Algorithmique avancée

Module : Algorithmes et structures de données

Douglas Teodoro

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

2019-2020

Objective

- Comprendre les principes de base de la récursivité
- Maîtriser l'écriture des algorithmes récursifs
- Maîtriser l'analyse des algorithmes récursifs classiques
- Analyser la complexité en utilisant des arbres d'appels

Sommaire

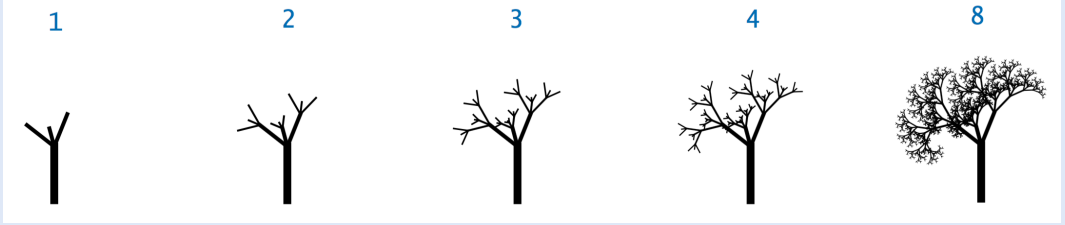
RÉCURSIVITÉ

Récursivité

Définition

La récursivité est la propriété que possède un objet ou un concept de s'exprimer en fonction de lui-même

Récursivité



Notion d'algorithme récursif

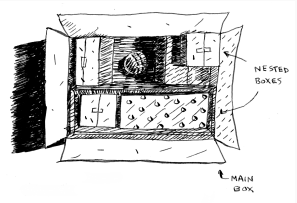
Reduction du problème

Avec la récursion, un problème est résolu en résolvant de plus petites instances du même problème

Recherche de fichiers

Parmi les exemples d'algorithmes s'exprimant simplement, on retrouve celui de la recherche de fichiers, présenté ici de manière informelle :

1. si le fichier à chercher est dans le répertoire actuel, on renvoie son chemin ;
2. sinon, on le cherche dans les sous-répertoires.



Récursivité

Définition

Fonction récursive : une fonction est récursive si son exécution peut conduire à sa propre invocation

- Concept important pour écrire des algorithmes, mais également pour décrire ou définir des structures de données

Algorithmes

- Fibonacci
- Factorielle
- Recherche dichotomique

Structures de données

- Liste
- Arbre
- Graphe

Fonction récursive

Pseudo-code

Algorithme : fonction récursive

Data : P : liste de paramètres

1 **Function** *f(P : params)*

    // instructions (1)

2   x = f(Q) // appel avec d'autres paramètres

    // instructions (2)

3   **return** resultat

Python

```python
def f(P):
    """
    Fonction récursive
    :param P: liste de params
    :return: resultat
    """
    # instructions (1)
    x = f(Q) # appel avec Q
    # instructions (2)
    return resultat
```

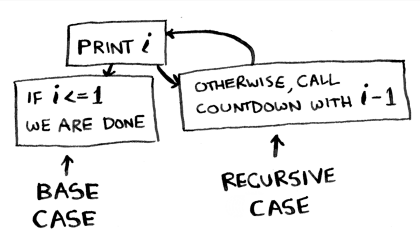
- aux constructions habituelles, on ajoute la possibilité d'appeler l'algorithme lui-même sur une autre donnée : (P→Q)
- le retour de valeur n'est pas obligatoire et dépendra du problème à résoudre

Principes de base

Pour que la récursion opère, **deux propriétés** doivent être satisfaites :

1. il faut au moins un **cas de base**, dans lequel la solution est directement calculée sans passer par la récursion

2. chaque **appel récursif** de la procédure doit se faire sur une instance plus petite du même problème, pour finir par atteindre un cas de base



Principes de base

Le **modèle** d'une fonction récursive correcte peut être résumé comme suit :

Algorithme : fonction récursive

Data : P : paramètre(s)

```
1 Function f(P : params)
2     if test(P) then          // condition d'arret
          // bloc sans appel récursif
3     else
          // bloc avec appel(s) récursif(s) sur
           des paramètres ``plus simples''
```

La signification de « plus simple » varie selon le contexte :

- si P est un **nombre naturel** et que la condition d'arrêt se base sur de petits nombres, alors $P' < P$
- si P est une **liste** et que la condition d'arrêt vérifie si la liste est vide, alors P' est une liste plus petite

Algorithmes récursifs basiques

Algorithmes récursifs basiques

Le cas le plus simple : on reçoit une définition déjà récursive, et il suffit alors de traduire cette définition dans un algorithme

Si ce n'est pas le cas :

1. on cherche d'abord une définition récursive
2. on s'assure que cette définition possède une ou plusieurs conditions d'arrêt
3. et après seulement, on écrit l'algorithme correspondant

La fonction factorielle récursive

Rappelons que le factoriel d'un nombre naturel n est $n!$:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1,$$

avec le cas particulier $0! = 1$

Exemple

$4! = 4 \times 3 \times 2 \times 1$
$4! = 24$

La fonction factorielle récursive

1. Trouver une formulation récursive : en écrivant la définition, on se rend compte que

$$n! = n \times \overbrace{(n-1) \times (n-2) \times \ldots \times 2 \times 1}^{=(n-1)!}$$

   On en déduit donc que $\mathbf{n! = n \times (n-1)!}$

2. Trouver une condition d'arrêt : elle nous est directement fournie dans la définition originale : si $n = 0$, alors le factoriel correspondant vaut 1
   On peut donc réécrire la définition de manière récursive :

$$n! = \begin{cases} 1, & \text{si } n \leq 1, \\ n \times (n-1)!, & \text{sinon.} \end{cases}$$

3. Traduire la définition en code : cette étape de traduction est la plus simple si la définition obtenue est suffisamment claire

La fonction factorielle récursive

Python

```python
def fact(n: int):
  """
  Calcule le factoriel de n
  :param n: int
  :return: int
  """
  if n <= 1:
    return 1
  else:
    return n * fact(n-1)
```

La fonction factorielle récursive

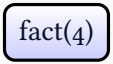
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

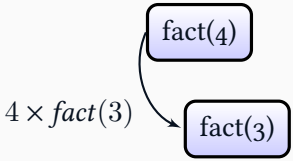
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

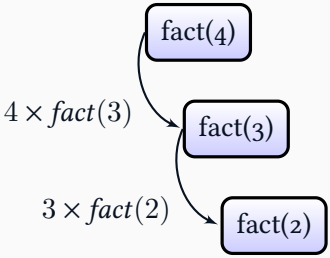
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

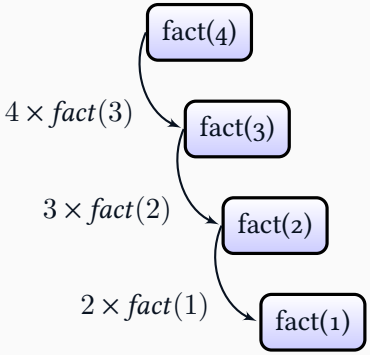
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

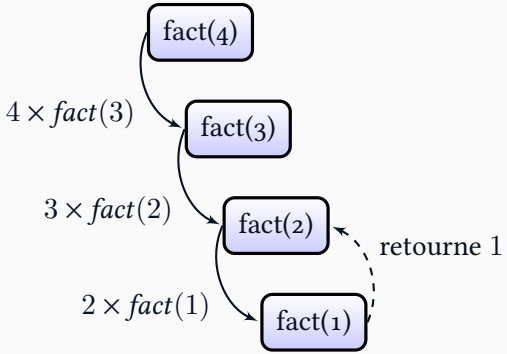
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

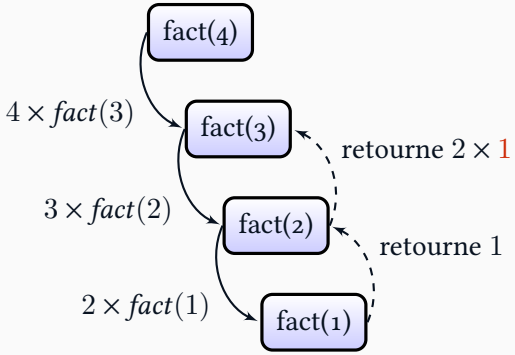
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

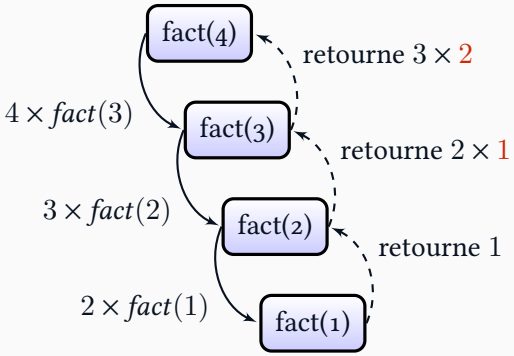
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive

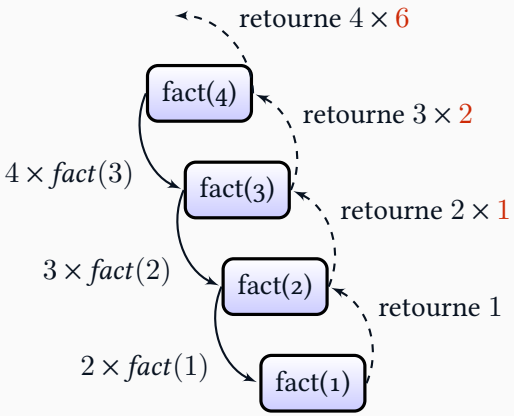
Python

```python
def fact(n: int):
    """
    Calcule le factoriel de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

```python
fact(4)
```

Exemple

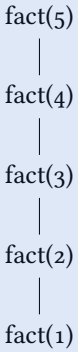
$4! = 4 \times 3 \times 2 \times 1$



La fonction factorielle récursive - Arbres d'appels

Les **arbres d'appels** permettent de visualiser ce qui se produit quand on appelle une fonction (récursive ou non)

Arbre d'appel pour `fact(5)`

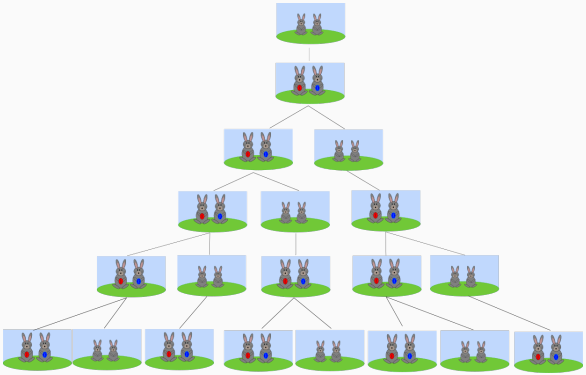


Les **arbres d'appels** aident aussi à évaluer la complexité de l'algorithme correspondant

- pour $n = 5$, on a donc 5 appels à effectuer
- comme chaque appel effectue 1 appel récursif, on se retrouve avec une complexité de $O(n)$

Suite de Fibonacci

« Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ? »

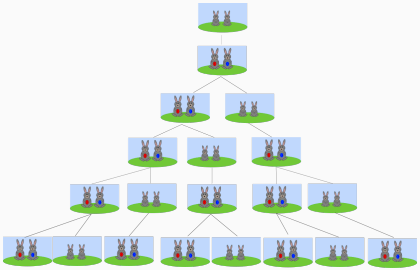


Suite de Fibonacci

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...
0	1	1	2	3	5	8	13	21	34	55	

Exemple

$F_2 = F_1 + F_0 = 1 + 0$
$F_3 = F_2 + F_1 = 1 + 1$
$F_4 = F_3 + F_2 = 2 + 1$
$\dots$



Suite de Fibonacci

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...
0	1	1	2	3	5	8	13	21	34	55	

Le n^{ème} nombre de Fibonacci F_n est donc donné par :

$$F_n = \begin{cases} n, & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2}, & \text{sinon.} \end{cases}$$

Suite de Fibonacci - Algorithme récursif

Il ne nous reste plus qu'à traduire cette définition en un algorithme :

1. Cas de base : `n <= 1`
2. Cas récursif : `fibo(n-1) + fibo(n-2)`

Pseudo-code

Algorithme : nombres de Fibonacci

Data : n : entier ≥ 0
Result : Fibonacci de n

```
1 Function fibo(n : int)
2     if n ≤ 1 then
3         return n
4     else
5         return fibo(n-1) + fibo(n-2)
```

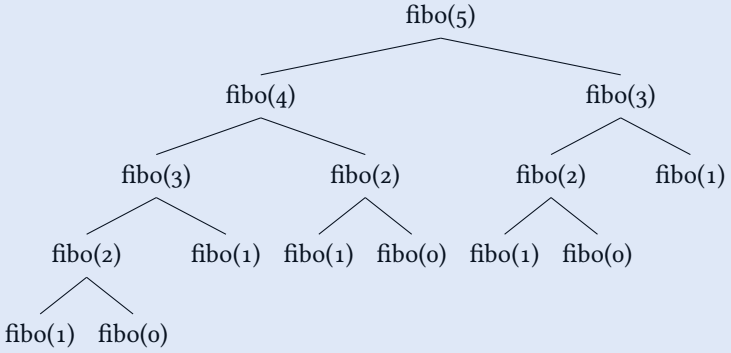
Python

```python
def fibo(n: int):
    """
    Fibonacci de n
    :param n: int
    :return: int
    """
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Suite de Fibonacci - Arbres d'appels

Si l'on exécute la fonction `fibo` avec la valeur 5 pour n, on obtient l'arbre d'appels :

Arbre d'appel pour `fibo(5)`



- pour $n = 5$, on a donc 15 appels à effectuer
- comme chaque appel effectue deux appels récursifs, on se retrouve avec une complexité de $O(2^n)$

Suite de Fibonacci - Arbres d'appels

DEMO - Arbre d'appels

PyCharm

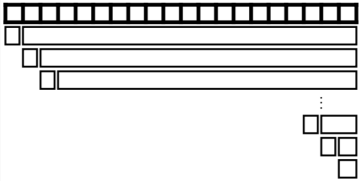
Exercice 1

Récursivité avec des structures de données

«Calculer la somme des éléments `E` d'une liste `L` »

La stratégie s'applique également ici :

1. Trouver une formulation récursive : on voit la liste `L` comme un élément `e` suivi d'une liste `L'`
2. Trouver une condition d'arrêt : elle nous est directement fournie dans la vision récursive d'une liste

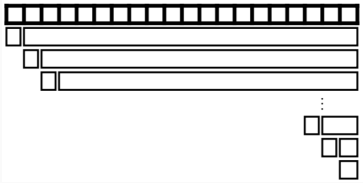


Récursivité avec des structures de données

On voit la liste `L` comme un élément `e` suivi d'une liste `L'`

Formulation récursive

- si `len(L') == 0`, on renvoie `e`
- si `len(L') != 0`, on renvoie le résultat de la somme de l'élément `e` plus la somme des éléments de `L'`



Condition d'arrêt

- soit la liste contient un seul élément
  → la somme est l'élément lui-même
- soit la liste est vide
  → la somme est zero

Somme récursive des éléments d'une liste

```python
def somme(a_liste: list):
    """
    Calcule la somme des éléments
    d'une liste
    :param a_liste: list
    :return: int ou float
    """
    if len(a_liste) == 0:
        return 0
    elif len(a_liste) == 1:
        return a_liste[0]
    else:
        return a_liste[0] + somme(a_liste[1:])
```

Complexité

Complexité

On a vu comment calculer la complexité des algorithmes itératifs

Les règles ne changent pas dans le cas des algorithmes récursifs :

- le nombre d'itérations est remplacé par le nombre d'appels récursifs

On a des coûts cachés liés à l'usage de la récursivité

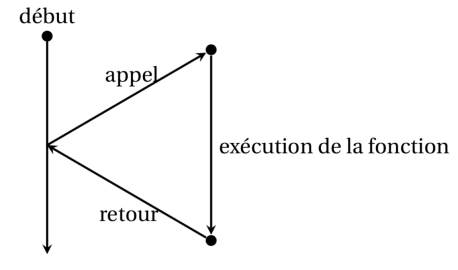
- à chaque appel de fonction, on doit sauvegarder le contexte, et on consomme donc de l'espace mémoire supplémentaire directement lié au nombre d'appels réalisés

Contexte et stack frames

Définition

Le contexte d'une fonction est l'ensemble des variables (et de leurs valeurs) qu'elle utilise

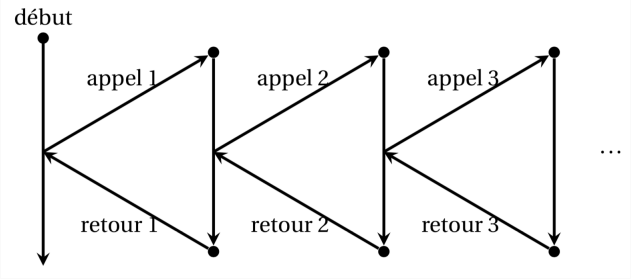
Lorsqu'une fonction est appelée, le déroulement du programme est interrompu :

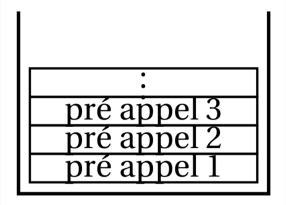


Il faut sauvegarder les données et l'état quelque part de manière à pouvoir les récupérer lorsque la fonction se termine

Contexte et stack frames

En particulier, **chaque appel récursif** nous obligera à sauvegarder le contexte





Impact sur la complexité spatiale

Si la fonction $\mathbf{f}(\cdot)$ utilise une nouvelle variable entière dans sa définition, qui coûte un espace constant, mais effectue n appels récursifs, alors sa complexité spatiale sera en $\mathcal{O}(n)$

Retour sur la factorielle

Algorithme : factorielle récursive

Data : n : entier ≥ 0
Result : factorielle de n

```
1 Function fact(n : int)
2     if n == 0 then                    // c2
3         return 1                      // c3
4     else
5         return n * fact(n − 1)  // c5 + T_fact(n-1)
```

D'où $T(n) =$

$$\begin{cases} T_{fact}(0) & = c_2 + c_3, \\ T_{fact}(n) & = c_2 + c_5 + T_{fact}(n-1) \end{cases}$$

Coût arithmétique

Si $T(n) = a + T(n–1)$
alors $T(n) = a \times n + T(0) = O(n)$

La fonction de coût d'un algorithme récursif obéit généralement elle-même à une équation récursive

Limitations pratiques

Python limite le nombre d'appels récursifs que l'on peut effectuer :

```
>>> factoR(997)
# ok, le résultat s'affiche
>>> factoR(998)
RuntimeError: maximum recursion depth exceeded in comparison
```

Le message ci-dessus signifie que l'on a effectué trop d'appels récursifs :

- soit parce que le code est correct mais qu'on l'a exécuté sur quelque chose de trop grand pour arriver au bout des appels
- soit parce que le code est erroné

Si l'on voit ce message même pour des données de petite taille, cela veut généralement dire qu'on a oublié une condition d'arrêt ou qu'un des appels récursifs est incorrect

PyCharm

Exercice 2

Sommaire

COMPARAISON DES APPROACHES INTERACTIVES ET RÉCURSIVES

Fibonacci - Récursive

1. La définition des nombres de Fibonacci est récursive
2. La condition d'arrêt ($n \leq 1$) est également fournit

Algorithme : fibo_recursif

Data : n : entier ≥ 0
Result : fibonacci de n

```
1 Function fibo(n : int)
2     if n ≤ 1 then
3         return n
4     else
5         return fibo(n-1) + fibo(n-2)
```

D'où $T(n) =$
$$\begin{cases} T_{fibo}(0) & = c_2 + c_3, \\ T_{fibo}(n) & = c_2 + c_5 + T_{fibo}(n-1) + T_{fibo}(n-2) \end{cases}$$

Complexité (en temps)

Si $T(n) = a + T(n-1) + T(n-2)$
alors
$T(n) \leq 2T(n-1) - T(n-3) \leq c2^n = O(2^n)$

Fibonacci - Interactive I

Une version de Fibonacci interactive :

Algorithme : fibo_interactive_1

Data : n : entier ≥ 0
Result : fibonacci de n

```
1 Function fibo(n : int)
2     L = [0,1] // initialise F_0 et F_1
3     if n > 1 then
4         for i ← 2 to n do
5             L[i+1] = L[i-1] + L[i]
                // calcule F_n en utilisant
                F_{n-1} et F_{n-2}
6     return L[n+1]
```

Complexité

En temps
$T(n) = c_2 + c_3 + c_4 n + c_5(n-1) + c_6 = O(n)$

En space : $S(n) = O(n)$

Mais, on peut alors améliorer cet algorithme car à chaque étape, seuls les deux derniers termes de la suite sont nécessaires ...

Fibonacci - Interactive II

Une deuxième version de la Fibonacci interactive :

Algorithme : fibo_interactive_2

Data : n : entier ≥ 0
Result : fibonacci de n

```
1 Function fibo(n : int)
2     if n ≤ 1 then
3         return n
4     L = [0,1]
5     for i ← 2 to n do
6         tmp = L[1]
7         L[1] = L[2] // mettre à jour
            F_{n-1}
8         L[2] = L[2] + tmp
            // calculer F_n en utilisant
            F_{n-1} et F_{n-2}
9     return L[2]
```

Complexité

En temps : $T(n) = O(n)$
En space : $S(n) = O(1)$

Vérifions la correction de cet algorithme :

- si $n \leq 1$, l'algorithme renvoie le nombre n ce qui est correct
- sinon, l'algorithme crée une liste de deux élément dont la première valeur est F_{i-1} et la seconde F_i
- au pas $i + 1$, l'algorithme crée une nouvelle liste dont le premier élément est F_i et le second $F_{i+1} = F_i + F_{i-1}$

Implementation Python

Fibonacci Récursive

```python
def fibo(n: int):
  assert n >= 0
  if n <= 1:
    return n
  else:
    return fibo(n-1) + fibo(n-2)
```

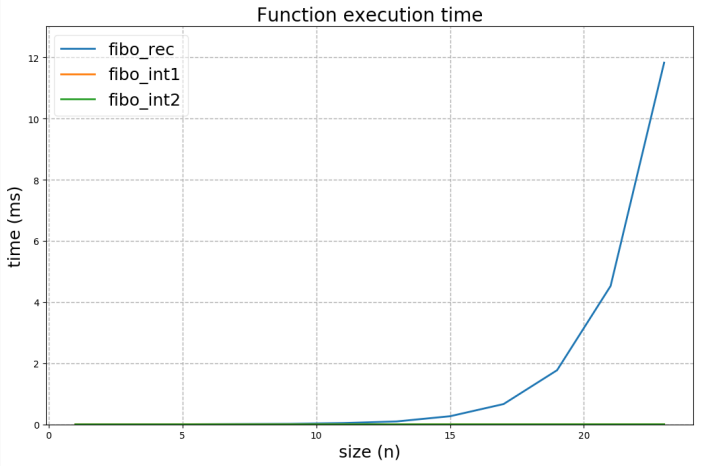
Fibonacci Interactive I

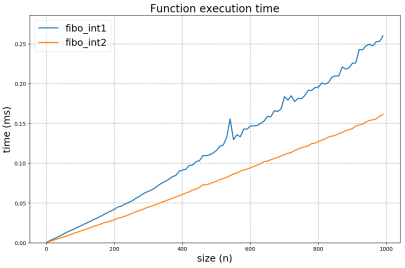
```python
def fibo(n: int):
  assert n >= 0
  L: list = [0,1]
  if n > 1:
    for i in range(2,n+1):
      L.append(L[i-1]+L[i-2])
  return L[n]
```

Fibonacci Interactive II

```python
def fibo(n: int):
  assert n >= 0
  L: list = [0,1]
  if n <= 1:
    return L[n]
  for i in range(2, n+1):
    tmp = L[0]
    L[0] = L[1]
    L[1] = L[1] + tmp
  return L[1]
```

Temps d'exécution





$T_{fibo\_rec}(n) = O(2^n)$
$fibo\_rec(20) \approx 1M$ d'appels

PyCharm

Exercice 3

Challenge !

Sommaire

Conclusion

- Avantages :
  1. Algorithmes concis et faciles à prouver
  2. Le raisonnement et l'écriture de code peut être plus simple
- Inconvénient : une appel récursif est assez coûteux (temps & espace)

On a souvent avantage à implémenter des versions non-récursives

La prochaine fois

- Preuves et corrections
- Algorithmes des tris

Référence

Algorithmes Notions de base : Pages 23 - 25 Cormen, http://hesge.scholarvox.com

Cyberlearn : 19_HES-SO-GE_633-1 ALGORITHMES ET STRUCTURES DES DONNÉES

http://cyberlearn.hes-so.ch

Extra

Contre-exemple I

Voici deux fonctions récursifs visant à réaliser la même tâche : afficher `****...****!`

Algorithme : f_affiche_1

Data : n : entier ≥ 0

```
1 Function f_affiche_1(n : int)
2     if n == 0 then
3         afficher `!'
4     else
5         afficher `*'
6         f(n)
```

Algorithme : f_affiche_2

Data : n : entier ≥ 0

```
1 Function f_affiche_2(n : int)
2     if n == 0 then
3         afficher `!'
4     else
5         afficher `*'
6         f(n - 1)
```

Question : Quel est le problème avec la mauvaise fonction `f_affiche_1` ?

Contre-exemple II

Voici une autre fonction visant à réaliser la même tâche que la fonction factorielle récursive :

$$\mathbf{n!} = (\mathbf{n+1})!/(\mathbf{n+1})$$

Algorithme : fact_rec_2

Data : n : int ≥ 0

```
1 Function fact_rec_2(n : int)
2     if n == 0 then
3         return 1
4     else
5         return fact_rec_2(n+1)/(n+1)
```

Bonne mathématique

$$\begin{aligned} 4! &= (4+1)!/(4+1) \\ &= 5 \times 4 \times 3 \times 2 \times 1/5 \\ &= 4 \times 3 \times 2 \times 1 \end{aligned}$$

Question : Quel est le problème avec la fonction `fact_rec_2` ?

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ?

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ? 1
2. `mystery(2,4)` ?

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ? 1
2. `mystery(2,4)` ? 16
3. `mystery(3,3)` ?

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ? 1
2. `mystery(2,4)` ? 16
3. `mystery(3,3)` ? 27

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ? 1
2. `mystery(2,4)` ? 16
3. `mystery(3,3)` ? 27

Question 2 : Que calcule la fonction ?

Exercice I

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if b == 0:
        return 1
    else:
        return a * mystery(a, b-1)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(2,0)` ? 1
2. `mystery(2,4)` ? 16
3. `mystery(3,3)` ? 27

Question 2 : Que calcule la fonction ? `exponentielle(a,b)`

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ?

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ? 2
2. `mystery(5,4)` ?

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ? 2
2. `mystery(5,4)` ? 1
3. `mystery(22,11)` ?

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ? 2
2. `mystery(5,4)` ? 1
3. `mystery(22,11)` ? 11

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ? 2
2. `mystery(5,4)` ? 1
3. `mystery(22,11)` ? 11

Question 2 : Qu'est-ce que la fonction calcule ?

Exercice II

Étant donné la fonction récursive :

```python
def mystery(a: int, b: int):
    """
    fonction mystère
    :param a: int
    :param b: int
    :return: la valeur mystère
    """
    if a == b:
        return a
    elif a > b:
        return mystery(a-b, b)
    else:
        return mystery(b, a)
```

Question 1 : Quelle est la valeur renvoyée par :

1. `mystery(6,4)` ? 2
2. `mystery(5,4)` ? 1
3. `mystery(22,11)` ? 11

Question 2 : Qu'est-ce que la fonction calcule ? `pgcd(a,b)`

Exercice III - Temps de exécution

Question : Quel est le temps d'exécution du algorithme de Fibonacci récursif pour calculer le Fibonacci de 50 en utilisant un processeur qui exécute 200'000MIPS ?

A) quelques seconds

B) quelques minutes

C) quelques heures

D) quelques jours

Exercice III - Temps de exécution

Question : Quel est le temps d'exécution du algorithme de Fibonacci récursif pour calculer le Fibonacci de 50 en utilisant un processeur qui exécute 200'000MIPS ?

A) quelques seconds

B) quelques minutes

C) quelques heures

D) quelques jours

Exercice III - Temps de exécution

Question : Quel est le temps d'exécution du algorithme de Fibonacci récursif pour calculer le Fibonacci de 50 en utilisant un processeur qui exécute 200'000MIPS ?

A) quelques seconds

B) quelques minutes

C) quelques heures

D) quelques jours

$$\texttt{temps d'exécution} = \frac{2^{50}}{200000 \times 10^6} \approx 5600 \text{ sec}$$

Exercice IV - Temps d'exécution

Question : Étant donné deux algorithmes de tri : 1) un récursif avec un temps d'exécution dans l'ordre de $O(n \log_2 n)$ et 2) un interactive avec un temps d'exécution dans l'ordre de $O(n^2)$, on demande quel algorithme sera plus efficient pour trier un tableau de 16 éléments ?

A) récursif

B) interactive

Exercice IV - Temps d'exécution

Question : Étant donné deux algorithmes de tri : 1) un récursif avec un temps d'exécution dans l'ordre de $O(n \log_2 n)$ et 2) un interactive avec un temps d'exécution dans l'ordre de $O(n^2)$, on demande quel algorithme sera plus efficient pour trier un tableau de 16 éléments ?

A) récursif

B) interactive

Exercice IV - Temps d'exécution

Question : Étant donné deux algorithmes de tri : 1) un récursif avec un temps d'exécution dans l'ordre de $O(n \log_2 n)$ et 2) un interactive avec un temps d'exécution dans l'ordre de $O(n^2)$, on demande quel algorithme sera plus efficient pour trier un tableau de 16 éléments ?

A) récursif

B) interactive

`récursif =`

$$T(16) = 16 \log_2 16 = 64$$

`interactive =`

$$T(16) = 16^2 = 256$$

Exercice V - Temps d'exécution

Question : Quels sont les temps d'exécution pour les trois implementation Python suivant ?

Puissance I

```
def pow(y, x):
  if y == 0:
    return 0
  elif x == 0:
    return 1
  else:
    return y*pow(y,x-1)
```

Puissance II

```
def pow(y, x):
  res = y
  if x == 0:
    res = 1
  elif y != 0:
    while x > 1:
      res = res*y
      x = x-1
  return res
```

Puissance III

```
def pow(y, x):
  if x == 0:
    return 1
  elif x == 1:
    return y
  elif x%2 == 0:
    return pow(y*y,x/2)
  else:
    return y*pow(y*y,(x-1)/2)
```

A) $O(n), O(n), O(\log_2 n)$

B) $O(n^2), O(n), O(n^2)$

C) $O(n), O(n), O(n^2)$

D) $O(n^2), O(n), O(n \log_2 n^2)$

Exercice V - Temps d'exécution

Question : Quels sont les temps d'exécution pour les trois implementation Python suivant ?

Puissance I

```
def pow(y, x):
  if y == 0:
    return 0
  elif x == 0:
    return 1
  else:
    return y*pow(y,x-1)
```

Puissance II

```
def pow(y, x):
  res = y
  if x == 0:
    res = 1
  elif y != 0:
    while x > 1:
      res = res*y
      x = x-1
  return res
```

Puissance III

```
def pow(y, x):
  if x == 0:
    return 1
  elif x == 1:
    return y
  elif x%2 == 0:
    return pow(y*y,x/2)
  else:
    return y*pow(y*y,(x-1)/2)
```

A) $O(n), O(n), O(\log_2 n)$

B) $O(n^2), O(n), O(n^2)$

C) $O(n), O(n), O(n^2)$

D) $O(n^2), O(n), O(n \log_2 n^2)$

Exercice V - Temps d'exécution