U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

II- Terminaison et calculabilité

I- Le problème de la terminaison
II- Preuve de terminaison
III-Cas de problèmes non calculables
IV-Calculabilité partielle

Le premier chapitre nous a révélé qu'il existe des problèmes qui ne **relèvent pas du domaine du calcul**.

Nous allons étudier des **problèmes** :

- apparemment **simples**,
- qu'**aucun algorithme** ne peut résoudre.

Le plus fameux d'entre eux est le problème de la terminaison d'un algorithme ou **problème de l'arrêt**.

I-Problème de la terminaison

Une situation souvent rencontrée dans l'exécution des algorithmes est que certains algorithmes ne **se terminent pas**.

Pour employer l'expression consacrée, on dit que **«ça boucle».**

Il serait donc très important, de pouvoir:

-**détecter** dans les algorithmes les **boucles sans fin**

-**avant** les exécuter.

Définition

Un algorithme se **termine**, si son exécution :

- s'arrête **toujours**
- **quelque soit** la nature des données en entrée

On peut résumer le **problème de l'arrêt** par l'exemple de la fonction **collatz** définie sur $\mathbb{N}^*$.

```
fonction collatz(n)
  debut
     // spécification: n > 0
      si (n=1) retourner 1;
      si (n%2 = 0 )     collatz(n/2);
      sinon                collatz(3*n+1);
  fin
```

Le calcul de cette fonction **se termine-t-il** ?

n=3

10 5 16 8 4 2 1

n=7

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Cas de la suite de Syracuse

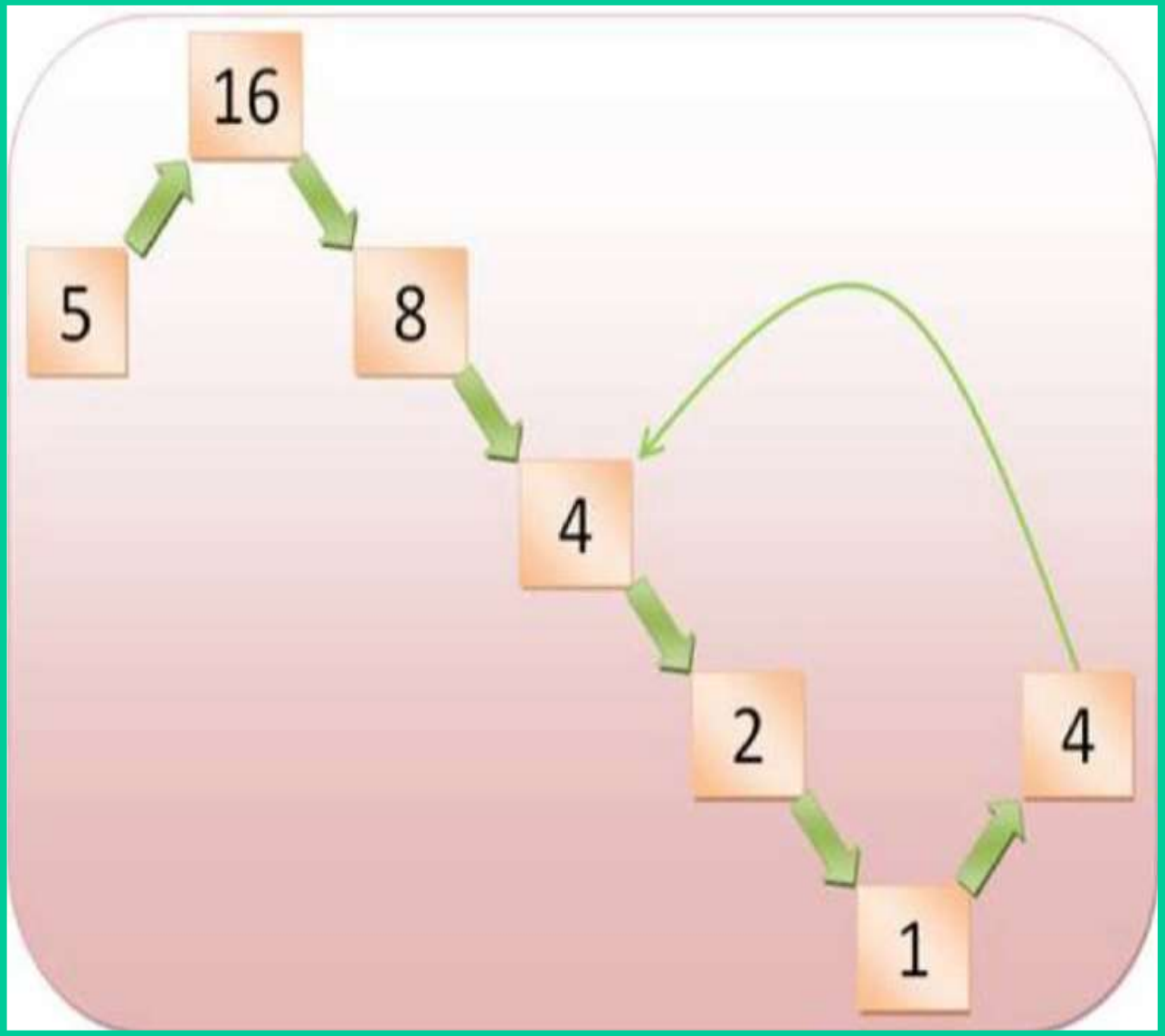
En appliquant la fonction de Collatz pour

n=**10**,

on construit la suite des nombres:

[**5, 16, 8, 4, 2, 1, 4, 2**]

qu'on appelle la **suite de Syracuse** du nombre **10**.


16
5
8
4
2
4
1

La conjecture de Collatz

A priori, il serait possible que :

- la **suite de Syracuse** de certaines valeurs de départ **n**

- n'atteignerait **jamais** la valeur **1**.

Soit elle aboutisse à un cycle différent du **cycle trivial**.

Soit elle **diverge** vers l'infini.

Or, on n'a **jamais trouvé** d'exemple de suite obtenue qui n'aboutisse pas à 1.

La conjecture de Collatz est l'**hypothèse** selon laquelle la suite de Syracuse de n'importe quel entier n>0 atteint **1**.

Certains avancent même que le problème serait **indécidable**.

Un résultat paradoxal

Supposons que l'on puisse écrire un programme appelé **testArrêt** qui :

- prend en paramètre une fonction
- retourne **vrai** si cette fonction s'arrête quelque soit la donnée d'entrée.

Que dire de la fonction suivante ?:

```
function F(x,y)
{
if (testArret (F(x,y)))
                                F(x,y);
}
```

Le calcul de **F(x,y)** s'arrête si et seulement si **il ne s'arrête pas !**

Cas de l'équation : $x^n + y^n n = z^n$

Soit le problème formulé comme suit:

« Etant donné un entier **n**, afficher **"hello world"** si l'équation suivante:

$$\mathbf{x^n + y^n = z^n}$$

possède **une solution** :

$$(x_0, y_0, z_0)$$

avec x_0, y_0, z_0 entiers »

Soit le programme P satisfaisant la spécification précédente.

```
entier n, total:=3 ,x,y,z;
aleatoire(x,y)
tanQue( exp(x,n)+ exp(y,n)= exp(z,n) )
    debut
       pour(x :=1; x<=total-2; x++)
            debut
            Pour (y:=1; x<=total-x-1; y++)
                        début
                        z:=total-x-y;
                        finpour
            finpour
    total++;
 fintanQue
afficher ("hello world");
```

Solution pratique

Dans la pratique, la solution utilisée consiste à **évaluer** et **allouer** un temps t_a pour l'exécution de l'algorithme.

On **arrête** l'exécution dès que la durée d'exécution t **dépasse** ce temps alloué :

$$t \geq t_a$$

Cette solution présente deux inconvénients évidents:

Inconvénient 1 :

Les algorithmes qui **"bouclent"** vont de toute façon épuiser la totalité du temps t_a qui leur est alloué.

Inconvénient 2 :

Le temps alloué t_a peut être:

- très approximatif
- voire impossible à évaluer.

Sous cette dernière hypothèse, l'algorithme risque de tourner court:

- on l'arrête dès

$$t \geq t_a$$

-alors qu'il pourrait aboutir **très peu** de temps après lorsque:

$$t = t_a + \Delta t_a$$

Solution algorithmique

La solution "idéale" serait de disposer d'un algorithme qui :

-étant donné un algorithme **A**
-et ses données en entrée **D**

puisse nous indiquer si A peut ou non se terminer en s'exécutant avec les entrées D.

Question

Doit-on investir dans la recherche d'un tel algorithme ?

Réponse

On va montrer qu'un tel algorithme **n'existe pas**: le **problème de l'arrêt** n'est **pas calculable**.

Preuve d'incalculabilité

Supposons qu'un tel algorithme **existe**. On le nomme :

testArrêt.

Il est censé :

- résoudre le **problème de la terminaison**
- pour **n'importe quel** algorithme.

testArrêt admet deux données en entrée:

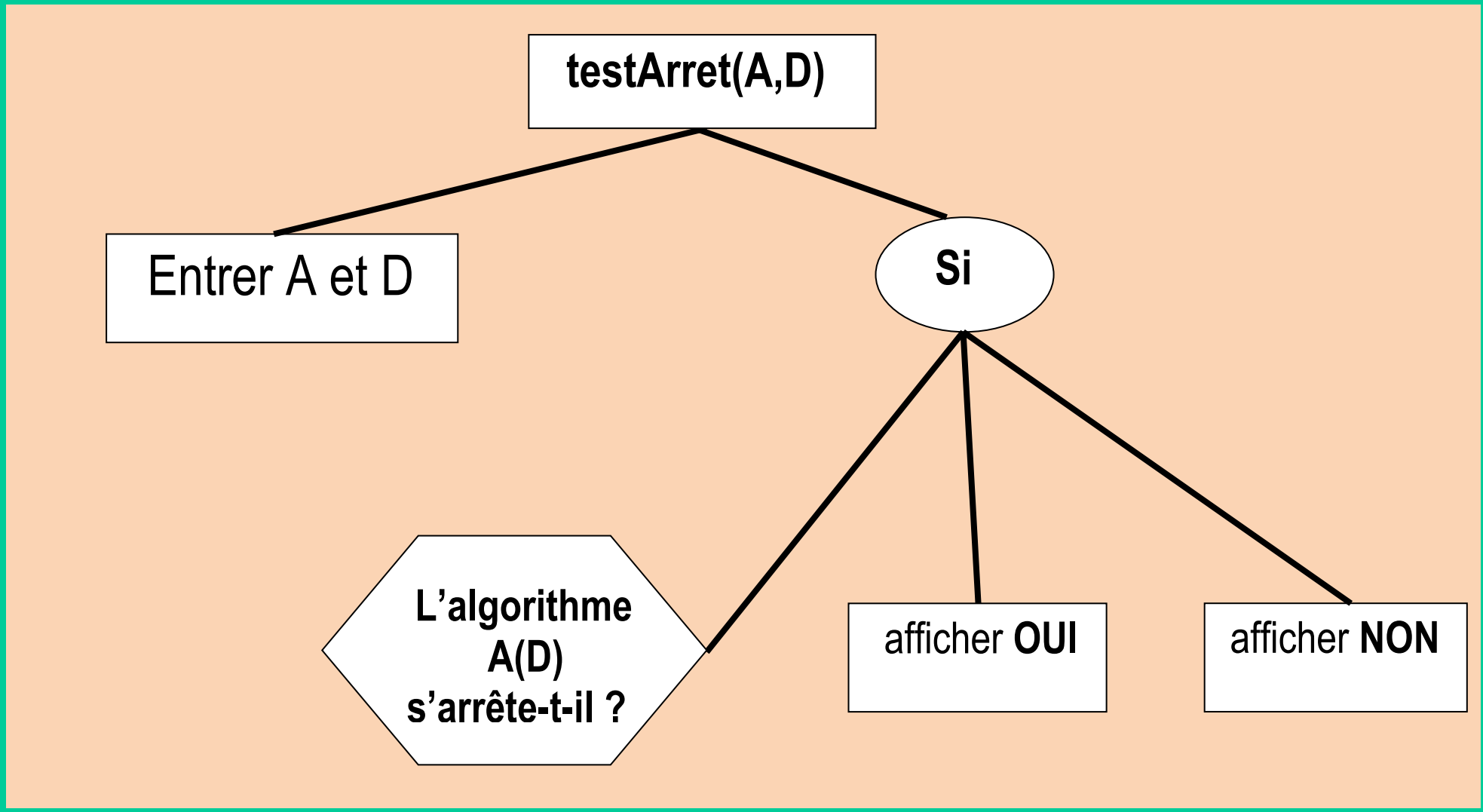
-l'algorithme **A**, à tester

-ses données **D** d'exécution.

Le programme **testArrêt**(**A**,**D**) peut retourner :

- « OUI » si **A** se termine lorsqu'il est exécuté avec les données **D**
- « NON » sinon.

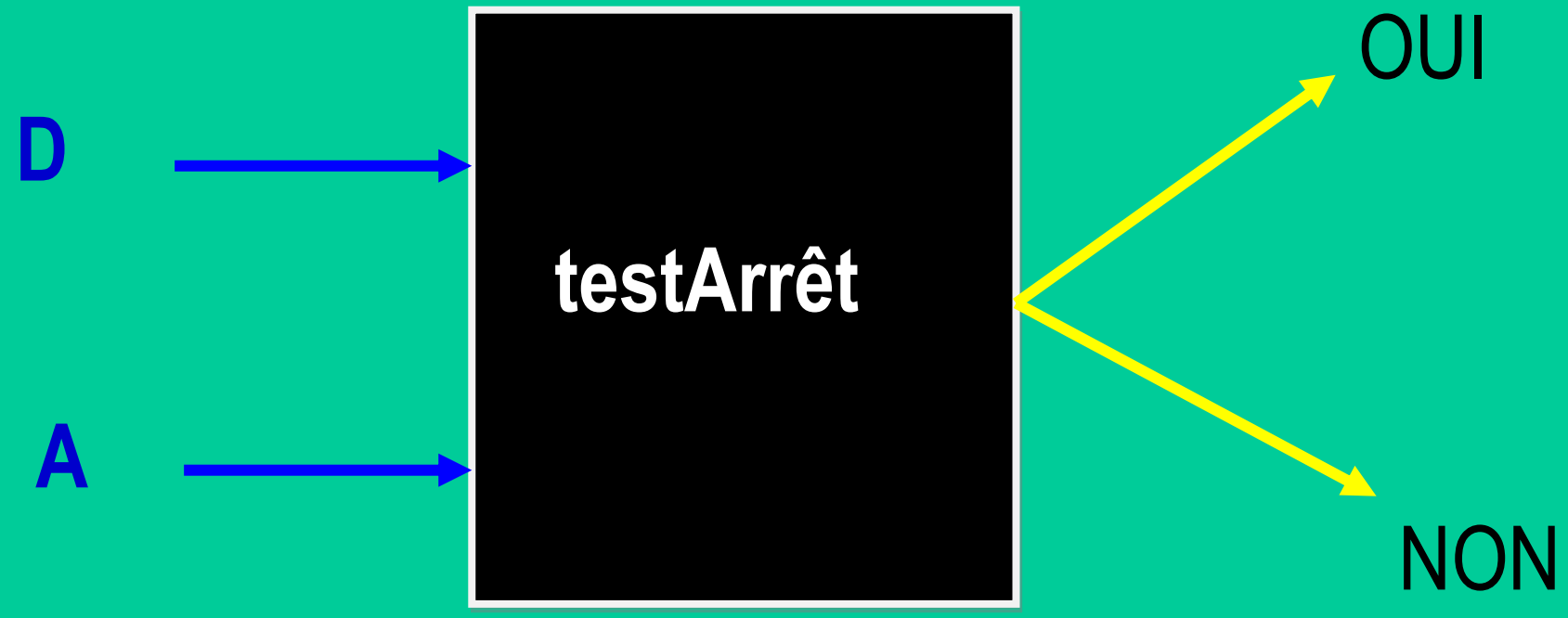
Voici l'exécution de **testArret** (A,D)



Voici la procédure **testArrêt**

```
testArrêt (A, D)
début

  si  (A "se termine en s'exécutant avec D")
        alors    afficher "OUI"
        sinon    afficher "NON"
  finsi
fin
```


D
A
testArrêt
OUI
NON

testArrêt est censé déterminer **systématiquement**:

- la terminaison de **n'importe quel** algorithme **A**
- pour **n'importe quel ensemble** de données **D**.

En particulier, on peut l'utiliser pour tester si le programme **P** précédent s'arrête.

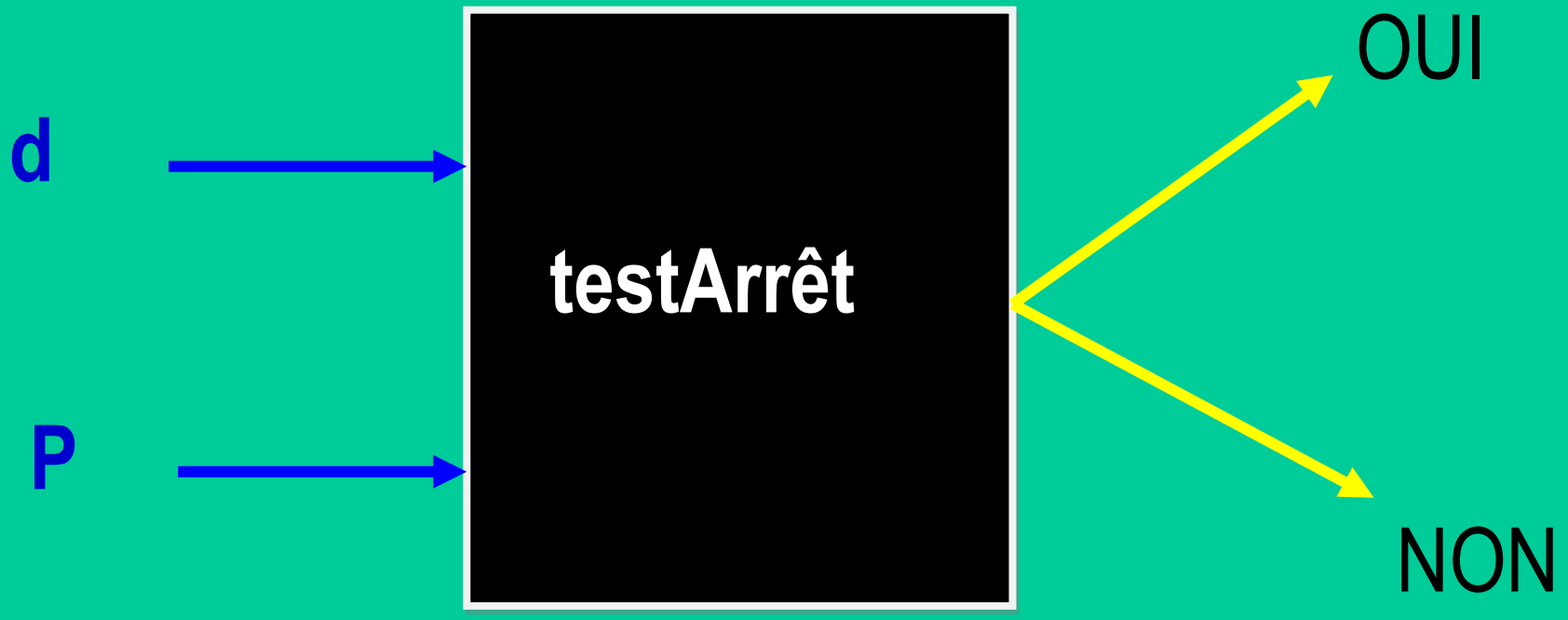
Si **P** s'arrête, alors il affiche "**hello world**".

Dans ce cas **testArrêt**:

-**prend en entrée** un programme **P** et une entrée **d**

-et **teste** si **P** affiche "**hello world**" quand il prend en entrée **d**.

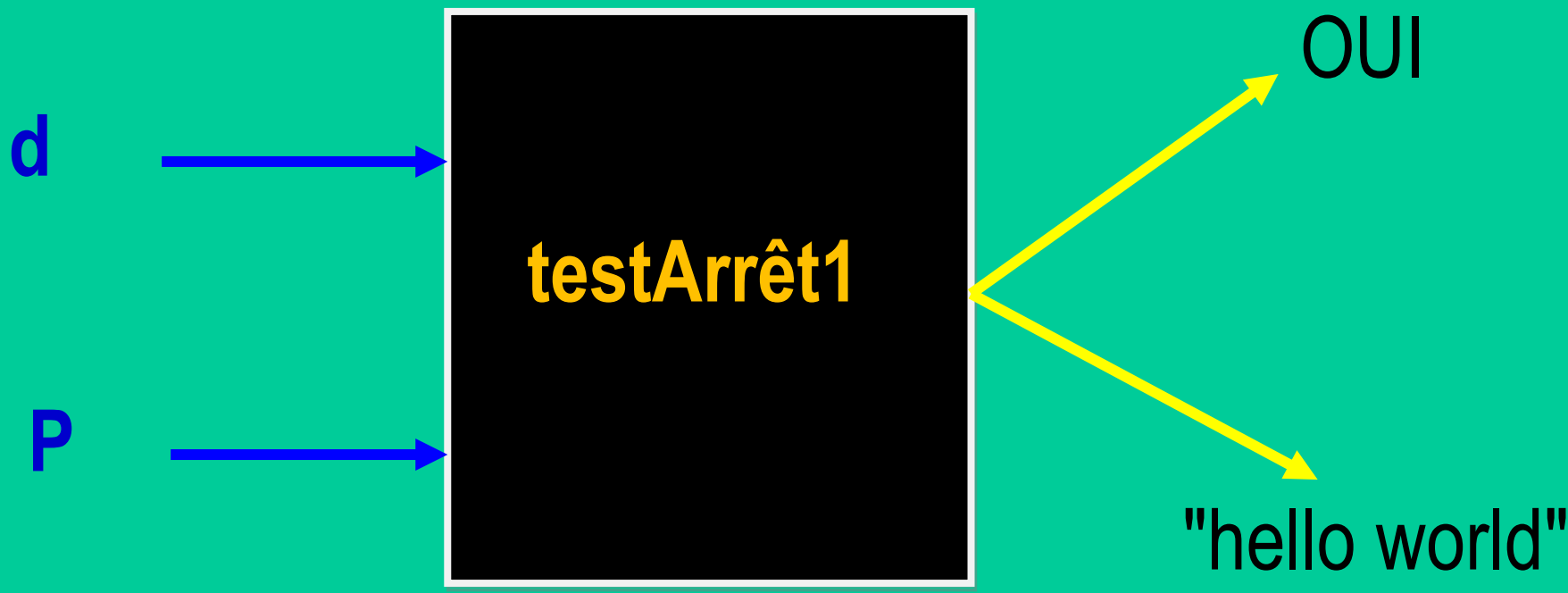
Si **P** affiche "**hello world**", alors **testArrêt** affichera "**oui**" sinon il affichera "**non**".



On modifie **testArrêt** pour obtenir **testArrêt1**.

testArrêt1 fait la même chose que **testArrêt** sauf que:

- il affiche **"hello world"**
- à la place de "**NON"**



Si **P** affiche "hello world", alors **testArrêt1** affichera "**oui**"
sinon il affichera **"hello world"**

On remplace ensuite **testArrêt1** par **testArrêt2** .

testArrêt2 simule le comportement de **testArrêt1**avec comme entrées :

- **P**
- et une copie de **P**.

Si **P** affiche "hello world", alors **testArrêt2** affichera "**oui**"
sinon il affichera **"hello world"**

Que fait **testArrêt2** quand il a en entrée une copie de lui-même: **testArrêt2** ?

Si :

testArrêt2 affiche **oui**

alors : **testArrêt2** affiche **"hello world"**

d'où la **contradiction !**

Si:

testArrêt2 affiche **"hello world"**

alors **testArrêt2** affiche **oui**.

d'où la contradiction !

En d'autres termes, l'exécution de **testArrêt2** ne peut :

- ni s'arrêter,

- ni boucler.

Cette contradiction ne peut être résolue qu'en admettant que **testArrêt2 ne peut pas exister !**

Or, la seule affirmation qui permet d'établir que **testArrêt2** existe est que **testArrêt1** existe.

Enfin, la seule affirmation qui permet d'établir que **testArrêt1** existe est que **testArrêt** existe.

Par conséquent si **testArrêt2** ne peut pas exister, il est évident que **testArrêt1** ne le peut pas non plus.

Et enfin, si **testArrêt1** ne peut pas exister, il est évident que **testArrêt** ne le peut pas non plus.

Ainsi, il est clair qu'il n'existe aucun algorithme **testArrêt** pour résoudre le **problème de l'arrêt**.

Conclusion:

Le problème de l'arrêt est **indécidable**

Preuve par diagonalisation.

La démonstration utilisée peut se résumer 4 étapes:

Etape1:on suppose qu'on peut concevoir un algorithme **testArrêt**,

Etape2: on l'utilise pour construire un autre algorithme **testArrêt2** en passant par un algorithme **intermédiaire testArrêt1**,

Etape3: on montre que **testArrêt2** a une propriété impossible: il ne peut ni se terminer, ni boucler,

Etape4: enfin, on conclut que l'affirmation de l'étape 1 est **impossible**.

Cette méthode de preuve est appelée **diagonalisation**.

Elle est fondée sur le fait d'**autoréférence**: un algorithme peut s'appliquer à lui-même.

De façon générale, cette méthode :

-peut s'avérer fastidieuse et difficile à suivre

-pour montrer qu'un problème est non calculable.

Donc, plutôt que de la répéter à chaque fois que le cas se présente, on se contente seulement de montrer que :

- s'il existait une solution algorithmique pour le problème considéré

- alors il existerait également une solution pour le **problème de l'arrêt**.

Comme le problème de l’arrêt n’est pas calculable, nous aurons montré que le problème considéré ne l’est pas non plus.

II-Preuve de terminaison

Pour montrer qu'un algorithme :

-se **termine**

-quel que soit le jeu de paramètres passé en entrée

il faut montrer que l'exécution que chaque **bloc** élémentaire se termine !

Les **blocs simples**, quelque le langage utilisé, sont :

— boucles **for** ;

— boucles **while** ;

— blocs conditionnels **if**, **else if**,...,**else**.

Le **découpage** d'un algorithme en blocs simples est essentiel.

Or, les blocs **for** et les blocs conditionnels **if**, **else if,** ...,**else** se **terminent** forcément.

Le souci ne pourrait venir donc que des blocs **while** !

Considérons par exemple le code simple suivant:

```
while n≠0
        n :=n-1
```

Si, avant la boucle **while**, la variable **n** contient un entier positif,

…alors cette boucle s'arrêtera au bout de **n** étapes.

Par contre, si elle contient un entier strictement négatif, c'est la catastrophe !

En effet, **n** prendra une **infinité** de valeurs, toutes strictement négatives.

Principe

Mise en évidence l'existence d'un **convergent.**

Un **convergent** est une quantité qui :

-qui diminue à chaque passage,

- et vit dans un ensemble **bien fondé**.

Rappel

Un ensemble **bien fondé** est un ensemble :

-totalement ordonné

-dans lequel il n'existe pas de **suite infinie strictement décroissante**.

En particulier :

$\mathbb{N}$, ou $\mathbb{N}^k$

munis de l'ordre **lexicographique**, sont des ensembles bien fondés.

On a $(a; b) < (c; d)$ pour l'ordre lexicographique lorsque $a < c$ ou $a = c$ et $b < d$.

Exemple du calcul du pgcd de deux entiers

```
Algorithme PGCD(a,b)
    début
     a , b:  e n t i e r s
     Variables l o c a l e s : x , y , r
     x := a ; y := b ;
     tantQue y != 0 f a i r e
                r := x % y
                x := y
                y := r      // ici y est le convergent
                            // nouvelle valeur de y < ancienne valeur
    finTantQue
    retourner x
    fin
```

L’algorithme se termine si la **condition d’arrêt** de la boucle se réalise:

$$y = 0$$

Dans la boucle, y est remplacé par x % y

Donc, à chaque passage, y **décroît strictement**, tout en restant >0

Par conséquent, **y** finit par atteindre 0.

Alors, on sort alors de la boucle **tantQue**: l'algorithme **se termine**

Terminaison de la boucle

Attention les deux conditions de convergent :

- **entier**
- et **strictement monotone**

sont généralement nécessaires pour qu'il finisse par dépasser une valeur seuil.

Ce seuil est un :

-"plancher" dans le cas décroissant ,

-"plafond" dans le cas croissant.

```
k = 2
n = 0
while (k >= 0):
    k = k - 1./(2**n)
    n = n + 1
```

Avec des valeurs de k **non entières** la condition que k est strictement décroissante ne suffit pas pour que le **convergent** prenne des valeurs strictement négatives.

En effet, ici:

$$1/2^0 + 1/2^1 + 1/2^2 + \ldots + 1/2^n < 2$$

Avec des **valeurs réelles** la boucle ne s'arrête pas car **k** ne prendrait que des valeurs positives :

$$k = 2 ;$$

$$k = k - (1/2^0 + 1/2^1 + 1/2^2 + \ldots + 1/2^n)$$

$$= 2 - (1/2^0 + 1/2^1 + 1/2^2 + \ldots + 1/2^n) > 0$$

III. Cas de problèmes non calculables

Après le problème de la terminaison, la question qui vient naturellement est :

« existe-t-il d'autres problèmes connus qui ne soient pas calculables ? »

On se propose de survoler quelques exemples classiques de ces problèmes.

Le problème de la totalité

Le problème de l'arrêt consiste à déterminer si :

-un algorithme quelconque A se termine ou non

-quand il est exécuté avec un **ensemble de données en entrée D**.

Une question corrélative consiste à savoir si oui ou non un algorithme quelconque A se termine **pour toute entrée quelle qu'elle soit**.

C'est ce qu'on appelle le problème de la **totalité** ou de l'**uniformité**.

Il n'est pas surprenant de constater que ce problème aussi ne comporte pas de solution algorithmique.

Le problème de l'équivalence

Supposons qu'une entreprise décide de s'équiper d'un nouvel ordinateur.

Elle réécrit alors pour cet ordinateur l'ensemble de ses logiciels qui ont déjà fonctionné sans erreur sur l'ancien ordinateur.

Comment peut-elle être sûre que ces nouveaux programmes seront aussi **fiables** que les anciens ?

Il est clair que la solution informatique idéale à ce problème consisterait à concevoir un algorithme.

Cet algorithme examine deux programmes :

-l'ancien,

-le nouveau

et les comparer pour déterminer s'ils mènent à bien la même tâche.

C'est ce que l'on appelle le problème de l'**équivalence** qui est aussi **non calculable**.

IV-Calculabilité partielle

Il existe en informatique de nombreux problèmes qui ne sont **pas calculables**.

Cependant, certains d'entre eux sont **encore moins calculables** que d'autres.

Considérons encore une fois le problème de la terminaison.

Pour un algorithme quelconque A et des données D, on aimerait savoir si A(D) se termine ou pas.

S'il se termine, il n'y a aucune difficulté pour le savoir : il suffit simplement de l'exécuter jusqu'à ce qu'il se termine.

On saura alors qu'il se termine.

Le problème survient uniquement, s'il ne se termine pas.

Dans ce cas, comme on l'a déjà vu, il n'existe pas d'algorithmique, qui puisse **affirmer que A(D) ne s'arrête pas**.

Peu importe le temps nécessaire à son exécution.

S'il ne s'arrête pas on ne peut rien dire.

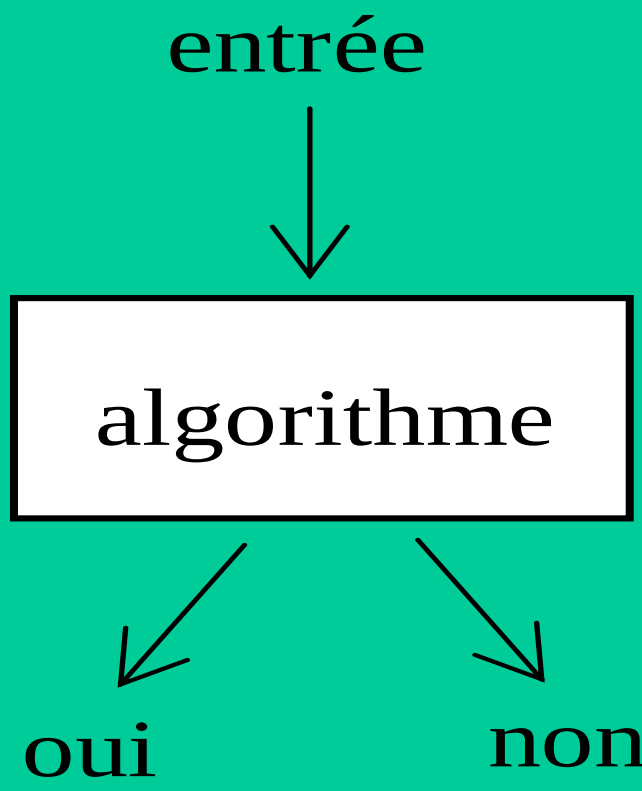
Et encore moins qu'il ne s'arrêtera jamais.

Comme il y a un algorithme qui sort :

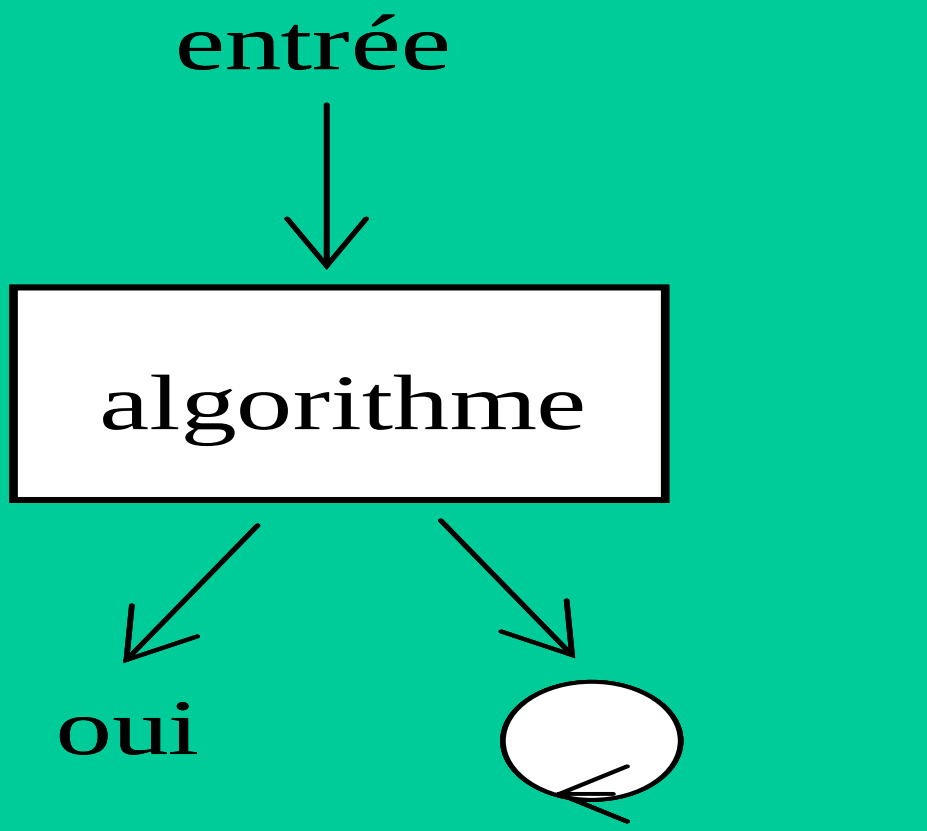
- « OUI » si A(D) se termine
- et qui boucle si A(D) ne se termine pas,

Le problème de l'arrêt est dit **partiellement calculable**.

Les schémas illustrent la différence entre ce qui est calculable et ce qui ne l'est que **partiellement.**



Problème calculable



Problème partiellement calculable

Contrairement au problème de l'arrêt, les problèmes :

- de la **totalité**
-et de l'**équivalence**

ne sont **pas partiellement calculables**.

Conclusion

Ce chapitre nous a permis d'approcher certains problèmes informatiques non calculables.

Le plus fameux d'entre eux est le problème de la terminaison.

Mais attention, cela ne veut pas dire qu'on ne peut pas prouver l'arrêt d'un algorithme donné.

Cela signifie juste qu'il n'existe pas de **méthode systématique** qui fonctionne à tous les coups.

Nous avons aussi appris qu'il existe des nuances dans la notion de non calculabilité.

En effet parmi les problèmes non calculables, il y en a qui sont plus calculables que d'autres: d'où la propriété de **calculabilité partielle**.

Cette propriété que possède le problème de la terminaison, joue un rôle important dans les systèmes de preuve.

En effet :

-pour tout problème **calculable** ou **partiellement** calculable,

-il existe une **méthode pour prouver** que la réponse est correcte chaque fois qu'elle l'est.

Cependant, si le problème :

- n'est **pas partiellement** calculable,
- il n'existe **pas de méthode générale** pour prouver l'exactitude d'une réponse donnée.