Lycée Loges

Chapitre 7

Récursivité
Exercices

Simon Dauguet
simon.dauguet@gmail.com

30 janvier 2025

Exercice 1 :
Le mathématicien perse Al-Khwârizmî (780-850), qui donna le terme algorithme, possède 27 pièces d'or. Un fieffé voleur dérobe une pièce et la remplace par une fausse pièce plus légère. Al-Khwârizmî possède une balance à fléau (balance qui précise seulement lequel des deux plateaux est le plus lourd). Combien de pesés doit-il faire au minimum pour déterminer à coup sûr la fausse pièce ?

Exercice 2 :
De manière rudimentaire si $(x, y) \in \mathbb{Z} \times \mathbb{N}$, le calcul de la somme $x + y$ se fait également par la relation : $s(x, y) = s(x + 1, y - 1)$ et $s(x, 0) = x$.

Proposer une fonction récursive qui calcule la somme $x + y$.

Exercice 3 :
On cherche à calculer la somme des valeurs d'un tableau d'entiers `tab` en remarquant que `somme(tab) = tab[0] + somme( tab[1:] )`.

1. Préciser l'état de la pile d'exécution pour `tab=[3, 7, 21]`
2. Écrire la fonction récursive `somme(tab: list) -> int`, qui répond à la question et étudier sa complexité temporelle.
3. Déterminer le variant pour montrer la terminaison, puis l'invariant pour montrer la correction de la fonction `somme()`.

Exercice 4 :
On cherche à définir une fonction récursive `rechercheTab(tab: list, v: int) -> bool`, qui renvoie `True` si v est une valeur de `tab`, `False` sinon.

1. Préciser l'état de la pile d'exécution pour `tab=[3, 7, 21]` et `v=7`.
2. Écrire la fonction récursive `rechercheTab(tab: list, v: int) -> bool`, qui répond à la question et étudier sa complexité temporelle.
3. Déterminer le variant pour montrer la terminaison, puis l'invariant pour montrer la correction de la fonction `rechercheTab()`.

4. Même exercice, cette fois-ci avec une fonction **`rechercheTab(tab: list, v: int, i0: int=0) -> bool`**, qui recherche la valeur `v` à partir de l'indice `i0`.

Exercice 5 :

Si `tab` est trié, une fonction de recherche dichotomique récursive **`rechercherDicho(tab: list, v: int) -> bool`**, qui renvoie `True` si v est une valeur de `tab`, `False` sinon, peut se définit ainsi :

- la fonction renvoie `True` si le tableau est de longueur 1 et v est la valeur du tableau, `False` si vide.
- sinon la fonction renvoie le résultat d'elle-même appliquée au demi-tableau de gauche ou le résultat d'elle-même appliquée au demi-tableau de droite

1. Préciser l'état de la pile d'exécution pour **`tab=[1,1,4,5,6,6,8,9]`** et `v=7`.
2. Implémenter la fonction **`rechercherDicho(tab:list, v:int) -> bool`**.
3. Déterminer le variant pour montrer la terminaison, puis l'invariant pour montrer la correction de la fonction **`rechercheDicho()`**.

Exercice 6 :

Déterminer ce que font chacune des fonctions suivantes,

```
1  from math import sqrt
2  def mystere1(m:int) -> int:      # 0 <= m
3      if m == 2:
4          return(True)
5      elif m == 1 or m % 2 == 0:
6          return(False)
7      return(mystereR1(m, 3))
8  def mystereR1(m:int, n:int) -> int:
9      if n > sqrt(m):
10         return(True)
11     elif m % n == 0:
12         return(False)
13     return(mystereR1(m, n+2))
14
15 def mystere2(n:int) -> str: # 0 <= n
16     if n == 0:
17         return(0)
18     mystere3(n//2)
19     print(n % 2 , end='')   # end='' permet de ne pas aller a la ligne a la fin du print
```

Exercice 7 :

On souhaite déterminer par dichotomie et de manière récursive l'approximation à ε près du zéro dans $[a, b]$ d'une fonction f croissante de sorte que $f(a)f(b) < 0$. Ainsi si $|b-a| > \varepsilon$, on calcule c milieu de $[a, b]$ et selon le signe de $f(a)f(c)$ on cherche un zéro de f dans $[a, c]$ ou $[c, b]$. Si $|b-a| \leq \varepsilon$, on renvoie la valeur de c.

1. Créer la fonction **`solutionDichoRec(f:'function', a:float, b:float, epsilon:float) -> float`**.
2. Préciser la terminaison et l'hérédité de votre fonction.
3. Faut-il modifier **`solutionDichoRec()`** si f est décroissante ? Si oui, quoi ?

Exercice 8 :

On souhaite calculer le pgcd de deux nombres entiers p et q non nuls à l'aide de l'algorithme d'Euclide.
On rappelle ci-contre les étapes de cet algorithme pour trouver : $\text{pgcd}(10, 24) = 2$.

p	q	r
24	10	4
10	4	2
4	2	0

1. Appliquer l'algorithme d'Euclide "à la main" pour trouver le pgcd de 35 et 126,
2. En utilisant le reste de la division euclidienne, proposer une fonction itérative `pgcdI(p:int, q:int) -> int`
3. En n'utilisant cette fois que les propriétés suivantes ($\text{pgcd}(p, q) = \text{pgcd}(q, p)$; $\text{pgcd}(p, q) = \text{pgcd}(p - q, q)$ et $\text{pgcd}(p, 0) = p$), proposer une fonction récursive `pgcdR(p:int, q:int) -> int`.

Exercice 9 :
On souhaite calculer les termes de la suite de Fibonacci. On rappelle qu'elle est définie par $f_0 = 0$, $f_1 = 1$ et la relation $\forall n \in \mathbb{N}^*,\ f_{n+1} = f_n + f_{n-1}$.

1. Proposer une fonction itérative `fiboI(n:int) -> int`, qui permette de calculer f_n,
2. Proposer une fonction récursive `fiboR(n:int) -> int`, qui permette de calculer f_n,
3. Préciser la relation donnant le coût temporel (puis le O correspondant) pour chacune des fonctions.

Exercice 10 (Exponentiation rapide) :
*sur tout l'exercice on n'utilisera pas l'opérateur puissance ('**')*
On cherche à calculer de manière efficace x^n pour $n \in \mathbb{N}$.

1. Puisque $x^n = x \times x^{n-1}$, proposer une fonction récursive naïve `puissanceRN(x:float, n:int) -> float`, qui calcule x^n.
2. On peut remarquer que lorsque
   - n est pair (*i.e.* $\exists k \in \mathbb{N},\ n = 2k$), on a $x^n = (x^2)^2$.
   - n est impair (*i.e.* $\exists k \in \mathbb{N},\ n = 2k + 1$), on a $x^n = x(x^2)^k$.

   On se propose de mettre en place une fonction récursive `puissanceR(x:float, n:int) -> float` qui utilise ces remarques.

   Dérouler un jeu de test de l'appel `puissanceR(2, 11)`.
3. Proposer une fonction `puissanceR(x:float, n:int) -> float`, qui mette en œuvre ces relations.
4. Préciser les coûts temporels des fonctions précédentes.

Exercice 11 :
On considère un jeu où l'utilisateur doit traverser une salle rectangulaire de l'entrée (en haut à gauche) à la sortie (en bas à droite) en maximisant le chemin utilisé (chaque case traversée ajoute la valeur de sa case au poids du chemin). Il ne peut se déplacer que vers le bas ou vers la droite.

→

3	1	0	1	0
2	0	4	1	1
1	2	0	2	3
2	0	2	4	0

→

1. Quel est le poids du chemin si l'utilisateur se déplace sur la droite tant qu'il peut puis descend jusqu'à la case de sortie ?
2. Avec les déplacements autorisés, montrer que le nombre de chemins possibles pour un rectangle (n, m) est $\binom{n+m-2}{n-1}$.
3. On souhaite connaître le chemin de poids maximal. On propose de faire un parcours récursif, qui, naïvement, va effectuer tous les chemins possibles et garder celui de poids maximal.

   Proposer une fonction récursive `parcours(grille: list, pos: tuple) -> tuple`, qui renvoie un tuple contenant le poids maximal pour atteindre la position `pos=(i,j)` ainsi que le tableau des positions traversées pour atteindre ce poids maximal.

Ainsi si le paramètre `grille` représente le tableau d'exemple, `parcours(grille, (1,1))` renverra le tuple `(5,[(0,0),(1,0),(1,1)])` tandis que `parcours(grille, (2,0))` renverra le tuple `(6,[(0,0),(1,0),(2,0)])`.

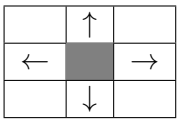
4. Quelle est la complexité de cette solution récursive ?
5. On souhaite améliorer cette solution avec une mémoïsation des meilleures chemins. Pour cela, on compte définir un tableau `route` de même taille que `grille`, qui contient dans chaque cellule un tuple `(poids, case précédente)`, où le poids correspond à la somme du poids de la case précédente et de la case actuelle, et la case précédente est choisie pour maximiser le poids. Autrement dit, entre la case à gauche et la case au dessus, on choisie la case dont la somme des deux poids est maximale.
   Compléter le tableau `route` pour l'exemple introductif

(3,None)	(4,(0,0))	(1,(0,1))	...	...
(5,(0,0))	(2,(1,0))	...		
...				
...				

6. Proposer une procédure récursive `mRoute(grille: list) -> list`, qui remplie le tableau `route` donnant le meilleur chemin.
7. En déduire une fonction `mParcours(grille: list, pos:tuple) -> tuple`, qui renvoie un tuple contenant le poids maximal, ainsi que le chemin correspondant, permettant d'atteindre la position `pos` de la grille.

Exercice 12 (Tétris unicouleur (*X-ENS 2019 partie IV*)) :

Dans cette partie, on considère une variante du jeu où le but du joueur est de former des régions unicolore au lieu d'alignements (par exemple un carré de 2×2 cases de la même couleur). Une région unicolore est un ensemble de cases, toutes de la même couleur, qui est connexe pour la relation d'adjacence suivante : deux cases sont adjacentes si elles partagent un côté. La région grisée est une région unicolore maximale : si on y ajoute une case quelconque, elle ne reste plus connexe et unicolore. En particulier, la case supérieure droite ne peut être ajoutée car elle n'est adjacente à aucune case de la région grisée.



voisins d'une case

					R
R	R			R	N
V	R	R	R	R	J
J	V	R	J	R	J
N	V	R	R	R	R

région unicolore maximale

1. Écrire une fonction récursive `tailleRegionUnicolore(grille,x,y)` qui renvoie le nombre de cases appartenant à la plus grande région unicolore de la grille contenant la case (x,y). Justifier la terminaison de votre fonction.

Considérons le code Python suivant, dont le but est de réaliser le même travail que la fonction `tailleRegionUnicolore` sans utiliser la récursivité.

Intuitivement, la fonction `exploreRegion(grille,x,y)` effectue un balayage de la grille, d'abord verticalement à partir de la case (x,y), puis verticalement à partir de chaque voisine horizontale des cases déjà explorées.

```python
def xDansGrille(g: list, x: int)->bool:
    return 0 <= x and x < len(g)

def yDansGrille(g: list, y: int)->bool:
    return 0 <= y and y < len(g[0])

def exploreVertical(grille, x, y, dir)->int:
    hauteur = len( grille[0] )
    couleur = grille[x][y]
    v = y + dir
    while yDansGrille(grille, v) :
        if grille[x][v] != couleur :
            return v - dir
        v = v + dir
    if dir == 1 : return hauteur - 1
    else : return 0

def exploreRegion(grille, x, y) -> int:
    inf = exploreVertical(grille, x, y, -1)    # explore vers le bas
    sup = exploreVertical(grille, x, y, 1)     # explore vers le haut
    d = exploreHorizontal(grille, x, y, 1)     # explore vers la droite
    g = exploreHorizontal(grille, x, y, -1)    # explore vers la gauche
    score = sup - inf + 1 + d + g
    return score
```

2. Déterminer si la fonction **`exploreRegion(grille,x,y)`** renvoie le nombre de cases appartenant à la plus grande région unicolore de la grille contenant la case (x,y). Si oui, justifier soigneusement que la fonction renvoie toujours une valeur correcte. Si non, donner un exemple de paramètres **`grille, x, y`** pour lesquels la valeur renvoyée par la fonction est incorrecte (on pourra dessiner la grille).