Récursivité

Récursivité simple

Une fonction peut être appelée n'importe où dans un programme (après sa définition), y compris par elle-même.

Cours

Une fonction **récursive** est une fonction qui s'appelle elle-même[1].

Prenons pour exemple une fonction qui renvoie le produit de tous les nombres entiers entre 1 et . Ce produit est appelé factorielle de et noté .

Un programme itératif[2]: peut s'écrire simplement avec un boucle `for` qui multiplie tous les entiers allant de `1` à `n` entre eux :

```
def fact(n):
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f
```

Mais il est aussi possible de remarquer que et que , ce qui permet d'écrire un programme récursif suivant :

```
def fact(n):
    if n == 1:
        return 1
    return  fact(n-1) * n
```

On peut toujours transformer une fonction récursive en itérative et vice versa.

Importance de la clause d'arrêt

Cours

Une fonction récursive doit toujours comporter **une clause d'arrêt**, pour ne pas « boucler ».

La fonction suivante :

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

ne s'arrêtera jamais car il manque une clause d'arrêt `if n == 1 : …` !

: warning: Il faut toujours prendre soin de bien définir la clause d'arrêt. Ici que se passe-t-il si on appelle `fact(5.1)` ou `fact(-1)` ? On préfèrera peut-être `if n <= 1: …` ou utiliser des assertions pour éviter ces cas.

En pratique un appel récursif doit obligatoirement comporter une **instruction conditionnelle** et une **variable de contrôle** : par exemple un entier naturel qui décroît strictement à chaque appel récursif jusqu'à atteindre la valeur d'un cas de base.

Récursivité croisée, multiple

Cours

Dans certains cas, une fonction appelle une autre fonction qui elle-même appelle la première. C'est une **récursivité croisée**.

Par exemple pour tester si un nombre est pair ou impair (sans utiliser l'opérateur `%2`)

```
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)
```

Cours

Il arrive aussi qu'une fonction s'appelle plusieurs fois, c'est une **récursivité multiple**.

Par exemple la suite de Fibonacci est définie par , et pour .

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Si la récursivité est **plus élégante et facile** à lire qu'un programme itératif, on atteint très vite ses limites en complexités[3] spatiale et temporelle[4].

Complexité spatiale et pile d'exécution

Analysons ce qu'il se passe si on appelle la fonction récursive `fact(n)` quand `n` devient relativement grand.

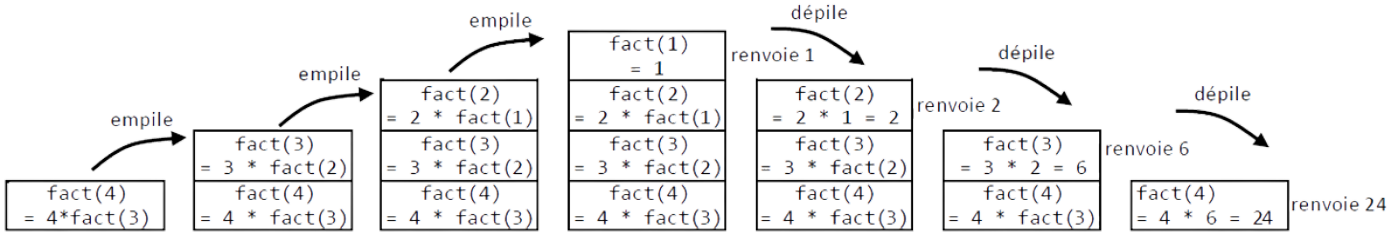
```
>>> fact(1000)
Traceback (most recent call last):
…
RecursionError: maximum recursion depth exceeded in comparison
```

Une erreur est levée , alors que le programme itératif fonctionne parfaitement !

Pour gérer des fonctions qui appellent d'autres fonctions, le système utilise une "pile d'exécution". Une pile d'exécution permet d'enregistrer des informations sur les fonctions en cours d'exécution dans un programme. C'est une pile, car les exécutions en attente "s'empilent" successivement les unes sur les autres.

La pile pour calculer `fact(4)` est la suivante :

- 1er appel de la fonction `fact` avec n = 4 : n n'est pas égal à 1, la fonction "empile" `fact(4)` et appelle `fact(3)`.
- 2ème appel de la fonction `fact` avec n = 3 : n n'est pas égal à 1, la fonction "empile" `fact(3)` et appelle `fact(2)`.
- 3ème appel de la fonction `fact` avec n = 2 : n n'est pas égal à 1, la fonction "empile" `fact(2)` et appelle `fact(1)`.
- 4ème appel de la fonction `fact` avec n = 1 : n est égal à 1, la fonction renvoie `1`.
- `fact(2)` est "dépilé", il renvoie 2 x 1 = 2.
- `fact(3)` est "dépilé", il renvoie 3 x 2 = 6.
- `fact(4)` est "dépilé" et peut enfin être calculé, il renvoie 4 x 6 = 26.



Cours

Complexité spatiale : La récursivité utilise une **pile d'appel** qui est un espace mémoire particulièrement limité, cela génère rapidement des **débordements de capacité**, c'est le fameux **stack overflow** !

Complexité temporelle

La suite de Fibonacci est définie par , et pour : .

Ce qui se traduit en une programmation itérative par :

```
def fib(n):
    if n < 2:
        return n
    else:
        a, b = 0, 1
        for i in range(2, n+1):
            a, b = b, a + b
    return b
```

Une programmation récursive est une simple traduction mot à mot de la définition :

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Plus facile à concevoir et à lire, la programmation récursive devient vite très lente à l'exécution. Comparons les temps d'exécution de ces deux programmes avec le module `time`.

```
from time import time

debut = time()
fib(30)
duree = time() - debut
print(duree, "secondes")
```

Les résultats obtenus sont les suivants (qui dépendent de la machine utilisée).

n	fib(n) itératif	fib(n) récursif
10	0.0 secondes	0.0 secondes
20	0.0 secondes	0.005 secondes
30	0.0 secondes	0.61 secondes
40	0.0 secondes	78.21 secondes
100	0.0 secondes	…

Alors que la fonction itérative est quasi instantanée pour toutes les valeurs de `n` testées, la fonction récursive devient très rapidement extrêmement lente, même pour des valeurs de `n` raisonnables, `fib(40)` demande plus d'une minute ! Ce n'est pas un problème de complexité spatiale, puisque `fib(40)` empile au plus 40 appels dans la pile d'appel, on est loin de la limite !

Regardons ce qui se passe pour calculer `fib(5)`. La fonction `fib` s'appelle 15 fois juste pour calculer `fib(5)` !

et on peut tout de suite imaginer que ce nombre de calculs augmente très rapidement pour `fib(30)` ou `fib(40)`. Ce n'est donc pas un problème de complexité spatiale mais plutôt un problème de nombre d'opérations effectuées pendant le calcul : c'est la **complexité temporelle**.

Cours

Complexité temporelle : La récursivité peut augmenter le nombre d'opérations.

Essayons d'en savoir plus sur le type de complexité de cette fonction en affichant le nombre d'appels de la fonction, en plaçant un compteur qui s'incrémente à chaque appel.

```
def fib(n):
    global cpt
    cpt += 1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

cpt = 0
fib(30)
print(cpt)
```

n	cpt
5	15
10	177
20	21 891
30	2 692 537
40	331 160 271

Il y a eu presque 3 millions d'appels alors que `fib(30)` ne nécessite, en théorie, que la connaissance de quelques dizaines de valeur ! Et `fib(40)` nécessite plus de 300 millions d'appels !

C'est une complexité qui semble de type exponentielle, en [5], c'est-à-dire qui ne peut pas être exécutée en temps acceptable pour n grand[6] !

Rappel

Principales complexités temporelles :

Désignation	Notation	Exemples
constante		Accès à un élément d'un tableau
logarithmique		Recherche dichotomique (tableau trié)
linéaire		Recherche dans un tableau
quasi linéaire		Tri fusion
quadratique		Tri à bulle, parcours de matrice
exponentielle		Sac à dos

Désignation	Notation	Exemples
factorielle		Voyageur de commerce

Mémoïsation

Nous avons vu qu'il y a eu presque 3 millions d'appels alors que `fib(30)` ne nécessite, en théorie, que la connaissance de quelques dizaines de valeur ! La fonction passe son temps à calculer des valeurs qu'elle a déjà calculées mais qu'elle n'a pas « notées ». Par exemple `fib(5)` calcule 3 fois la valeur de `fib(2)`. L'algorithme itératif n'a pas ce problème, il retient chaque valeur de la suite.

Une solution pour limiter le nombre de calcul consiste à ne calculer les termes de la suite qu'une seule fois et de les garder en mémoire. C'est la **memoïzation**.

Cours

La memoïsation consiste à garder en mémoire les valeurs déjà calculées.

Par exemple avec un dictionnaire déclaré en variable globale[7] :

```
memoise = {}
def fib(n):
    if n in memoise:
        return memoise[n]
    if n < 2:
        memoise[n] = n
    else :
        memoise [n] = fib(n-1) + fib(n-2)
    return memoise[n]
```

Ou plus court en renseignant les premières valeurs dans le dictionnaire:

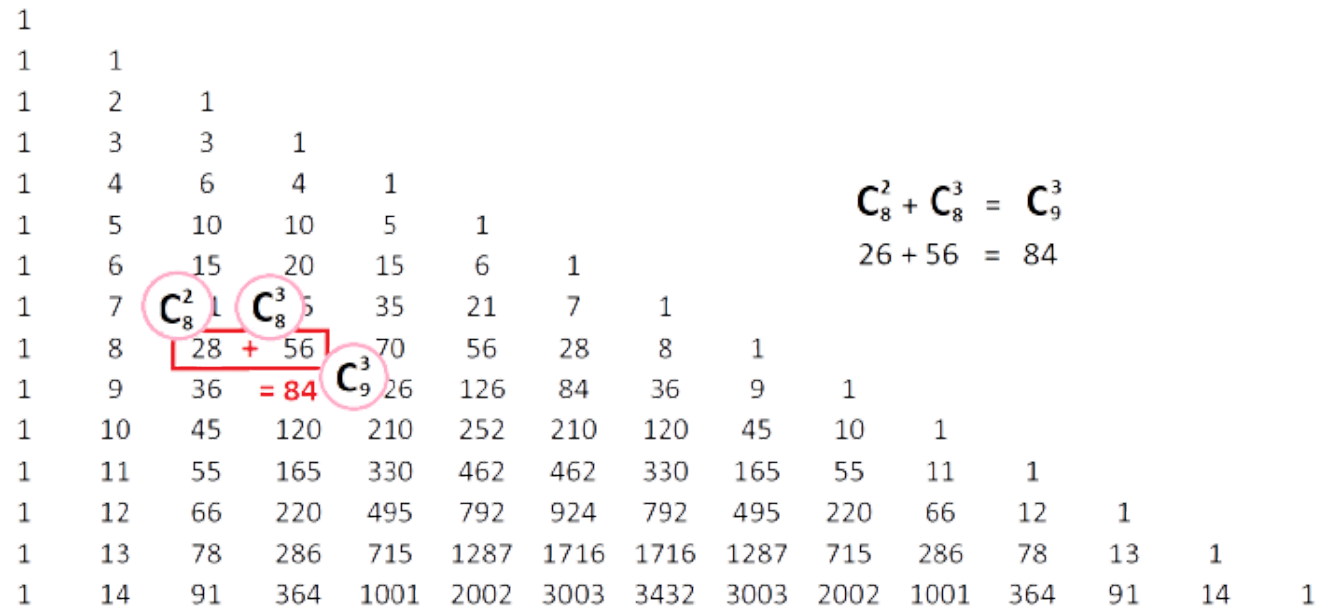
```
memoise = {0: 0, 1: 1 } # fib(0) et fib(1)
def fib(n):
    if n not in memoise:
        memoise[n] = fib(n-1) + fib(n-2)
    return memoise[n]
```

8

Exercice corrigé

En mathématiques, les coefficients binomiaux, définis pour tout entier naturel et tout entier naturel inférieur ou égal à , donnent le nombre de parties à éléments d'un ensemble à éléments. On les note ou . Les coefficients binomiaux interviennent dans de nombreux domaines des mathématiques : développement du binôme en algèbre, dénombrements, développement en série, lois de probabilités, etc.

1) Écrire une fonction récursive `C(n, k)` qui renvoie la valeur des coefficients binomiaux en utilisant la formule du triangle de Pascal :



2) Ajouter la mémoisation à la fonction `C(n, k)`.

Réponse

Note : La mémoisation est un exemple classique d'utilisation des décorateurs Python (hors programme)[9]. On pourra aussi explorer le décorateur @functools.lru_cache().

Complexité temporelle des fonctions récursives

Prenons l'exemple de la fonction récursive `fact`.

Si le calcul de `fact(n-1)` s'effectue en un nombre d'opérations connu, noté , alors le calcul de `fact(n)` s'effectue en effectuant cinq opérations élémentaires supplémentaires :

- une instruction conditionnelle (`if`),
- une comparaison (`n == 1`),
- un appel de fonction (`fact(n-1)`) ,
- une multiplication (`n * fact(n-1)`) et
- un `return`.

donc .

Ce qui peut s'écrire en ordre de grandeur : . La complexité de la fonction récursive `fact` est en .

Cours

La complexité d'une fonction récursive se calcule en trouvant une relation entre le nombre d'opérations d'un problème de taille et . Cette relation (de récurrence) permet de déduire .

Relation entre Tn et Tn-1	Complexité	Désignation
		Linéaire
		Quadratique
		Exponentielle

Comparons les complexités de la suite de Fibonacci avec nos trois programmes :

Programme itératif

```
def fib(n):
    if n < 2:                    # 1 condition + 1 comparaison
        return n
    else:
        a, b = 0, 1              # 2 affectations
        for i in range(2, n+1):  # n-1 affectations
            a, b = b, a + b      # n-1 affectations x 2
    return b                     # 1 return
```

Au total, il y a opérations élémentaires. Le coût est linéaire en .

Programme récursif

```
def fib(n):
    if n < 2:
        return n              # 1 condition + 1 comparaison
    return fib(n-1) + fib(n-2)  # 1 addition + Tn-1 + Tn-2
```

En faisant l'hypothèse que , nous obtenons , donc le cout est exponentiel en

Programme récursif avec mémoïsation

```
memoise = {}
def fib(n):
    if n in memoise:                # 1 condition + 1 recherche de clé dans un
```

```
    dictionnaire en O(1)
            return memoise[n]              # 1 accès au dictionnaire, au pire en O(n)
        if n < 2:                          # 1 condition + 1 comparaison
            memoise[n] = n
        else :
            memoise[n] = fib(n-1) + fib(n-2)     # 2 appels et 1 addition n fois -> O(n)
        return memoise[n]                   # 1 return + 1 un accès au dictionnaire (au pire en
    O(n))
```

Au total, le coût dans le pire des cas est en , c'est-à-dire un coût linéaire en .

---

1. Le mot "récursivité" en informatique a la même racine que "récurrence" utilisée pour les suites mathématiques. ↩
2. Une structure de contrôle est dite "itérative" qaund elle exécute plusieurs fois une séquence d'instructions (boucles `for`, `while`). ↩
3. Les mots "complexité" ou "coût" sont employés indifféremment. ↩
4. Attention à ne pas confondre les deux complexités : la complexité temporelle (ou en temps) mesure l'ordre de grandeur du nombre d'opérations élémentaires, la complexité spatiale (ou en espace) mesure l'espace mémoire requis par un programme. ↩
5. Plus exactement en ou . ↩
6. De manière générale, les algorithmes qui ont une complexité temporelle du type avec , dits de type NP, ne peuvent pas être exécutés en temps acceptable pour grand, contrairement à ceux en dits de type P qui peuvent l'être. ↩
7. Ici on peut modifier la variable globale `memoise` dans la fonction sans utiliser `global memoise` car elle est de type muable. ↩
8. Une technique pour ne pas garder de variable globale dans le reste programme consiste à transformer `fib(n)` en une fonction locale de `fibo(n)` :

```
def fibo(n):
    memoise = {}
    def fib(n):
        if n in memoise:
            return memoise[n] …
        if n < 2:
            memo_fib[n] = n
        else:
            memo_fib[n] = fib(n-1) + fib(n-2)
        return memo_fib[n]
    return fib(n)
```

↩

9.

```
def memoised(fonction):
    memoise = {}
    def fonction_memo(*args):
        if args not in memoise:
            memoise[args] = fonction(*args)
        return memoise[args]
    return fonction_memo

@memoised
def fib(n):
    if n < 2:
```

```
        return n
    return fib(n-1) + fib(n-2)
```

↵