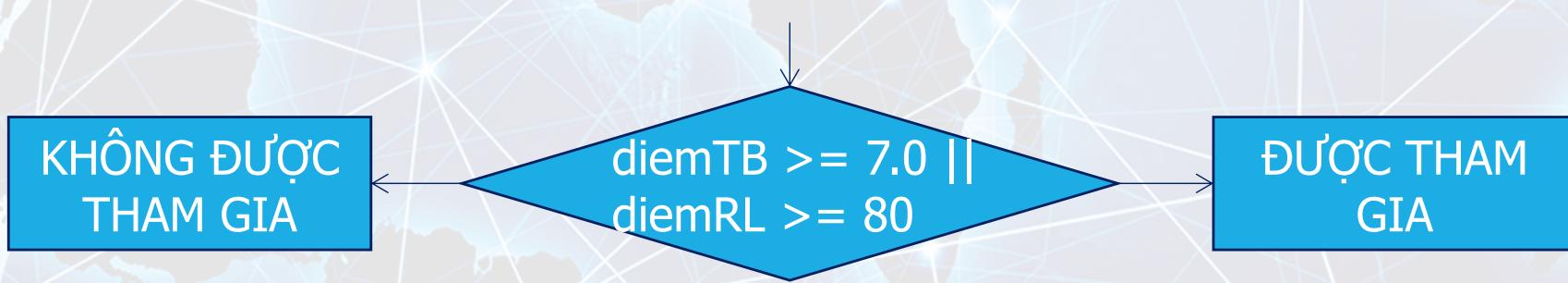


# Phủ đa điều kiện



Phủ đa điều kiện (multi-conditions coverage): mỗi điều kiện trong biểu thức vị từ và **các kết hợp** giữa chúng được thực hiện ít nhất một lần cho các trường hợp true và false.

Ví dụ: một sinh viên được xét tham gia chiến dịch tình nguyện của trường nếu hoặc điểm trung bình từ 7.0 trở lên hoặc điểm rèn luyện từ 80 trở lên.





# Phủ đa điều kiện



Với các test case sau đảm bảo phủ điều kiện:

`diemTB = 7.0 (true) và diemRL = 80 (true)`

`diemTB = 6.5 (false) và diemRL = 75 (false)`

Để phủ đa điều kiện cần bổ sung các test case:

`diemTB = 6.5 (false) và diemRL = 80 (true)`

`diemTB = 7.0 (true) và diemRL = 75 (false)`

Ghi chú: với các ngôn ngữ lập trình hiện đại thì test case này không thể xảy ra vì `diemTB = 7.0 (true)` đủ quyết định kết quả biểu thức nên nó sẽ không xét biểu thức con còn lại.



# Kiểm thử luồng dữ liệu

Phương pháp kiểm thử luồng dữ liệu sẽ kiểm thử **vòng đời của biến** trong từng luồng thực thi của chương trình.

Vòng đời của một biến được thể hiện thông qua ba hành động:

Định nghĩa biến (**Define**).

Sử dụng biến (**Use**).

Xóa biến (**Delete**).



# Kiểm thử luồng dữ liệu

Kiểm thử luồng dữ liệu lựa chọn các đường dẫn để kiểm thử dựa trên **vị trí** định nghĩa (**define**) và sử dụng (**use**) các biến trong chương trình.

**Định nghĩa DEF(*v, n*):** DEF(*v, n*) biến *v* được định nghĩa trong câu lệnh nào đó của nút *n* trong đồ thị luồng. Các lệnh nhập, lệnh gán, lệnh gọi thủ tục là các lệnh định nghĩa biến.



# Kiểm thử luồng dữ liệu



**Định nghĩa USE( $v, n$ ):** USE( $v, n$ ) giá trị của biến  $v$  được sử dụng trong một lệnh nào đó của nút  $n$  trong đồ thị luồng. Các lệnh xuất dữ liệu, tính toán, các biểu thức điều kiện, các lệnh trả về trong hàm, phương thức là các lệnh sử dụng biến.

Nếu câu lệnh sử dụng biến  $v$  là biểu thức vị từ thì ký hiệu là p-use (predicate use).

Nếu câu lệnh sử dụng biến  $v$  là biểu thức tính toán thì ký hiệu là c-use (computation use).



# Kiểm thử luồng dữ liệu

## Ví dụ

Định nghĩa biến a

Sử dụng (p-use)  
biến a

Sử dụng (c-use)  
biến a

Định nghĩa biến b

Sử dụng (p-use)  
biến b

Sử dụng (c-use)  
biến b

```
int a, b;  
cin >> a >> b;  
  
while (a != b)  
{  
    if (a > b)  
        a -= b;  
    else  
        b -= a;  
  
    cout << "UCLN(a, b) = " << a;
```

*Chương trình minh họa bằng C++*



# Kiểm thử luồng dữ liệu



Kiểm thử luồng dữ liệu giúp phát hiện các vấn đề sau:

Một biến được **khai báo, nhưng không sử dụng.**

Một biến **sử dụng nhưng không khai báo.**

Một biến được **định nghĩa nhiều lần** trước khi sử dụng.

**Xóa biến trước khi sử dụng.**



# Kiểm thử luồng dữ liệu

Cho biến  $x \in \text{DEF}(S) \cap \text{USE}(S')$ , trong đó S và S' là các câu lệnh.

**Đường DC (definition-clear path)** của biến  $x$  là đường nối từ S đến S' trên đồ thị luồng sao cho **không tồn tại** một định nghĩa nào khác của  $x$  trên đường này.

**Cặp DU** (definition-use pairs) của biến là cặp S và S', sao cho **tồn tại ít nhất** một đường DC nối S và S'.



# Kiểm thử luồng dữ liệu

$\text{DEF}(1) = \{x, \dots\}$

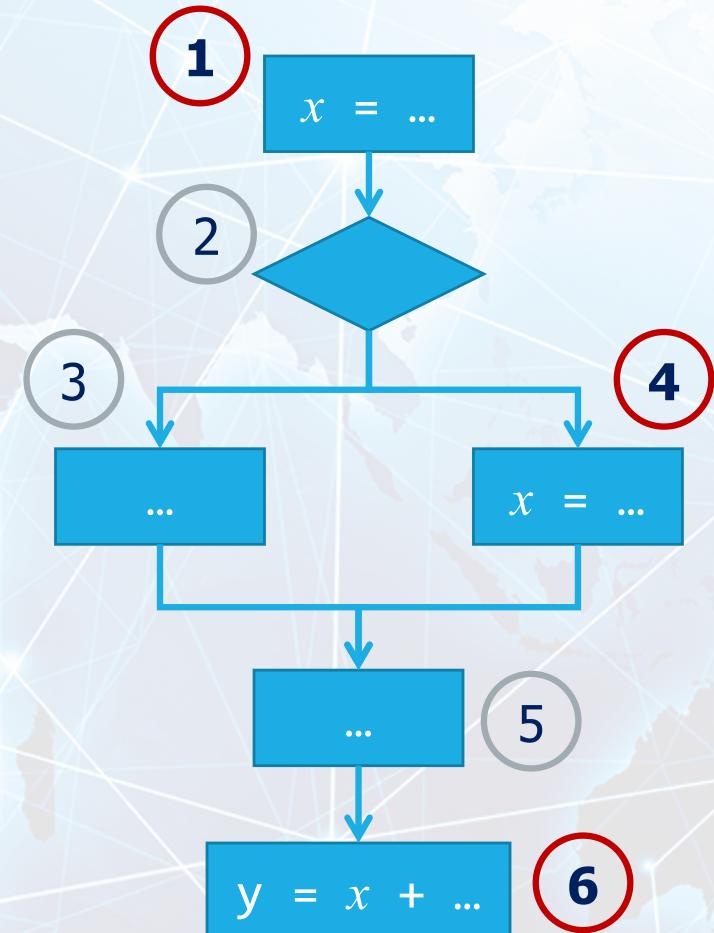
$\text{DEF}(4) = \{x, \dots\}$

$\text{USE}(6) = \{x, \dots\}$

(1, 2, 3, 5, 6) và (4, 5, 6) là các đường DC của biến  $x$ .

(1, 2, 4, 5, 6) không là đường DC của biến  $x$  vì nó được định nghĩa lại ở câu lệnh 4.

Các cặp DU là (1, 6) và (4, 6)





# Bài tập

Cho biết đâu là cặp DU của biến `heSo`

```
float tinhLuong(int loaiNhanVien, int soGioLam)
{
    float heSo = 1.0f; // (1)
    if (loaiNhanVien == 1)
    {
        heSo = 1.5f; // (2)
        if (soGioLam > 40)
            heSo = heSo + 0.2f; // (3)
    } else if (loaiNhanVien == 2)
        heSo = 1.2f; // (4)

    return 1200000 * soGioLam * heSo; // (5)
}
```





# Các mức độ phủ luồng dữ liệu



All-defs: all-defs của biến  $v$  là tập tất cả các đường definition-clear từ **mọi đỉnh định nghĩa** biến  $v$  đến **một đỉnh sử dụng** biến  $v$ .

All-uses: all-uses của biến  $v$  là tập các đường definition-clear từ **mọi đỉnh định nghĩa** biến  $v$  đến **mọi đỉnh sử dụng** biến  $v$  và các đỉnh tiếp sau sử dụng biến  $v$ .



# Các mức độ phủ luồng dữ liệu



All-p-uses/some-c-uses: all-p-uses/some-c-uses của biến  $v$  là tập các đường definition-clear từ **mọi định nghĩa** biến  $v$  đến **mọi định p-use** của biến  $v$ , nếu không tồn tại p-use như vậy thì tồn tại **một đường** đến ít nhất một c-use.

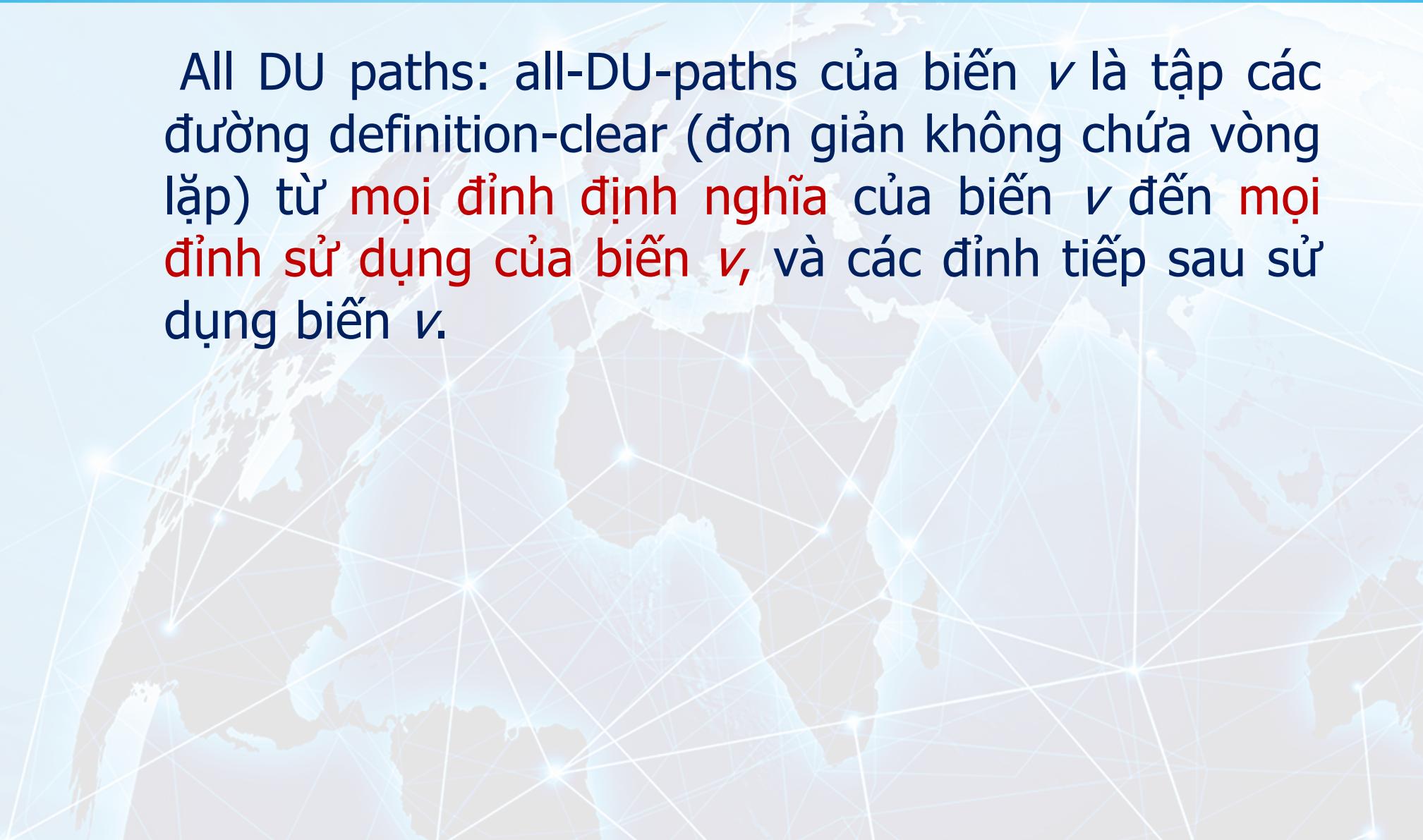
All-c-uses/some-p-uses: all-c-uses/some-p-uses của biến  $v$  là tập các đường definition-clear từ **mọi định nghĩa** của biến  $v$  đến **mọi định c-use** của biến  $v$ , nếu không tồn tại một c-use như vậy thì tồn tại **một đường** đi đến p-use.



# Các mức độ phủ luồng dữ liệu



All DU paths: all-DU-paths của biến  $v$  là tập các đường definition-clear (đơn giản không chứa vòng lặp) từ **mọi đỉnh định nghĩa** của biến  $v$  đến **mọi đỉnh sử dụng** của biến  $v$ , và các đỉnh tiếp sau sử dụng biến  $v$ .





# Các mức độ phủ luồng dữ liệu



Xác định all-defs, all-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, all DU paths của biến  $kq$

```
bool ktNguyenTo(int n)
{
    bool kq = false; // (1)
    if (n >= 2)
    {
        kq = true; // (2)
        for (int i = 2; i <= sqrt(n) && kq == true; i++) // (3)
            if (n % i == 0)
                kq = false; // (4)
    }

    return kq; // (5)
}
```



# Các mức độ phủ luồng dữ liệu



Ta có

$\text{DEF(kq, 1), DEF(kq, 2), DEF(kq, 4)}$

$\text{P-USE(kq, 3), C-USE(kq, 5)}$

<b>all-defs</b>	$(1) \rightarrow (5), (2) \rightarrow (3) \rightarrow (5),$ $(4) \rightarrow (3) \rightarrow (5)$
<b>all-uses</b>	$(1) \rightarrow (5), (2) \rightarrow (3), (4) \rightarrow (3) \rightarrow (5)$
<b>all-c-uses/ some-p-uses</b>	$(1) \rightarrow (5), (2) \rightarrow (3) \rightarrow (5),$ $(4) \rightarrow (3) \rightarrow (5)$
<b>all-p-uses/ some-c-uses</b>	$(2) \rightarrow (3), (4) \rightarrow (3)$
<b>all DU paths</b>	$(1) \rightarrow (5), (4) \rightarrow (3) \rightarrow (5), (2) \rightarrow (3)$ $\rightarrow (5)$



# Kiểm thử vòng lặp



Kiểm tra **tính hợp lệ** của cấu trúc vòng lặp.

Có 4 loại cấu trúc vòng lặp

Lặp đơn giản (simple loop)

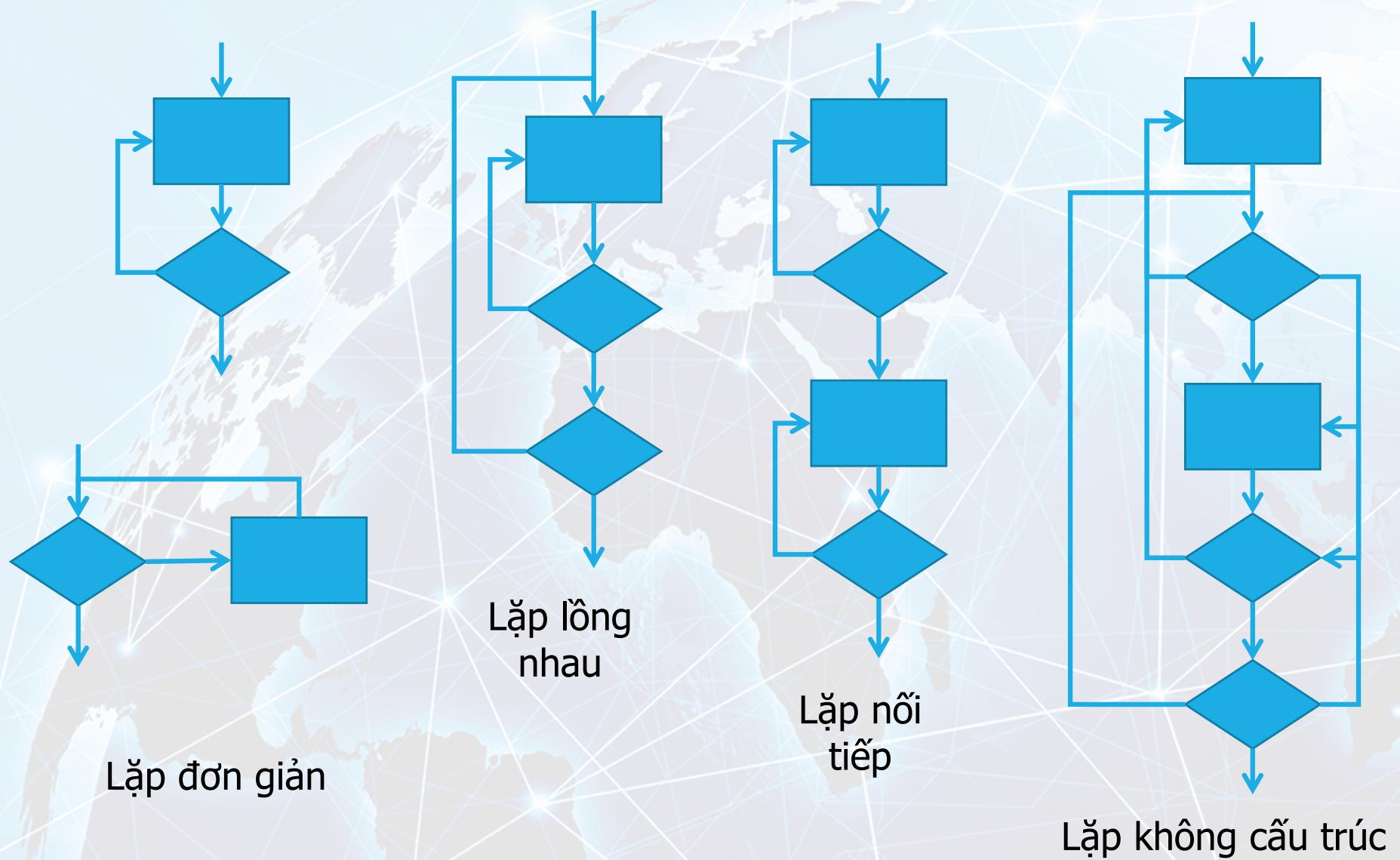
Lặp lồng nhau (nested loop)

Lặp nối tiếp (concatenated loop)

Lặp không cấu trúc (unstructured loop)



# Kiểm thử vòng lặp





# Kiểm thử vòng lặp

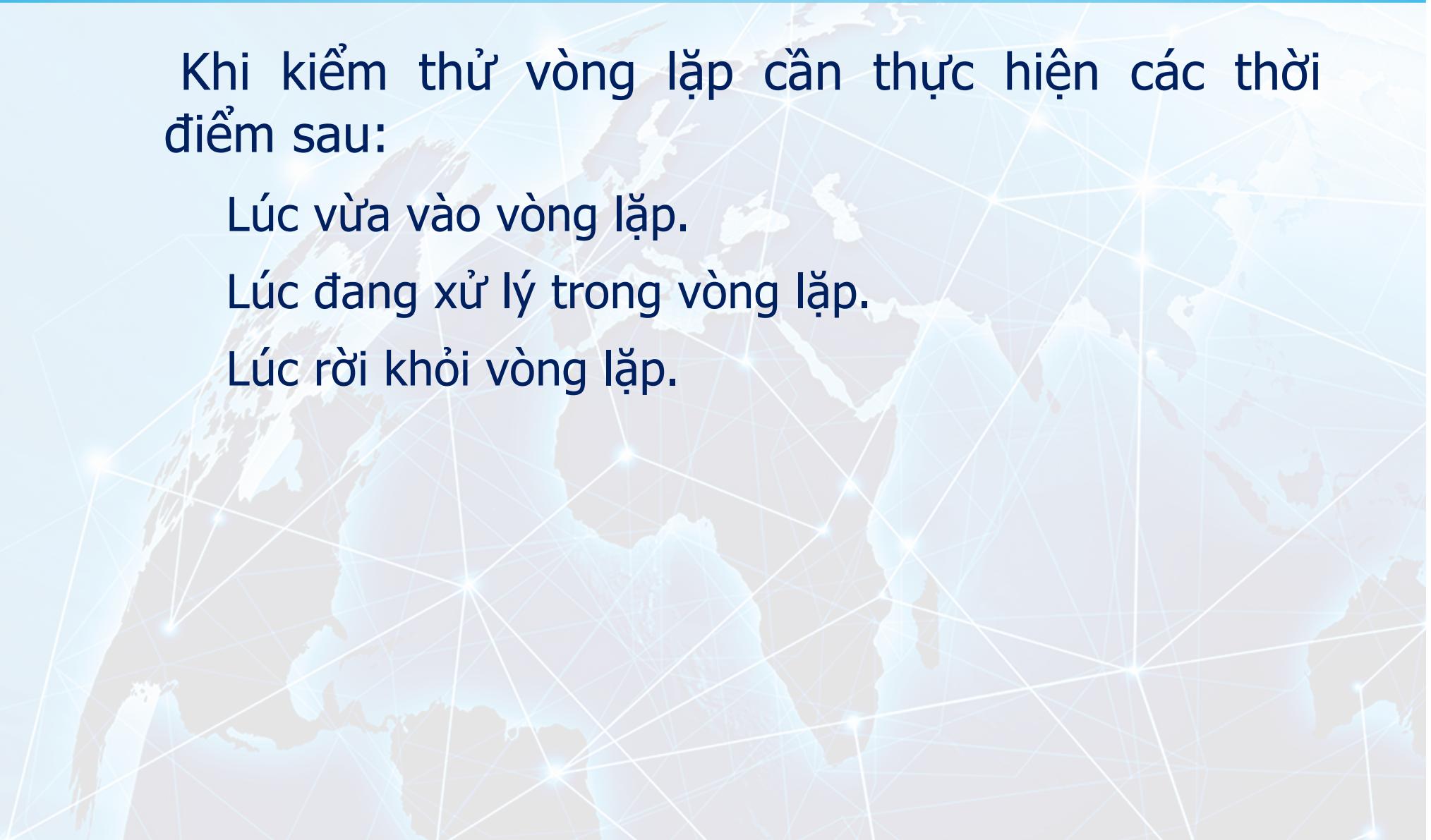


Khi kiểm thử vòng lặp cần thực hiện các thời điểm sau:

Lúc vừa vào vòng lặp.

Lúc đang xử lý trong vòng lặp.

Lúc rời khỏi vòng lặp.





# Lặp đơn giản



Tập các test case sau cần được thực hiện cho các vòng lặp đơn giản, trong đó n là số tối đa vòng lặp có thể thực thi.

**Không** thực hiện lần lặp nào.

Thực hiện **1** lần lặp.

Thực hiện **2** lần lặp.

Thực hiện **m** lần lặp, trong đó  $m < n$ .

Thực hiện  **$n - 1, n, n + 1$**  lần lặp.



# Lặp đơn giản



Ví dụ kiểm thử vòng lặp của đoạn chương trình kiểm tra số nguyên dương  $n$  có phải số nguyên tố không.

$n = 3 \rightarrow$  vòng lặp không thực thi lần nào

$n = 4 \rightarrow$  vòng lặp thực thi 1 lần

$n = 15 \rightarrow$  vòng lặp thực thi 2 lần ( $< \sqrt{n} \approx 3.9$ )

$n = 49 \rightarrow$  vòng lặp thực thi 6 lần ( $= \sqrt{n} = 7$ )

$n = 35 \rightarrow$  vòng lặp thực thi 4 lần ( $< \sqrt{n} \approx 5.9$ )



# Lặp lồng nhau



Bắt đầu test vòng lặp **trong cùng** (innermost) và các vòng lặp khác thiết lập một giá trị tối thiểu.

Dùng chiến lược kiểm thử vòng lặp đơn giản cho vòng lặp trong cùng và giữ các vòng lặp ngoài ở giá trị tối thiểu.

Thực hiện tương tự cho các vòng lặp ngoài, cho đến khi vòng lặp ngoài cùng (outermost) được test.

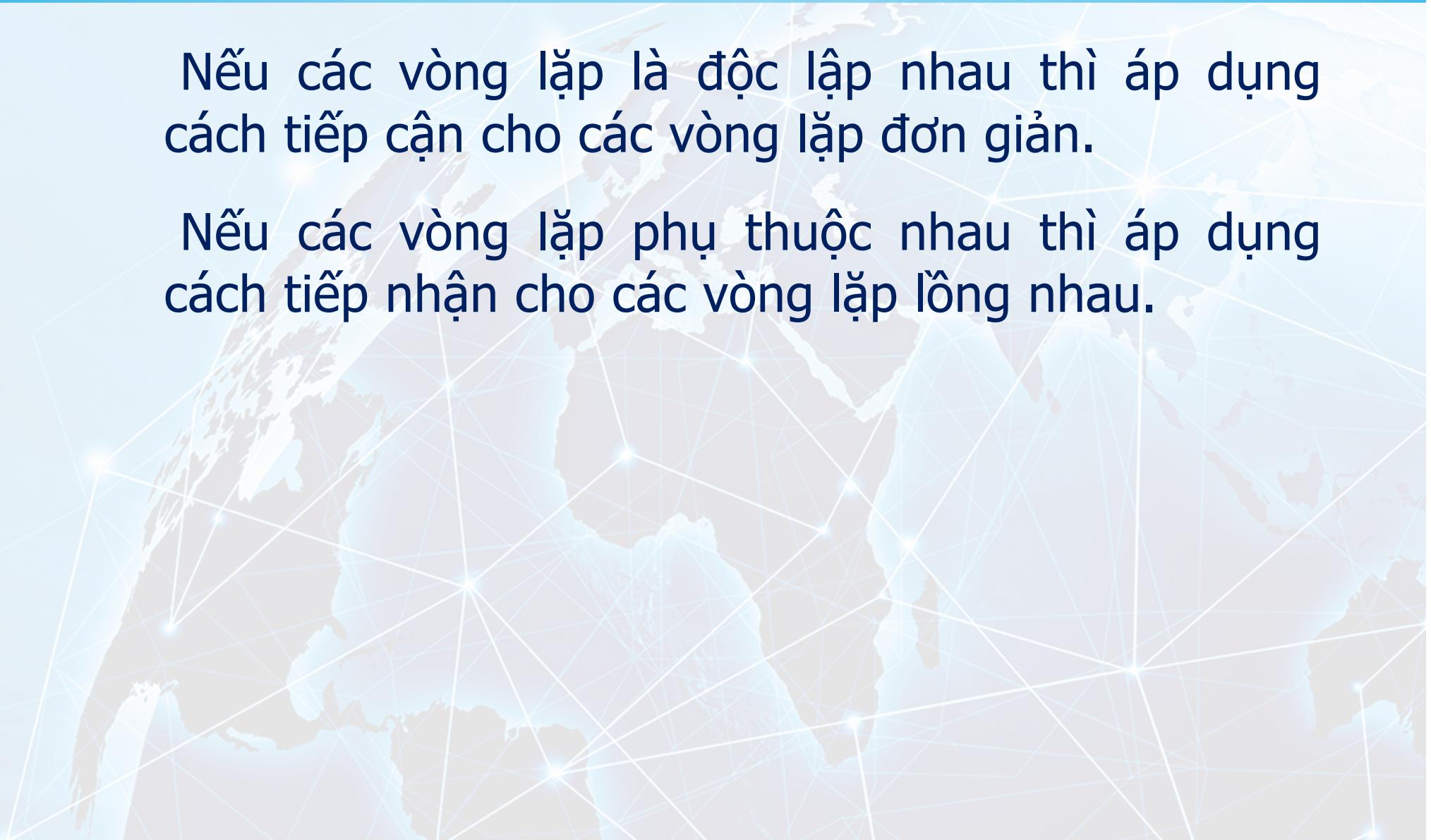


# Lặp nối tiếp



Nếu các vòng lặp là độc lập nhau thì áp dụng cách tiếp cận cho các vòng lặp đơn giản.

Nếu các vòng lặp phụ thuộc nhau thì áp dụng cách tiếp nhận cho các vòng lặp lồng nhau.





# Lặp không cấu trúc

Đối với trường hợp này nên yêu cầu thiết kế lại chương trình để đảm bảo tính cấu trúc của chương trình.



