

**UNIVERSITE DE CORSE**  
**2024-2025**

**Licence ST 3ème année**  
**Option INFORMATIQUE**

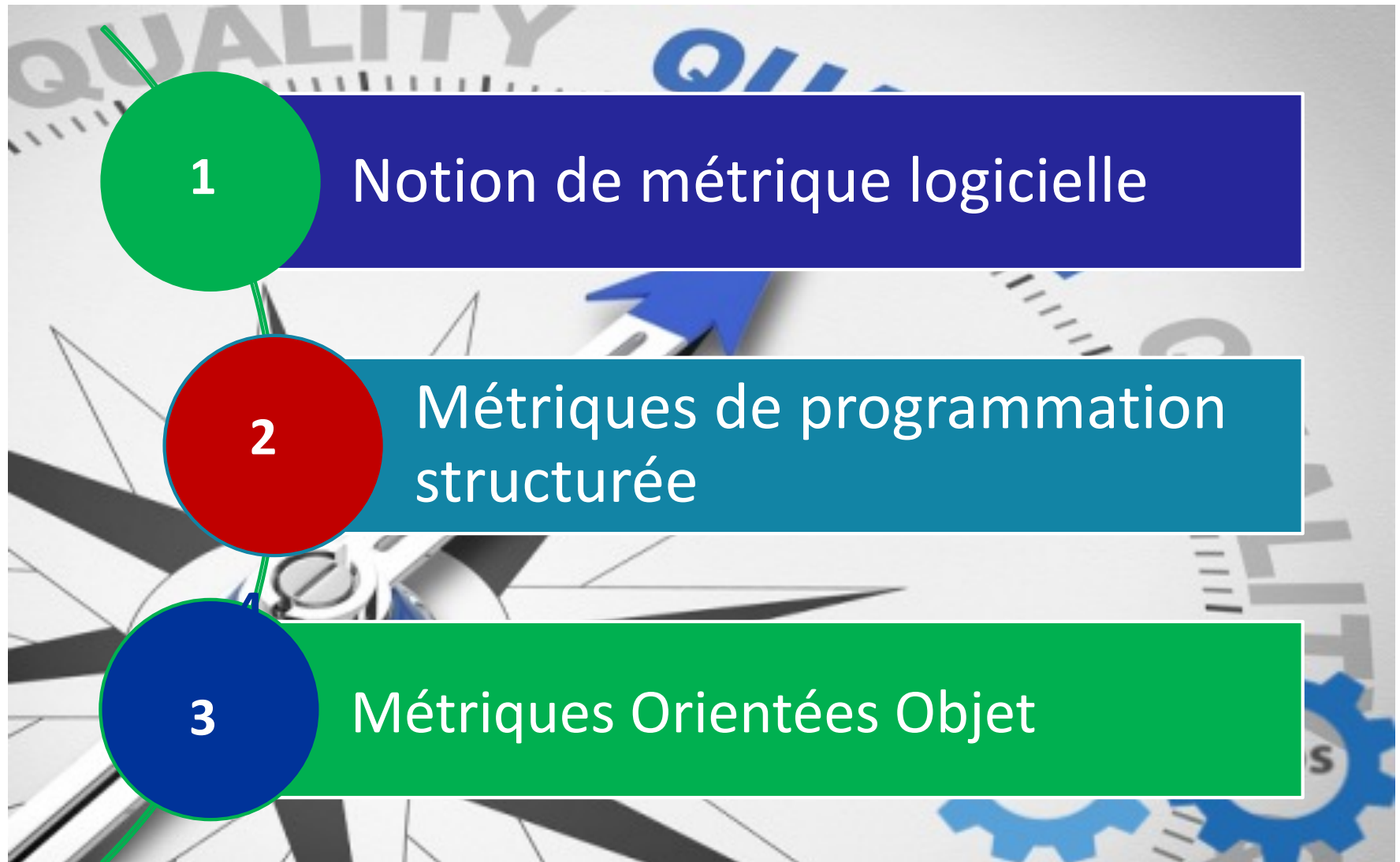
# UE Qualité Logicielle et Tests

## CH2.2 –Métriques de programmation structurée



Evelyne VITTORI  
[vittori@univ-corse.fr](mailto:vittori@univ-corse.fr)

# CH2- Métriques logicielles



# Les métriques de programmation structurée

Métriques « traditionnelles » utilisées depuis les années 1970

- Nombre de lignes de code (LOC ou KLOC)
- Complexité cyclomatique de McCabe (CC)
- Métriques de Halstead (ou Software Science)
- Index de maintenabilité



# Nombre de lignes de code



# LOC (Line Of Code)

- La métrique la plus simple!!
- LOC= nombre de ligne de code sans comptabiliser les lignes blanches et les commentaires. (KLOC si exprimée en kilo)
- En plus précis:
  - LOCphy= nombre de lignes physiques d'un fichier
  - LOCfonction= Nombre moyen de ligne de code par fonction
  - LOCbl= nombre de lignes vides
- LOCCom= nombre de lignes de commentaires
- Pourcentage de Commentaires =  $\text{LOCCom} / \text{LOC}$

# Recommandations (exemple)

1. LOCFunction doit être compris entre 4 et 40 lignes maximum
2. LOCPhy doit être compris entre 40 et 400 lignes
3. Pourcentage de commentaires devrait être compris entre 30% et 75%:
  - Si moins de 30% : code trivial ou manque d'explications
  - Si plus de 75%: le fichier est un document pas un programme!

Un principe élémentaire: Plus LOC est élevé et plus l'effort de maintenance sera important



# Complexité cyclomatique de McCabe

- Objectifs
- Graphe de contrôle
- Modes de calcul
- Interprétations
- Exercices d'application



# MCC: Complexité Cyclomatique de McCabe

- Evaluation de la **complexité statique** d'un programme



Analyse d'un code  
source sans exécution

- Indicateur **indépendant** du langage de programmation
- Objectifs
  - Estimation de la maintenabilité du programme (testabilité et compréhensibilité)
  - Evaluation du degré de «confiance» que l'on peut avoir dans un programme



# Comment calculer la complexité cyclomatique?

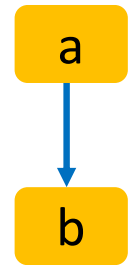
1. Définir le **graphe de contrôle** du programme
2. Evaluer le nombre de chemins indépendants possibles dans ce graphe
  - Deux modes de calculs
    - Comptage des arcs et des nœuds
    - Comptage des prédicats

# Graphe de contrôle d'un programme

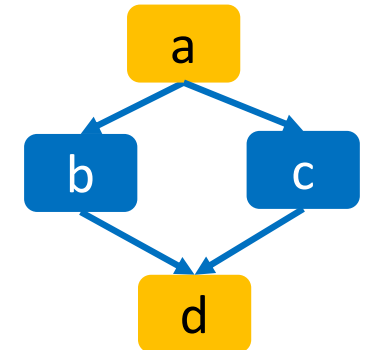
Soit un programme  $S$ , son **graphe de contrôle**, noté  $[S]$ , est un graphe orienté :

- Sommets (nœuds): instructions
- Arcs: conditions (tests et boucles)
- chaque graphe comporte un nœud «entrée» et un nœud « sortie»

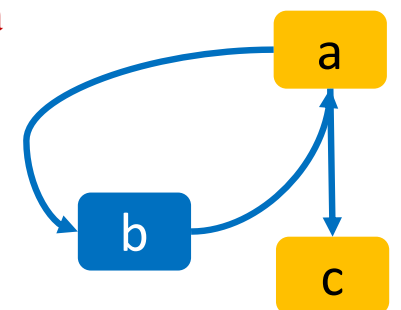
**Instructions  
séquentielles**



**Schéma  
alternatif**



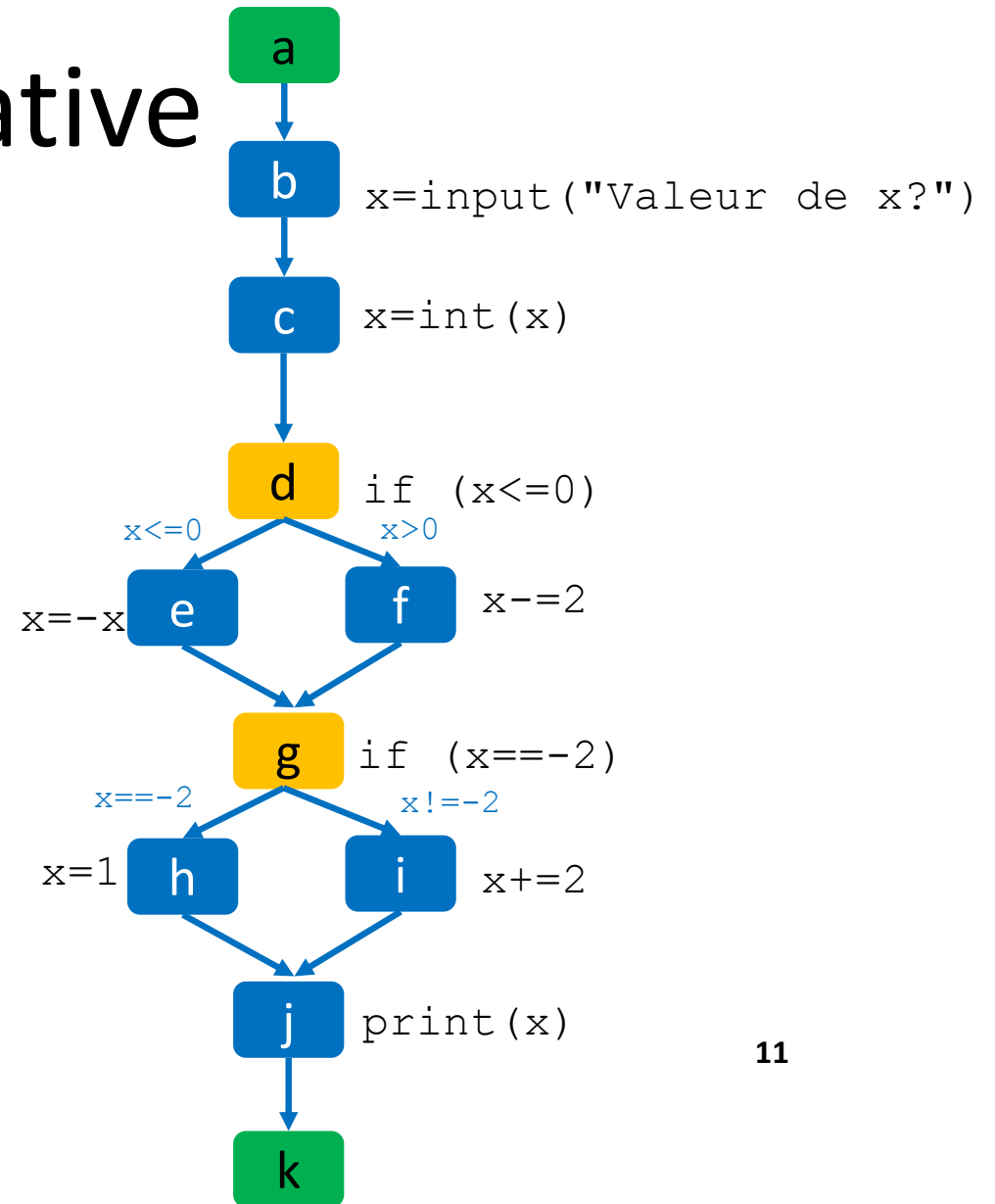
**Schéma  
itératif**



# Exemple 1

## Graphe avec alternative

```
x=input("Valeur de x?")
x=int(x)
if (x<=0):
    x=-x
else:
    x-=2;
if (x== -2):
    x=1
else:
    x+=2
print(x)
```



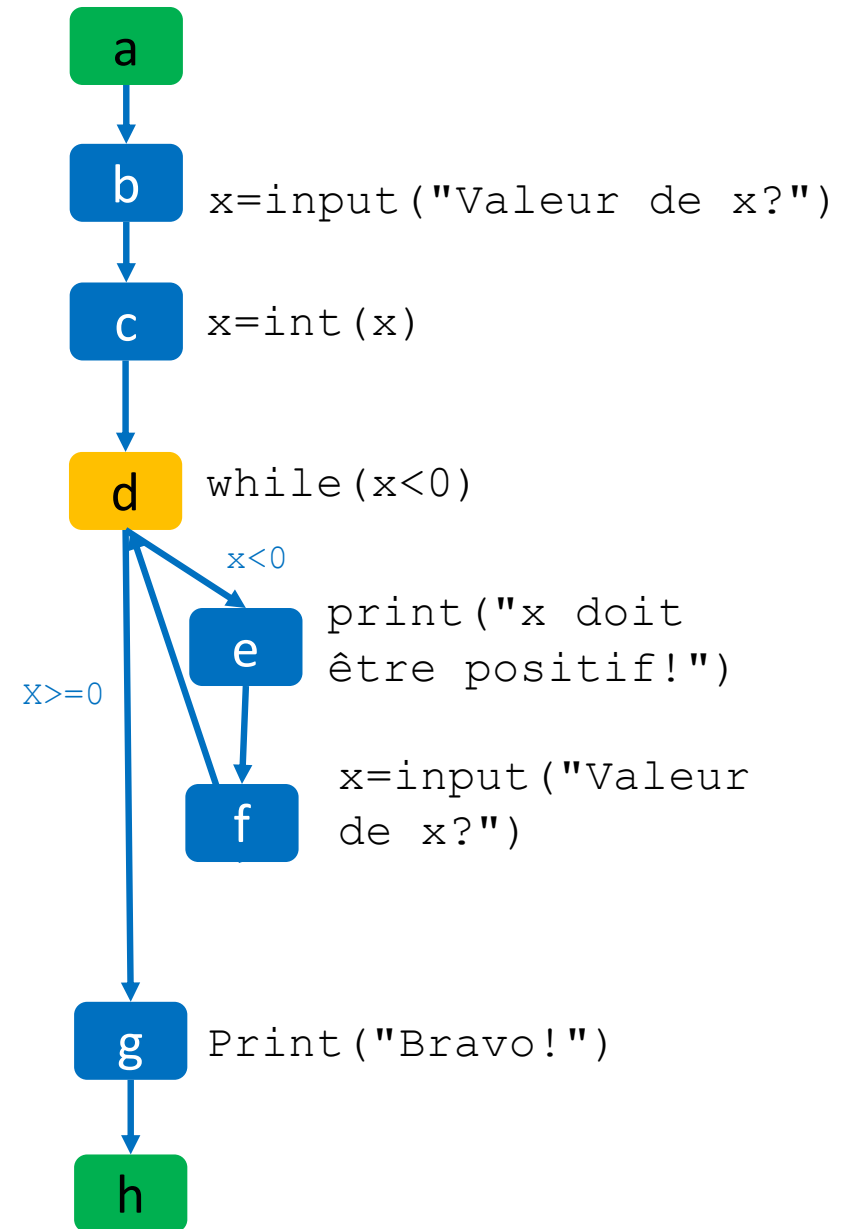
# Exemple 2

## Graphe avec itération

```
x=input("Valeur de x?")
x=int(x)
while (x<0):
    print("x doit
être positif!")
    x=input("Valeur
de x?")
print("Bravo!")
```

*Remarque:*

Les boucles for sont représentées comme des boucles while particulières



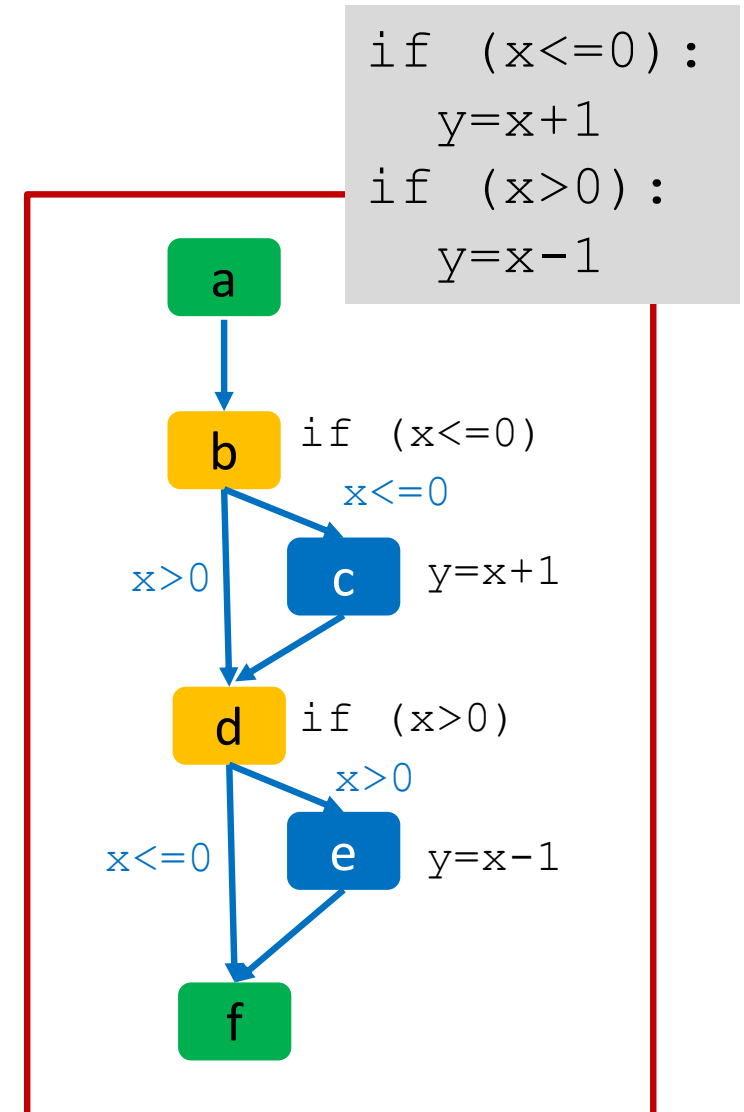
# Chemin de contrôle

- Un chemin exécutable ou **chemin de contrôle** correspond à une exécution possible du programme

Ex= [a,b,c,d,f] , [a,b,d,e,f]

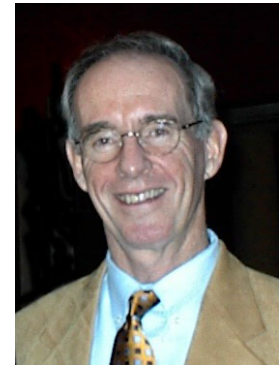
- Un chemin est dit « non exécutable » s'il ne correspond pas à une exécution possible

Ex= [a,b,c,d,e,f] , [a,b,d,f]





# Complexité Cyclomatique de McCabe



- Définie par Thomas McCabe en 1976
- CC = Complexité cyclomatique (ou **nombre cyclomatique**) = nombre de chemins indépendants dans un graphe de contrôle
- Principe:
  - Si ce nombre est trop grand cela va compliquer les tests et la compréhensibilité du code et rendre sa maintenance difficile.

# Calcul de la Complexité Cyclomatique

CC (ou  $V(G)$ ) est calculé à partir du graphe de contrôle

$$CC = E - N + 2P \text{ avec}$$

- $E$  = nombre d'arcs
- $N$  = nombre de nœuds
- $P$  = nombre de sous-graphiques déconnectés (nombre de composantes connexes)

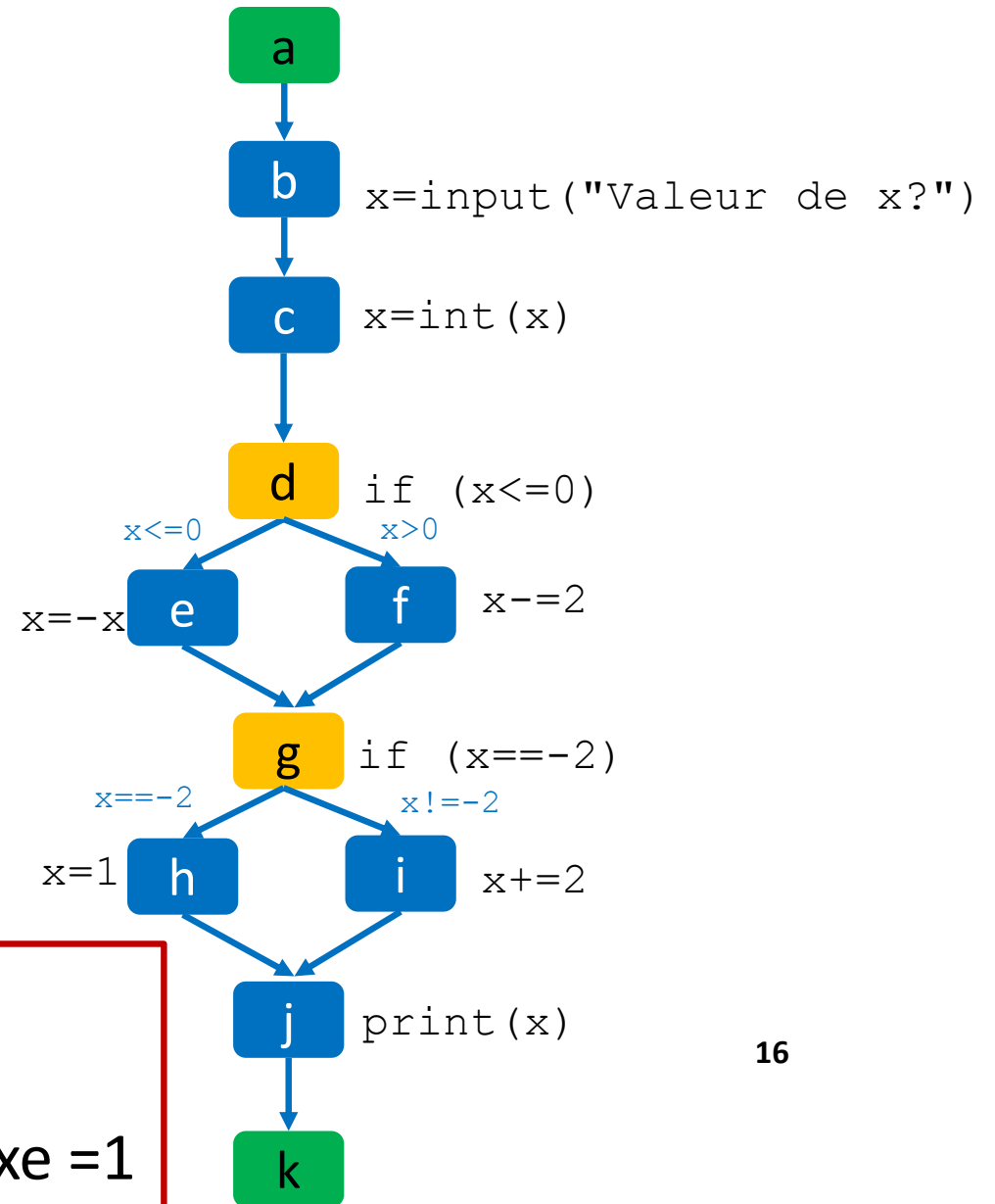
*Toujours égal à 1 dans les exemples étudiés*

*Souvent noté  $V(G)$*

# Calcul de la complexité cyclomatique

## Exemple 1

```
x=input("Valeur de x?")
x=int(x)
if (x<=0):
    x=-x
else:
    x-=2;
if (x== -2):
    x=1
else:
    x+=2
print(x)
```



$E = \text{nombre d'arcs} = 12$

$N = \text{nombre de nœuds} = 11$

$P = \text{nombre de composantes connexe} = 1$

$CC = E - N + 2P = 12 - 11 + 2 = 3$

# Calcul simplifié de la complexité cyclomatique

Calcul simplifié

$$CC = \pi + 1$$

Avec  $\pi$  = nombre de prédicats élémentaires (sans connecteur logique)

- Une instruction IF > compte 1 par prédicat
- Une boucle FOR ou WHILE > compte 1 par prédicat
- Une instruction CASE traitant N cas > N-1 prédicats

# Calcul simplifié de la complexité cyclomatique

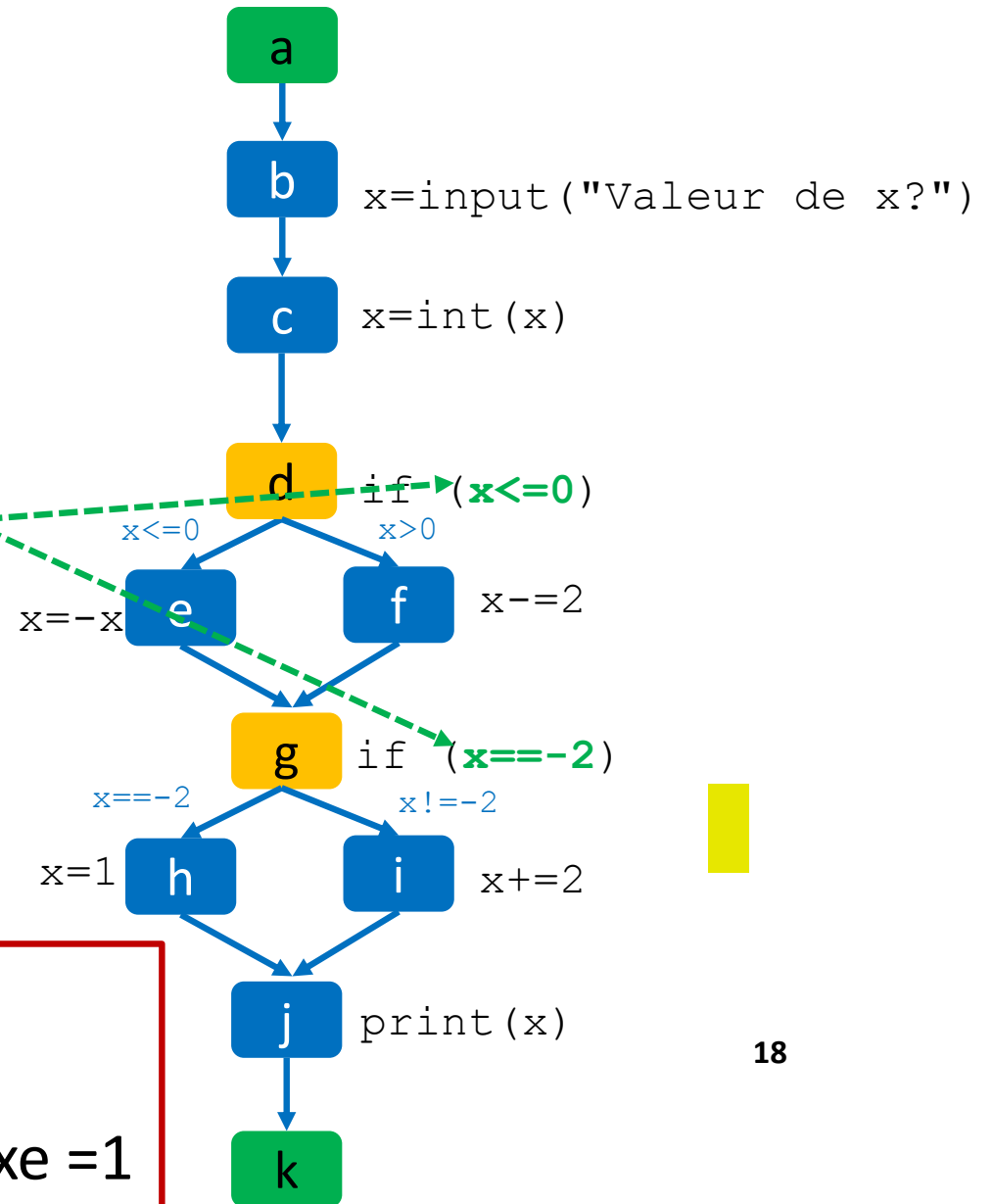
## Exemple 1 (suite)

```
x=input("Valeur de x?")
x=int(x)
if (x<=0):
    x=-x
else:
    x-=2;
if (x===-2):
    x=1
else:
    x+=2
print(x)
```

*Calcul simplifié*

$\pi$  = nombre de  
prédicats = 2

**CC** =  $\pi + 1 = 3$



**E** = nombre d'arcs = 12

**N** = nombre de nœuds = 11

**P** = nombre de composantes connexe = 1

**CC** = **E** - **N** + 2**P** = 12 - 11 + 2 = 3



# Calcul de la complexité cyclomatique

## Exemple 2

```
x=input("Valeur de x?")
x=int(x)
while (x<0):
    print("x doit
être positif!")
    x=input("Valeur
de x?")
print("Bravo!")
```

*Calcul simplifié*

$\pi$  = nombre de  
prédicats = 1

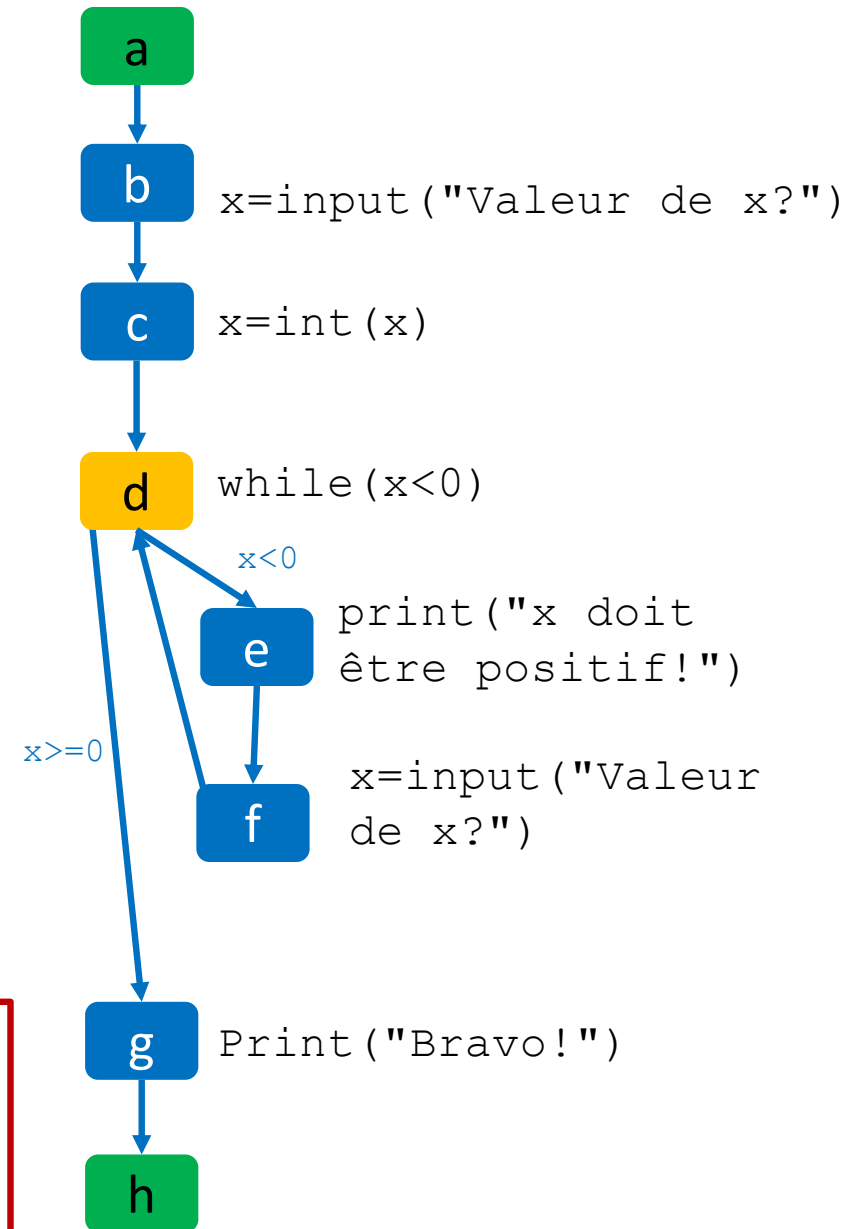
**CC** =  $\pi + 1 = 2$

E = nombre d'arcs = 8

N = nombre de nœuds = 8

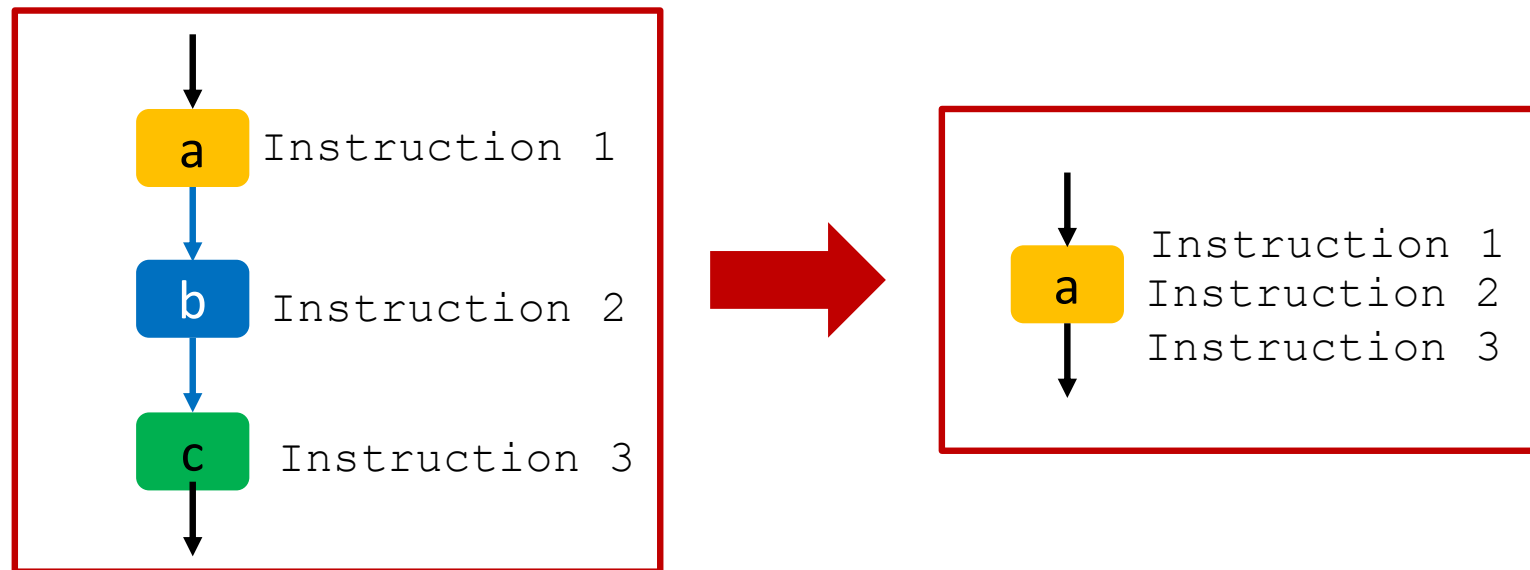
P = nombre de composantes connexe = 1

**CC** = **E** - **N** + **2P** = 8 - 8 + 2 = 2



# Réduction d'un graphe de contrôle

- Un graphe de contrôle peut être réduit en regroupant les nœuds représentant des instructions simples successives



Le nombre cyclomatique est conservé

Il n'y a pas de perte d'informations sur le flot de contrôle

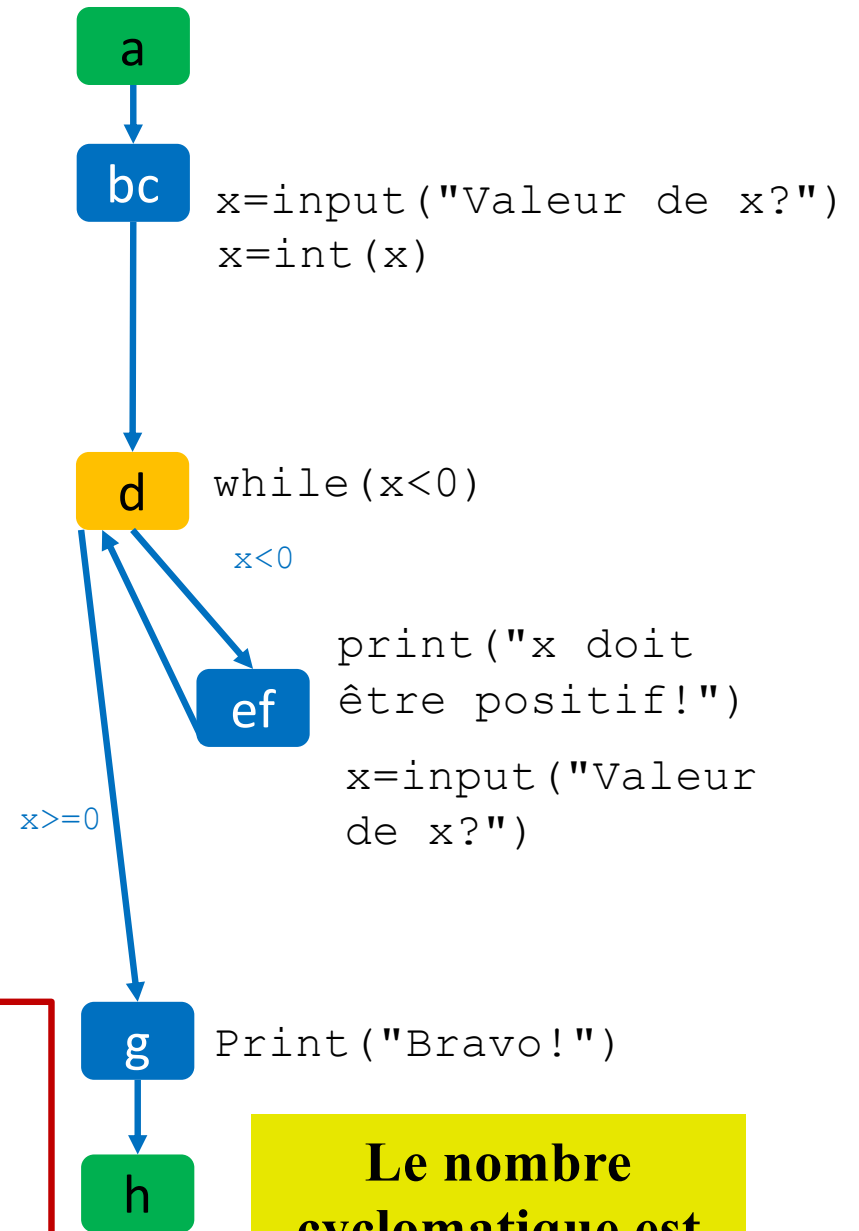
# Calcul de la complexité cyclomatique sur graphe réduit

## Exemple 2 (suite)

```
x=input("Valeur de x?")
x=int(x)
while (x<0):
    print("x doit être positif!")
    x=input("Valeur de x?")
print("Bravo!")
```

*Calcul simplifié*  
 $\pi$  = nombre de  
prédicats = 1  
 $CC = \pi + 1 = 2$

E = nombre d'arcs = 6  
N = nombre de nœuds = 6  
P = nombre de composantes connexes = 1  
 $CC = E - N + 2P = 6 - 6 + 2 = 2$



**Le nombre  
cyclomatique est  
conservé**

# Exercice 1



- Réécrivez le programme suivant en utilisant uniquement des prédicats élémentaires
- Définissez le graphe de contrôle
- Calculez la complexité cyclomatique de deux manières différentes

```
if (x==1 and y==0) :  
    if (z==1) :  
        print("cas1")  
    else:  
        print("cas 2")  
else:  
    print("cas 3")
```

# Exercice 2 – Graphes de contrôle



- Définissez les graphes de contrôle des fonctions python suivantes et calculez leur complexité cyclomatique

```
def calcul(p: int, q:int)->int:
    while (p != q):
        if (p > q):
            p = p - q
        else:
            q = q - p
    return p
```

*Remarque:*

Les boucles for sont représentées comme des boucles while particulières

```
def somme1(a:list,inf:int,sup:int)->int:
    sum = 0
    for i in range(inf,sup) :
        sum = sum + a[i]
    return sum
```



# Complexité cyclomatique et Tests

- Un jeu de test correspond à un ensemble de chemins dans un graphe de contrôle
- Le nombre cyclomatique est un indicateur du nombre de cas de tests « suffisant » pour une couverture pertinente du code.
- $CC$  = nombre de cas
  - maximal pour atteindre une couverture complète du code (tous les nœuds)
  - minimal pour une couverture complète des chemins (*path coverage*) (tous les chemins possibles)

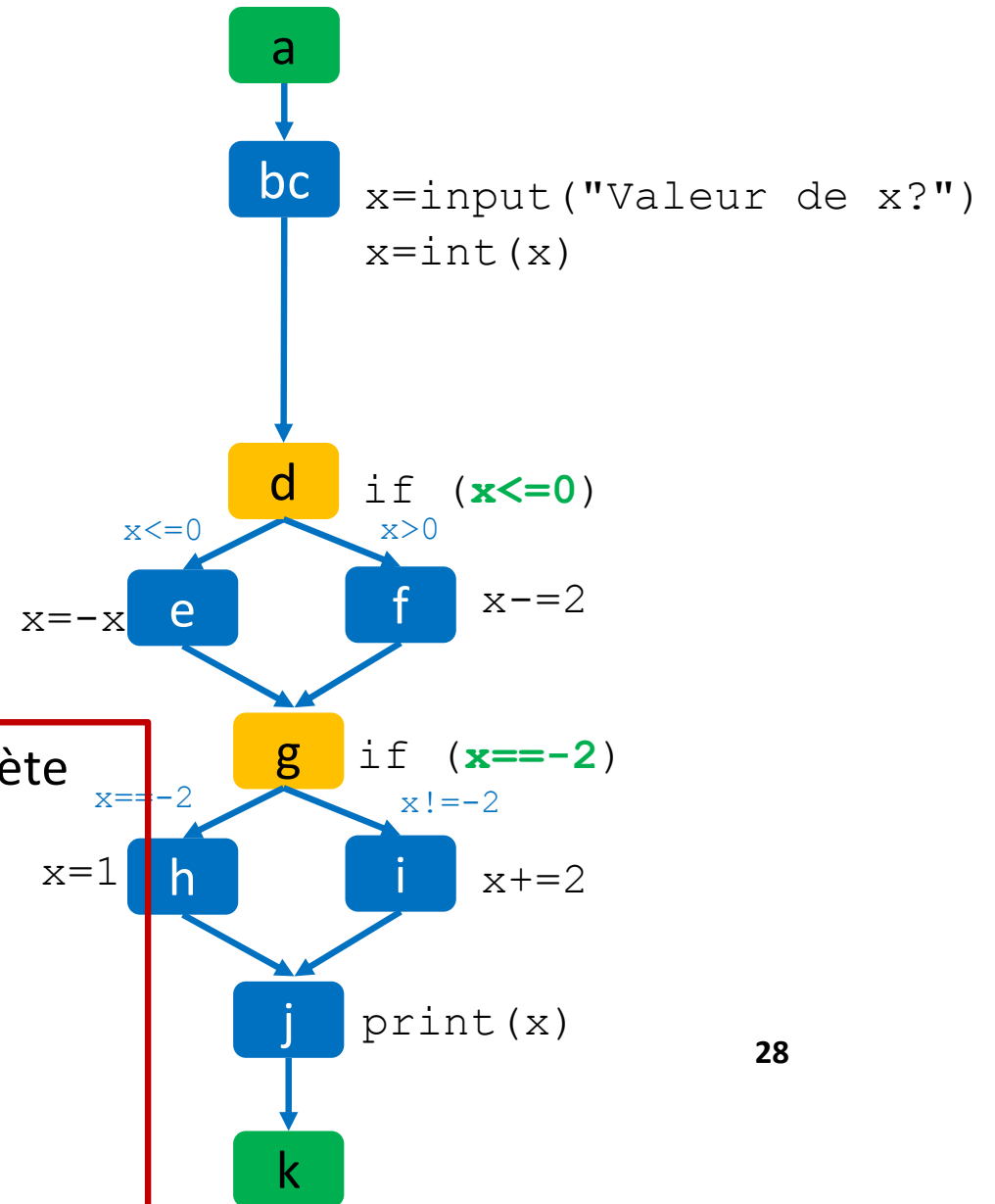
# Détermination d'un jeu de tests

## Exemple 1 (suite)

```
x=input("Valeur de x?")
x=int(x)
if (x<=0):
    x=-x
else:
    x-=2;
if (x===-2):
    x=1
else:
    x+=2
print(x)
```

*Nombre cyclomatique*  
**CC= 3**

- **maximum pour couvrir le code**
- **minimum pour couvrir tous les chemins**



2 cas suffisants pour une couverture complète des nœuds

[abc, d, f, g, i, j, k]

[abc, d, e, g, h, j, k]

4 cas pour couvrir tous les chemins

[abc, d, f, g, i, j, k]

[abc, d, e, g, i, j, k]

[abc, d, f, g, h, j, k] ] cas impossible

[abc, d, e, g, h, j, k] cas impossible

Nous y reviendrons .. CH4- Tests

# Exercice 3

1. Définissez le graphe de contrôle de la méthode operation et calculez sa complexité cyclomatique.
2. Les testeurs ont identifié 3 cas de tests :
  - solde insuffisant, montant nul, un crédit.

Cela vous semble-t-il pertinent?

```
public class Banque {  
    private Double solde;  
    public void operation(String type, double montant) {  
        if(montant != 0) {  
            if(type.equals("+")) {  
                solde += montant;  
            } else  
            if(type.equals("-")) {  
                if(montant > solde) {  
                    System.err.println("Erreur : Solde insuffisant !");  
                }  
                else {  
                    solde -= montant;  
                }  
            }  
            else {  
                System.err.println("Erreur : Type d'opération invalide.");  
            }  
        } else {  
            System.err.println("Erreur: Montant nul");  
        }  
    }  
}
```


# Interprétation de la Complexité Cyclomatique de McCabe

- Principe:
  - Si ce nombre est trop grand cela va compliquer les tests et la compréhensibilité du code et rendre sa maintenance difficile.

Complexité cyclomatique	évaluation des risques
1-10	programme simple, risque minimal
11-20	risque modéré
21-50	programme complexe, risque élevé
Plus de 50	programme non testable, risque très élevé

La complexité cyclomatique d'un programme est un excellent indicateur de sa testabilité.

# Quand calculer la complexité cyclomatique?

- Phase de développement
  - Si CC dépasse 10  il faut diviser le programme
- Conception des tests
  - CC donne une indication sur le nombre de cas de tests minimaux
- Maintenance
  - Mesurer CC avant et après une modification permet de minimiser le risque de modifications supplémentaires
- Refactoring (Reingénierie)
  - Evaluation du risque de « refactoring » d'une partie du code.





# Métriques de Halstead

# Métriques de Halstead



- Définies en 1977 par Maurice Halstead.
- Mesures quantitatives de complexité basées sur l'analyse syntaxique du code source.
- Permet d'évaluer le **risque d'erreur** et d'estimer le temps de développement
- Un programme est considéré comme un texte : une suite finie de **jetons**:
  - **Opérateur**= symbole ou mot clé qui spécifie une action
  - **Opérande**= symbole qui représente une donnée (constante, variable, littéral, ...) ou un type

# Métriques élémentaires de Halstead

- $n1$  = nombre d'opérateurs distincts
- $n2$  = nombre d'opérandes distincts
- $N1$  = nombre total d'occurrences des opérateurs
- $N2$  = nombre total d'occurrences des opérandes

Mots clés,  
symboles  
opérateurs,  
parenthèses ...

Exemple  $x = x + 2 + y;$

$n1 = 3$	(= + ;)
$n2 = 3$	(x y 2)
$N1 = 4$	(=: 1, +: 2, ;: 1)
$N2 = 4$	(x :2, 2:1, y:1)

**Toutes les métriques  
de Halstead sont  
dérivées des  
nombres  $n1$ ,  $n2$ ,  $N1$ ,  
 $N2$**

# Métriques de base de Halstead

- **Vocabulaire** du programme  $n = n_1 + n_2$
- **Taille** du programme  $N = N_1 + N_2$
- **Volume** du programme (estimation du nombre de bits nécessaires)  $V = N \log_2(n)$ 
  - Non sensible à la disposition du code
  - Minimum 20 (1 ligne) et maximum à 1000.
- Niveau de **difficulté**  $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$ 
  - propension d'erreurs du programme
  - plus le nombre d'opérandes augmente et plus le risque d'erreur augmente
- Niveau de programme  $L = \frac{1}{D}$  (inverse de la difficulté)

# Métriques d'estimation de Halstead

- Effort d'implémentation  $E = V \times D$
- Temps d'implémentation  $T = \frac{E}{18}$ 
  - Approximation du temps de d'écriture du programme exprimé en secondes.
- Taille estimée du programme  $Le = n1 \log_2(n1) + n2 \log_2(n2)$
- Nombre de bugs estimés  $B = \frac{E^{2/3}}{S}$

avec  $S = \text{habilité du développeur} = 3000$

Valeurs recommandées

- Volume d'une fonction :  $20 < V < 1000$

# Exemple de calcul

```
int x,y,z;
z=0;x=10;y=5;
while (x>0) {
    z=z+y;
    x=x-1; }
System.out.println(z);
```

Métriques		Valeur	Formule
Opérateurs	n1	10	
Opérandes	n2	7	
Occ.Opérateurs	N1	21	
Occ.Opérandes	N2	18	
Vocabulaire	n	17	$n1+n2$
Taille	N	39	$N1+N2$
Volume	V	159,41	$N*\log_2(n1+n2)$
Difficulté	D	12,86	$(n1/2)*(N2/n2)$
Taille estimée	Le	52,87	$n1 \log_2(n1)+n2 \log_2(n2)$
Effort	E	2049,57	$V \times D$
Temps	T (secondes)	113,87	$E/18$
Nombre de bugs	B	0,05	$E^{2/3} / 3000$

Opérateurs		Opérandes	
=	5	x	5
while	1	y	3
println	1	z	5
-	1	0	2
+	1	10	1
>	1	5	1
{}	1	1	1
int	1		
,	2		
;	7		

# Exercice 4

fichier Excel à récupérer sur l'ENT

- Calculez les mesures de Halstead pour la méthode suivante:

```
void sort(int[] px, int n) {  
    int i, j, temp;  
    for(i=2; i<n; i++) {  
        for(j=1; j<i; j++) {  
            if(px[i]<px[j]) {  
                temp=px[i];  
                px[i]=px[j];  
                px[j]=temp;  
            }  
        }  
    }  
}
```





# Index de maintenabilité





# Index de maintenabilité

- Calculé à partir des métriques LOC, de la complexité cyclomatique et du Volume de Halstead.
- Indique quand il devient moins cher et moins risqué de réécrire le code plutôt que de le corriger.
- 3 variantes:
  - **MIwoc** : Maintainability Index without comments
  - **MIcw** : Maintainability Index comment weight
  - **MI** : Maintainability Index

# Calcul de l'Index de Maintainabilité MI

- **MI** = MIwoc + MIcw
- **Maintenabilité sans les commentaires**

$$\text{MIwoc} = 171 - 5.2 * \ln(\text{aveV}) - 0.23 * \text{aveCC} - 16.2 * \ln(\text{aveLOC})$$

- **aveV** = valeur moyenne du volume V d'Halstead par module
- **aveCC** = valeur moyenne de la complexité cyclomatique CC par module
- **aveLOC** = nombre moyen de lignes de code par module (LocPhy)

- **Maintenabilité des commentaires**

$$\text{MIcw} = 50 * \sin(\sqrt{2.4 * \text{perCM}})$$

- **perCM** = pourcentage des commentaires dans le code ( $\text{cLoc}/\text{loc}$ )

# Interprétation de MI

- $MI \geq 85$  : bonne maintenabilité
- $65 < MI < 85$  : maintenabilité modérée
- $MI \leq 65$  : maintenabilité difficile



**Il est préférable de  
réécrire le code**

# Exemple de calcul

```
int x,y,z;  
z=0;x=10;y=5;  
while (x>0) {  
    z=z+y;  
    x=x-1; }  
System.out.println(z);
```

METRIQUE		Valeur
Volume (Halstead)	V	159,41
Nombre de lignes	LOC	6,00
Complexité cyclomatique	CC	2,00
Maintenabilité sans Com	MIwoc	142,38
Maintenabilité des Com	MIcw	0
Pourcentage Commentaires	perCM	0
Index de Maintenabilité	MI	142,38

# Exercice 4 (suite)

- Calculez l'indice de maintenabilité de la méthode suivante:

```
void sort(int[] px, int n) {  
    int i, j, temp;  
    for(i=2; i<n; i++) {  
        for(j=1; j<i; j++) {  
            if(px[i]<px[j]) {  
                temp=px[i];  
                px[i]=px[j];  
                px[j]=temp;  
            }  
        }  
    }  
}
```