Récursivité. Paradigme diviser pour régner.

Développeur Python

Sommaire

1. Récursivité.
2. Paradigme “diviser pour régner“.

1. Récursivité.

1. Récursivité.

Définition et principe

- Un sous-programme (procédure ou fonction) est dit récursif s'il s'appelle lui-même.

- Pour effectuer une tâche ou un calcul, on se ramène donc à la réalisation d'une tâche similaire mais de complexité moindre.

- On recommence ainsi jusqu'à obtenir une tâche élémentaire.

1. Récursivité.

Principe

- En pratique un sous-programme récursif va s'appeler lui-même avec un paramètre plus “petit“.
- Cet appel en induira un autre, puis un autre, etc. D'appel en appel, la taille du paramètre va ainsi diminuer.
- On s’arrêtera quand cette taille sera celle d'un problème immédiatement résolvable.

1. Récursivité.

Principe

- Les différents problèmes intermédiaires, ceux permettant de passer du problème initial au problème élémentaire, seront stockés successivement en mémoire dans une pile.

- On utilisera ainsi en premier le résultat du problème élémentaire, puis de proche en proche on arrivera à celui du problème initial.

1. Récursivité.

Exemple : la fonction factorielle

- Rappelons que

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

- On a donc la formule de récurrence

$$n! = n \times (n-1)!$$

1. Récursivité.

Exemple : la fonction factorielle

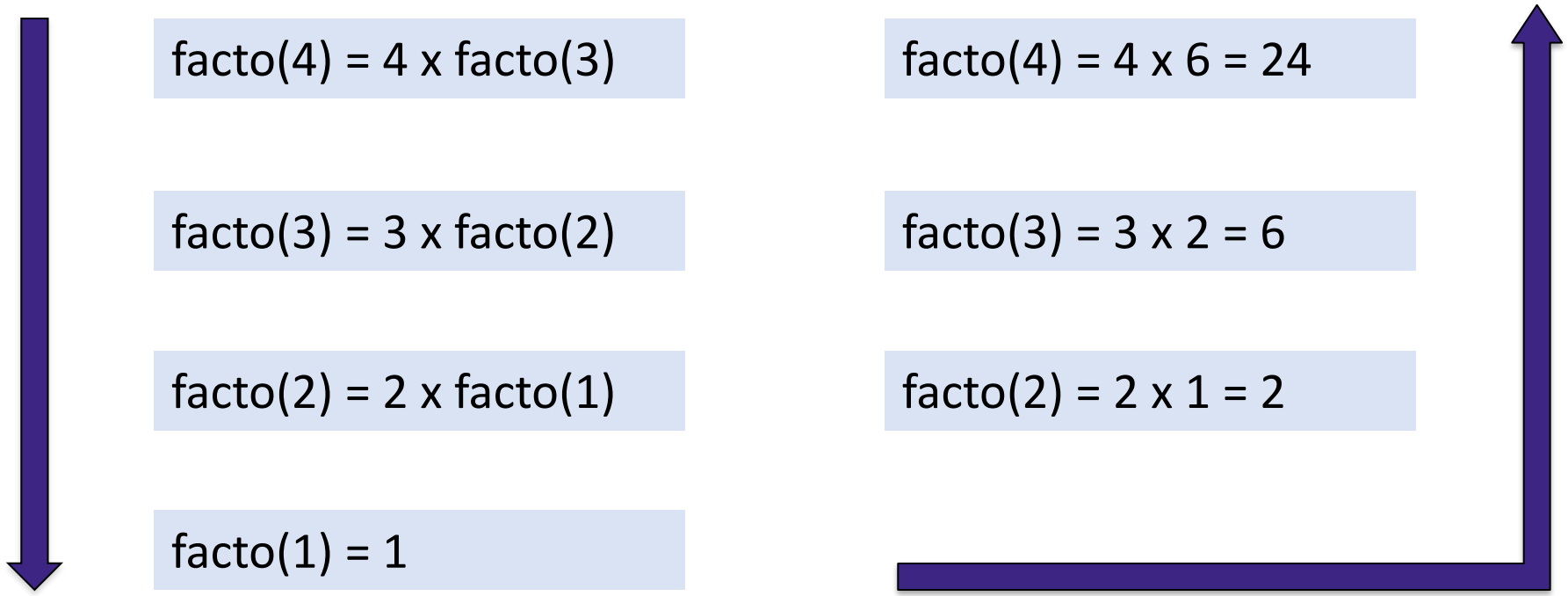
- Implémentation récursive :

```
def factorielleRecursive(n):
    if n==0 or n==1:
        return 1
    else:
        return n*factorielleRecursive(n-1)
```

1. Récursivité.

Exemple : la fonction factorielle

- Exécution de la fonction pour $n = 4$:



1. Récursivité.

Remarque fondamentale

- Il est indispensable de prévoir une condition d'arrêt à la récursion sinon le programme va s'appeler une infinité de fois.

- Exemple à ne pas suivre :

```
def factorielleRecursiveBadJob(n):
      return n*factorielleRecursiveBadJob(n-1)
```

1. Récursivité.

Récursivité versus itération

- Par opposition, on qualifiera d’itératif un sous-programme qui ne s'appelle pas.
- On peut démontrer qu'il est toujours possible de transformer un algorithme récursif en un algorithme itératif et inversement.
- L’algorithme itératif sera plus rapide une fois implémenté dans un langage de programmation mais souvent plus complexe à écrire.

1. Récursivité.

Exemple : la fonction factorielle

- Implémentation itérative :

```
def factorielleIterative(n):
    resultat = 1
    for i in range(2,n+1):
        resultat *= i
    return resultat
```

1. Récursivité.

Intérêts de la récursivité

- Technique de programmation très élégante et lisible (elle évite souvent le recours à de nombreuses structures itératives).

- Elle est très utile pour concevoir des algorithmes sur des structures complexes comme les listes, les arbres et les graphes.

1. Récursivité.

Inconvénient majeur de la récursivité

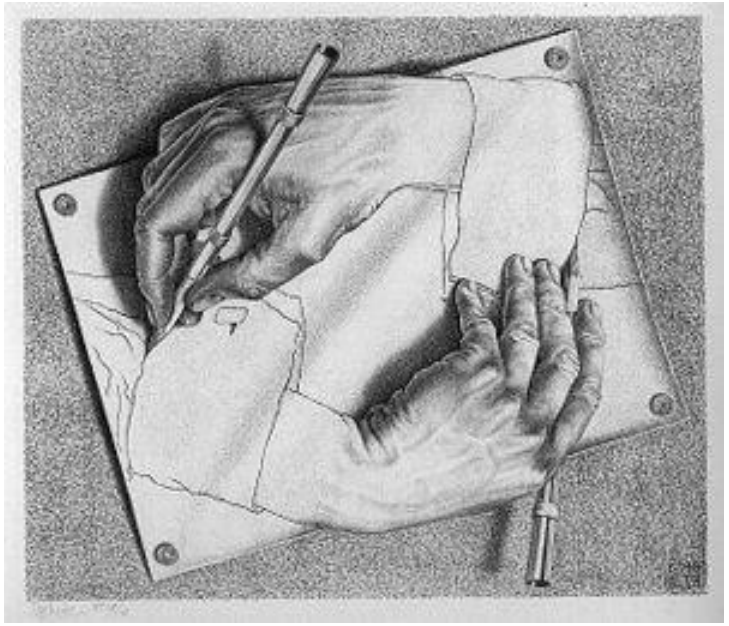
- Une fois implémentée dans un langage de programmation, cette technique est très gourmande en mémoire.

- Elle peut même provoquer des débordements de capacité.

1. Récursivité.

Récursivité croisée : définition

- C’est un cas bien particulier de récursivité où une fonction appelle une autre fonction qui elle-même appelle la première.



1. Récursivité.

Récursivité croisée : exemple

- Fonction récursive testant la parité d'un nombre :

```
def pair(n):
    if n == 0:
        return True
    else:
        return impair(n-1)

def impair(n):
    if n == 0:
        return False
    else:
        return pair(n-1)
```

1. Récursivité.

Récursivité multiple : définition

- Cas de figure où un sous-programme récursif réalise plusieurs appels à lui-même.

1. Récursivité.

Récursivité multiple : exemple

- Formule de récurrence des coefficients binomiaux

$$\binom{n}{k} = \begin{cases} 1 & \text{si} \quad k = 0 \\ 1 & \text{si} \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases}$$

1. Récursivité.

Récursivité multiple : exemple

- Fonction récursive calculant les coefficients binomiaux :

```
def coeffs(n,k):
    if k == 0 or k == n:
        return 1
    else:
        return coeffs(n-1,k-1) + coeffs(n-1,k)
```

1. Récursivité.

Récursivité imbriquée : définition

- Cas de figure où l'appel à lui-même d'un sous-programme récursif contient un autre appel à lui-même.

1. Récursivité.

Récursivité imbriquée : exemple

- La fonction 91 de McCarthy, définie sur $\mathbb{Z}$ par

$$f(n) = \begin{cases} n - 10 & si \quad n > 100 \\ f\big(f(n + 11)\big) & si \quad n \leq 100 \end{cases}$$

1. Récursivité.

Récursivité imbriquée : exemple

- Fonction récursive de McCarthy :

```
def f91(n):
    if n > 100:
        return n-10
    else:
        return f91(f91(n+11))
```

1. Récursivité.

2. Paradigme “diviser pour régner“.

2. Paradigme “diviser pour régner“.

Principe

- Les algorithmes de la partie précédente étaient basés pour la plupart sur la simple exploitation d’une formule de récurrence.

- Mais beaucoup de problèmes plus complexes se résolvent aussi naturellement de façon récursive, avec des algorithmes s’appelant eux-mêmes une ou plusieurs fois sur des données de tailles inférieures.

- Il faudra alors combiner les résultats issus de chacun de ces appels.

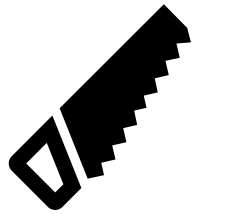
2. Paradigme “diviser pour régner“.

Les trois étapes du paradigme “diviser pour régner“

1. **Diviser** : on divise les données initiales en plusieurs sous-parties.

2. **Régner** : on résout récursivement chacun des sous-problèmes associés (ou on les résout directement si leur taille est assez petite).

3. **Combiner** : on combine les différents résultats obtenus pour obtenir une solution au problème initial.



2. Paradigme “diviser pour régner“.

Premier exemple : calcul du maximum d’une liste de nombres

- Idée de base : calculer récursivement le maximum de la première moitié de la liste et celui de la seconde, puis les comparer.
- Le plus grand des deux sera le maximum de toute la liste.
- La condition d’arrêt à la récursivité sera l'obtention d'une liste à un seul élément, son maximum étant bien sûr la valeur de cet élément.

2. Paradigme “diviser pour régner“.

Premier exemple : calcul du maximum d’une liste de nombres

Résolution :

1. Diviser la liste en deux sous-listes en la “coupant” par la moitié.
2. Calculer récursivement le maximum de chacune de ces sous-listes. Arrêter la récursion lorsque les listes n'ont plus qu'un seul élément.
3. Retourner le plus grand des deux maximums précédents.

2. Paradigme “diviser pour régner“.

Premier exemple : calcul du maximum d’une liste de nombres

- La fonction récursive :

```
def maximumRecursive(l,d,f):
    if d == f:
        return l[d]
    m = (d+f) // 2
    x = maximumRecursive(l,d,m)
    y = maximumRecursive(l,m+1,f)
    return x if x > y else y
```

2. Paradigme “diviser pour régner“.

Premier exemple : calcul du maximum d’une liste de nombres

- La fonction réalisant le premier appel de la fonction récursive :

```
def maximum(l):
    return maximumRecursive(l,0,len(l)-1)
```

2. Paradigme “diviser pour régner“.

Second exemple : recherche d’un élément dans une liste

- Idée de base : rechercher récursivement l’élément dans la première moitié de la liste et dans la seconde, puis de combiner les résultats via l’opérateur logique or.

- En effet, l’élément recherché sera dans la liste s'il est dans la première moitié ou dans la seconde.

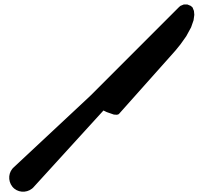
- La condition d’arrêt à la récursivité sera l'obtention d'une liste à un seul élément, car il est alors immédiat de conclure si l’élément recherché appartient à une telle liste ou non.

2. Paradigme “diviser pour régner“.

Second exemple : recherche d’un élément dans une liste

Résolution :

1. Diviser la liste en deux sous-listes en la “coupant” par la moitié.
2. Rechercher la présence de l’élément dans chacune de ces sous-listes. Arrêter la récursion lorsque les listes n'ont plus qu'un seul élément.
3. Combiner avec l’opérateur logique **or** les résultats obtenus.



2. Paradigme “diviser pour régner“.

Second exemple : recherche d’un élément dans une liste

- La fonction récursive :

```
def rechercheRecursive(l,x,d,f):
    if d == f:
        return l[d] == x
    m = (d+f) // 2
    return rechercheRecursive(l,x,d,m) or rechercheRecursive(l,x,m+1,f)
```

2. Paradigme “diviser pour régner“.

Second exemple : recherche d’un élément dans une liste

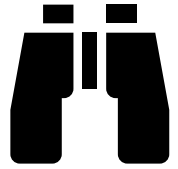
- La fonction réalisant le premier appel de la fonction récursive :

```
def recherche(l,x):
    return rechercheRecursive(l,x,0,len(l)-1)
```

2. Paradigme “diviser pour régner“.

Troisième exemple : recherche d’un élément dans une liste triée

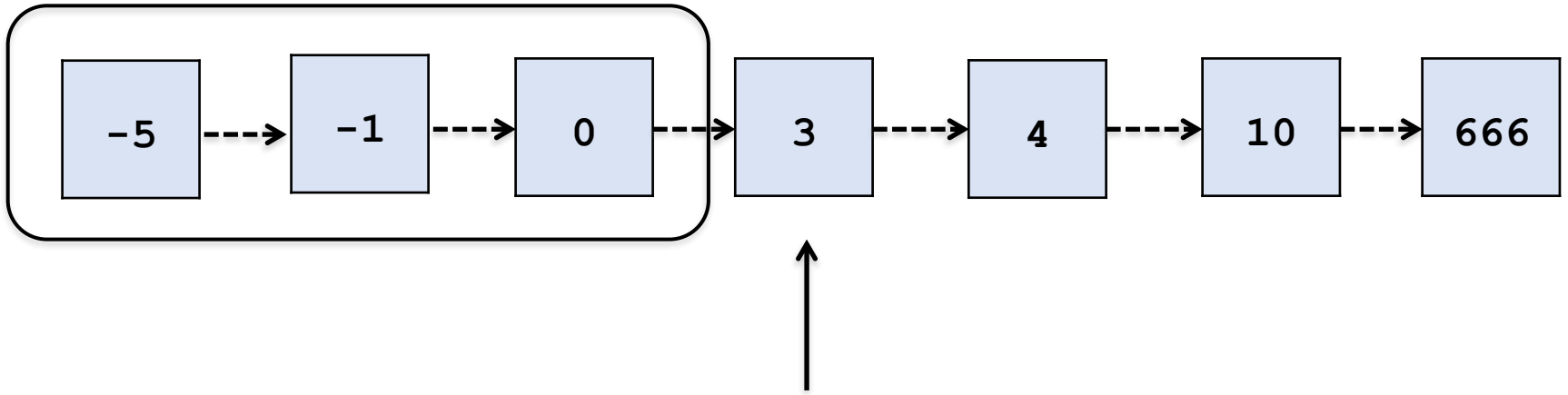
- Idée de base : utiliser une recherche dichotomique.
- La liste étant triée, après comparaison avec l’élément du “milieu“ il est en effet facile de voir dans quelle moitié peut éventuellement se trouver l’élément cherché.
- On aura plus alors qu'à recommencer récursivement la recherche.



2. Paradigme “diviser pour régner“.

Troisième exemple : recherche d’un élément dans une liste triée

- Réduction de la recherche à une seule moitié de la liste, par exemple pour -4 :



- La valeur -4 étant plus petite que la valeur centrale 3, on va donc continuer la recherche uniquement dans la première moitié de la liste.

2. Paradigme “diviser pour régner“.

Troisième exemple : recherche d’un élément dans une liste triée

Résolution :

1. Diviser la liste en deux sous-listes en la “coupant“ par la moitié.
2. Rechercher récursivement la présence de l’élément recherché dans la “bonne” des deux sous-listes après l'avoir comparé à l’élément situé au milieu de la liste.
3. Pas de résultats à combiner puisque l’on ne “travaille” que sur l'une des deux sous-listes.

2. Paradigme “diviser pour régner“.

Troisième exemple : recherche d’un élément dans une liste triée

- Il est facile de voir que la recherche dichotomique est moins gourmande en nombre d’opérations que la recherche “classique“.

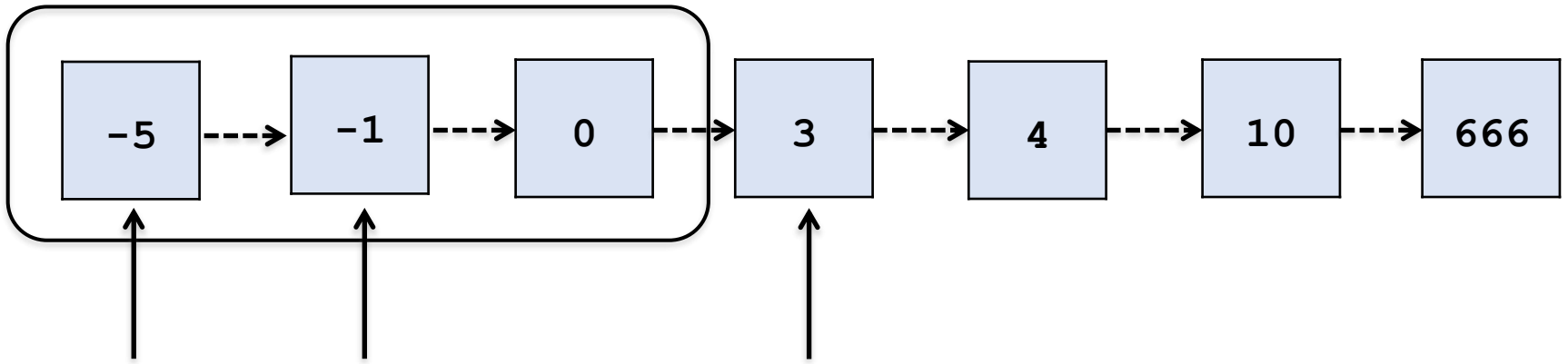
- Pour la première méthode, si un élément n’appartient pas à une liste de n éléments il faudra n comparaisons pour le détecter. En effet, tous les éléments de la liste seront testés un par un.

- Alors qu'avec une recherche dichotomique il faudra seulement effectuer environ $\log(n)$ comparaisons.

2. Paradigme “diviser pour régner“.

Troisième exemple : recherche d’un élément dans une liste triée

- Avec 3 comparaisons, on s’aperçoit que l’élément -4 n’appartient pas à cette liste :



2. Paradigme “diviser pour régner“.