# Cours 7

M.Adam – JF.Kamp – L.Naert

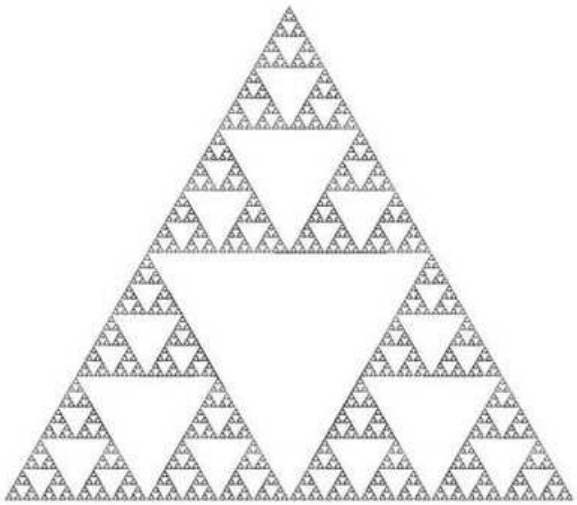
18 octobre 2021

## Table des matières

# 1 Première approche de la récursivité

## 1.1 La vache qui rit


## 1.2 Le triangle de Sierpinski



Par PiAndWhippedCream — Travail personnel, Domaine public, https ://commons.wikimedia.org/w/index.php
rid=2155993

Évolution du triangle de Sierpinski

Par !Original :SaperaudVector : Wereon — Travail personnel basé sur : Sierpinsky triangle (evolution).png, Domaine public, https ://commons.wikimedia.org/w/index.php ?curid=1357937

# 2 La récursivité

## 2.1 Définition

Une méthode est dite récursive si elle s'appelle directement ou indirectement.

## 2.2 Récursivité directe

```
void p (...) {
    ...
    p (...);
    ...
}
```

La méthode `p()` est une méthode récursive.

### 2.2.1 Récursivité indirecte

```
void p (...) {
    ...
    f (...);
    ...
}
```

```
void f(...) {
    ...
    p (...);
    ...
}
```

Les méthodes `p()` et `f()` sont des méthodes récursives

### 2.2.2 L'exemple du PGCD

Pour rappel, un entier q est le Plus Grand Commun Diviseur (PGCD) de deux entiers a et b ssi :

- q divise a et q divise b,
- il n'existe pas de plus grand entier que q divisant à la fois p et q.

#### 2.2.3 Méthode soustractive

- $PGCD(a,b) = PGCD(a-b,b)$ si $a > b$,
- $PGCD(a,b) = PGCD(a,b-a)$ si $a < b$,
- $PGCD(a,b) = a$ si $a = b$.

### 2.3 Construction d'une méthode récursive

Les étapes à suivre :

- déterminer la signature de la méthode,
- écrire les appels récursifs,
- écrire les cas d'arrêt :
  - la solution est trouvée,
  - l'appel suivant provoquerait une erreur.

#### 2.3.1 La signature de la méthode

```
/**
* calcule le PGCD de deux entiers positifs
* @param a premier entier
* @param b deuxième entier
* @return le PGCD de a et b
*/
int pgcd (int a, int b) {

}
```

#### 2.3.2 Les appels récursifs

```
/**
* calcule le PGCD de deux entiers positifs
* @param a premier entier
* @param b deuxième entier
* @return le PGCD de a et b
*/
int pgcd (int a, int b) {
    if (a > b) {return pgcd (a - b, b);}
    if (a < b) {return pgcd (a, b - a);}
}
```

### 2.3.3 Les cas d'arrêt

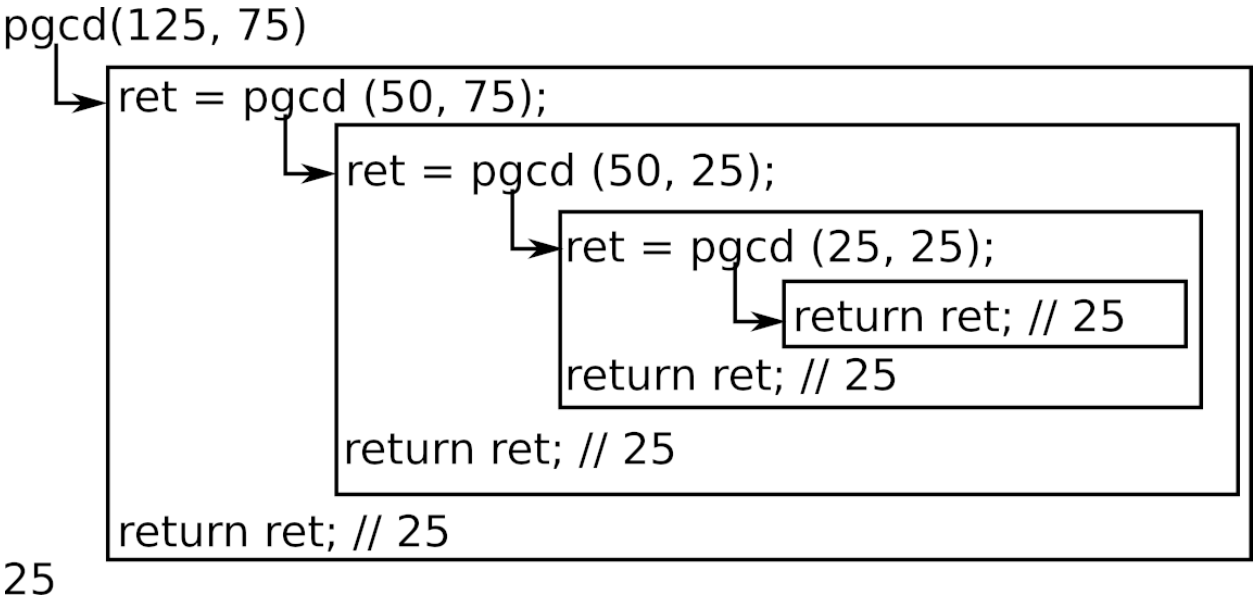
```
/**
* calcule le PGCD de deux entiers positifs
* @param a premier entier
* @param b deuxième entier
* @return le PGCD de a et b
*/
int pgcd (int a, int b) {
    if (a > b) {return pgcd (a - b, b);}
    if (a < b) {return pgcd (a, b - a);}
    return a; // a == b
}
```

### 2.3.4 Plus conforme à notre convention

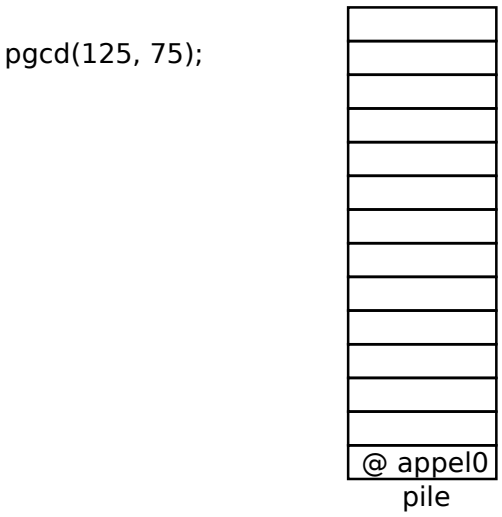
```
/**
* calcule le PGCD de deux entiers positifs
* @param a premier entier
* @param b deuxième entier
* @return le PGCD de a et b
*/
int pgcd (int a, int b) {
    int ret = a; // a == b
    if (a > b) {ret = pgcd (a - b, b);}
    if (a < b) {ret = pgcd (a, b - a);}
    return ret;
}
```

# 3 L'exécution

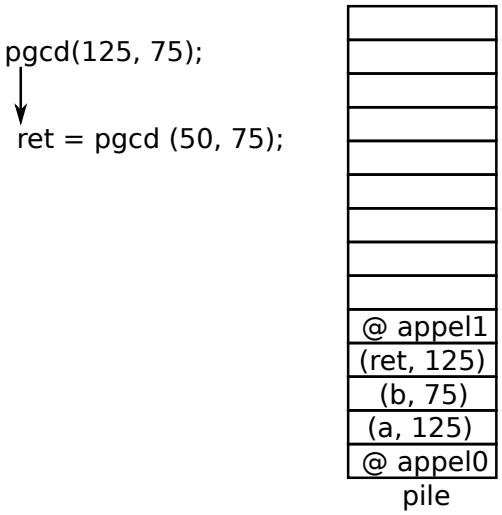
## 3.1 Le déroulement de l'exécution



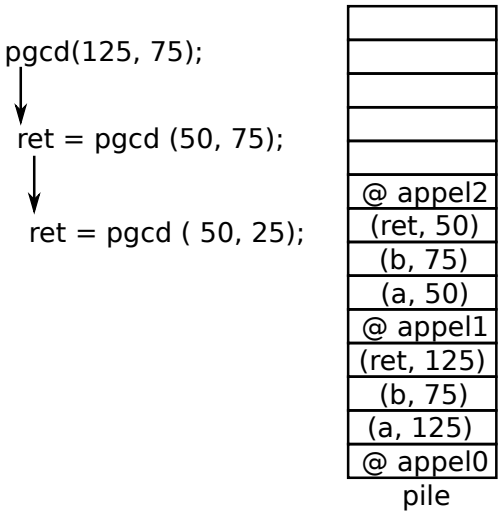

## 3.2 La pile d'exécution



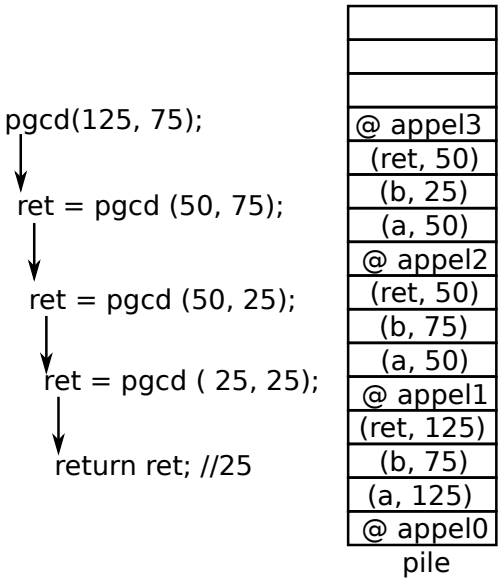
### 3.2.1 Appel 1



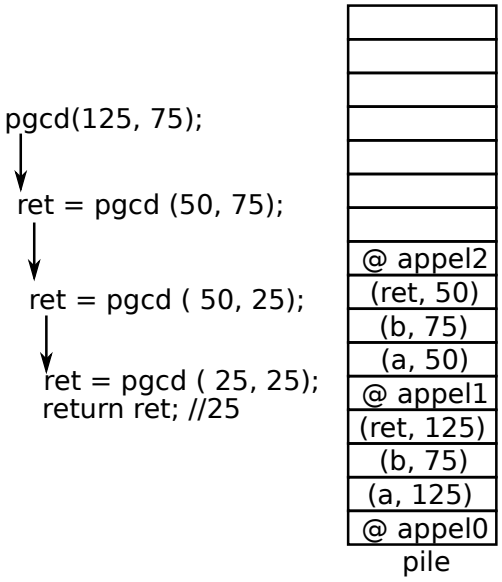

### 3.2.2 Appel 2



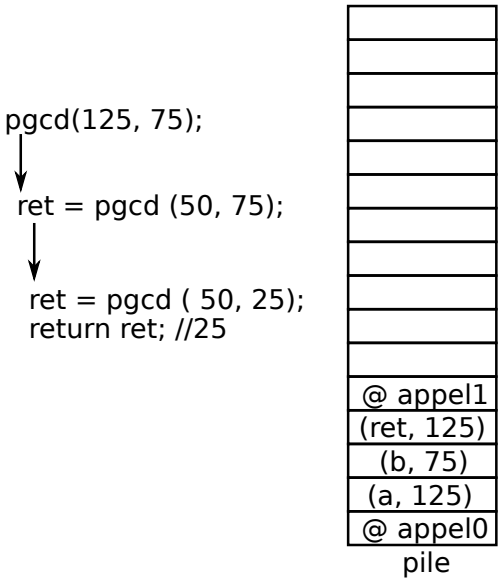
### 3.2.3 Appel 3



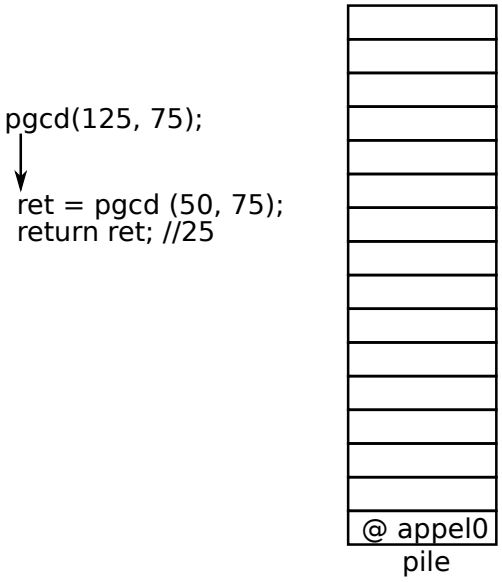

### 3.2.4 Retour de l'appel 3



### 3.2.5 Retour de l'appel 2




### 3.2.6 Retour de l'appel 1



# 4 Récursivité et itération

## 4.1 Itération vers Récursivité

Toute itération peur être transformée en une méthode récursive.

### 4.1.1 Exemple de la factorielle

```
int factorielle (int n) {
   int ret = 1;
   while (n > 1) {
      ret = n * ret;
      n = n - 1;
   }
   return ret;
}
```

```
int factorielle (int n) {
    int ret = 1;
    if (n > 1) {
        ret = n * factorielle (n - 1);
    }

    return ret;
}
```

Ce type de récursivité est dite à droite ou terminale.

## 4.2 Récursivité vers itération

Toute méthode récursive peut s'écrire sous forme itérative.

- pour la récursivité à droite c'est facile,
- pour les autres formes c'est plus compliqué.

## 4.3 Quel choix ?

La récursivité est consommatrice en mémoire au niveau de la pile. Le meilleur choix est donc l'itération.

Quand l'écriture du code est plus "simple" de manière récursive, il faut sans doute favoriser la récursivité.

# 5 Retour sur trace

## 5.1 Définition

Le retour sur trace ou retour arrière (backtracking en anglais) est une famille d'algorithmes pour résoudre des problèmes de satisfaction de contraintes (optimisation ou décision).

Ces algorithmes permettent de tester systématiquement l'ensemble des affectations potentielles du problème.

Ils consistent à sélectionner une variable du problème, et pour chaque affectation possible de cette variable, à tester récursivement si une solution valide peut-être construite à partir de

cette affectation partielle. Si aucune solution n'est trouvée, la méthode abandonne et revient sur les affectations qui auraient été faites précédemment.

## 5.2 L'exemple de la somme d'un sous-ensemble

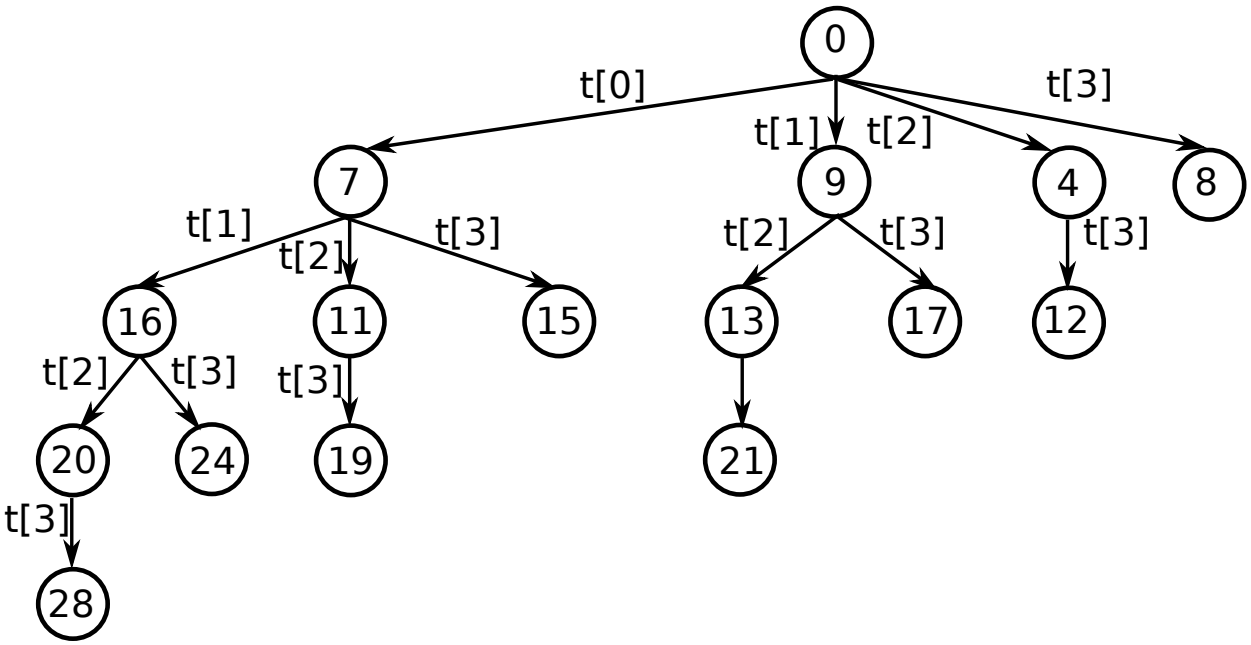
Soit un tableau `t` d'entiers, tous positifs et différents. Le problème à résoudre consiste à déterminer si il existe une sous-somme des élements de `t` égale à une valeur `k`.

- 21 est une sous-somme de `{7, 9, 4, 8}`,
- 22 n'est pas une sous-somme de `{7, 9, 4, 8}`.

Librement inspiré de https://www.geeksforgeeks.org/

### 5.2.1 Esquisse de solution

Une solution consiste à calculer toutes les sous-sommes possibles jusqu' à trouver ou pas le nombre `k`.




Il y a donc 16 soit $2^4$ totaux. Avec un tableau de n entiers, le nombre de valeurs calculables est de $2^n$.

### 5.2.2 La signature de la méthode

```
/**
 * déterminer si un nombre est sous-somme d'un tableau
```

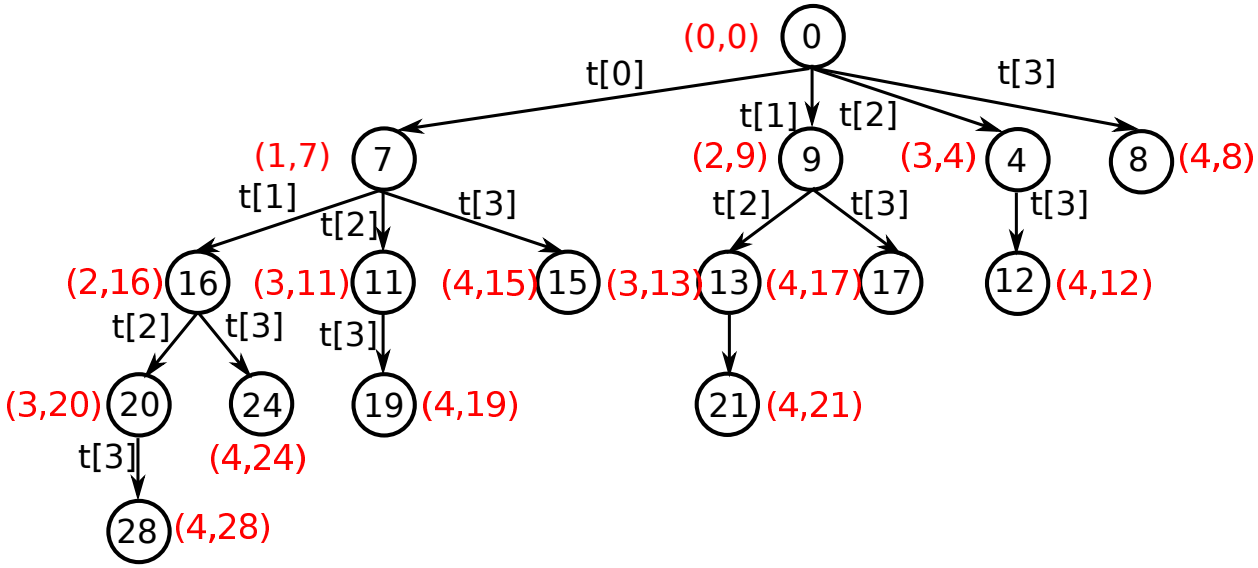
```
 * @param t tableau d'entiers positifs
 * @param somme sous-somme à déterminer
 */
boolean estSousSomme (int[] t, int somme) {
```

Cette méthode n'est pas récursive, il manque des informations pour pouvoir utiliser la récursivité.

### 5.2.3 La méthode récursive

Il faut ajouter deux informations :

- la somme des valeurs parcourues,
- la marque des valeurs à parcourir.




```
/**
 * Méthode récursive pour déterminer si un nombre est sous-somme d'un tableau
 * @param t tableau d'entiers positifs
 * @param somme sous-somme à déterminer
 * @param init premier indice à traiter
 * @param cumul total calculé des valeurs traitées
 */
boolean estSousSommeRec (int[] t, int somme, int init, int cumul) {
    boolean ret = false;
    if (somme == cumul) {
        ret = true;
    } else {
        int i = init;
        while (i < t.length && !ret) {
            ret = estSousSommeRec (t, somme, i + 1, cumul + t[i]);
```

```
            i = i + 1;
        }
    }
    return ret;
}
```

### 5.2.4 Le début de la récursivité

```
/**
 * déterminer si un nombre est sous-somme d'un tableau
 * @param t tableau d'entiers positifs
 * @param somme sous-somme à déterminer
 */
boolean estSousSomme (int[] t, int somme) {
    return estSousSommeRec (t, somme, 0, 0);
}
```

### 5.2.5 Quelles améliorations ?

La solution proposée consiste à énumérer les $2^n$ cas possibles avec $n$ la taille du tableau jusqu'à trouver, ou pas, la somme recherchée. Comment être plus efficace ?

-
-
-
-
-

Ce problème est une variante du problème du sac à dos.

## 6 Et pour en finir avec la récursivité

- Une méthode est dite récursive si elle s'appelle directement ou indirectement
- Toute itération peur être transformée en une méthode récursive
- Toute méthode récursive peut s'écrire sous forme itérative
- L'itération est à privilégier par rapport à la récursivité
- La récursivité peut être utilisée quand elle rend la lecteure, la mise au point du code plus facile
- La récursivité est adaptée à la résolution des problèmes d'essai-erreurs avec retour arrière (back tracking)