Initiation à l'algorithmique et programmation

L3 2025-2026

Travaux Pratiques 2 – Solutions

Exercice 1. *Somme des Éléments d'une Liste*
Écrivez une fonction récursive `somme_liste` qui calcule la somme des éléments d'une liste d'entiers.

SOLUTION

```
def somme_liste(lst):
    if not lst:
        return 0
    else:
        return lst[0] + somme_liste(lst[1:])
```

Exercice 2. *Imprimer des nombres*

1. Ecrivez une fonction utilisant la récursivité pour imprimer des nombres compris entre `n` et 0.
2. Ecrivez une fonction utilisant la récursivité pour imprimer les nombres de 0 à `n`. (Il vous suffit de changer une ligne dans le programme) du problème 1).

SOLUTION

```
def imprimer_desc(n):
    if n < 0:
        return
    print(n)
    imprimer_desc(n - 1)

def imprimer_asc(n, start=0):
    if start > n:
        return
    imprimer_asc(n, start + 1)
    print(start)
```

Exercice 3. *Longueur d'une Chaîne*
Créez une fonction récursive `longueur_chaine` qui retourne la longueur d'une chaîne de caractères sans utiliser la fonction intégrée `len()`.

SOLUTION

```
def longueur_chaine(chaine):
    if chaine == '':
        return 0
    else:
        return 1 + longueur_chaine(chaine[1:])
```

Exercice 4. *Inversion d'une Chaîne*

Implémentez une fonction récursive `inverse_chaine` qui prend en entrée une chaîne de caractères et renvoie la chaîne inversée.

SOLUTION

```
def inverse_chaine(chaine):
    if len(chaine) <= 1:
        return chaine
    else:
        return inverse_chaine(chaine[1:]) + chaine[0]
```

Exercice 5. *Parenthèses Équilibrées*

Une chaîne de caractères contenant uniquement des parenthèses "(" et ")" est dite *équilibrée* si chaque parenthèse ouvrante "(" a une parenthèse fermante correspondante ")" qui apparaît après elle, et si chaque paire de parenthèses est correctement imbriquée et appariée.

Écrire une fonction récursive `parentheses_equilibrees` qui vérifie si les parenthèses dans une chaîne de caractères sont équilibrées.

Exemples :

```
> > > parentheses_equilibrees("")
True
> > > parentheses_equilibrees("(())")
True
> > > parentheses_equilibrees("(()")
False
> > > parentheses_equilibrees("((()))")
True
```

SOLUTION

```
def parentheses_equilibrees(chaine, index=0, compteur=0):
    if index == len(chaine):
        return compteur == 0
    if compteur < 0:
        return False
    if chaine[index] == '(':
        return parentheses_equilibrees(chaine, index + 1, compteur + 1)
    elif chaine[index] == ')':
        return parentheses_equilibrees(chaine, index + 1, compteur - 1)
    else:
        return parentheses_equilibrees(chaine, index + 1, compteur)
```

Exercice 6. *Sac à dos*

Le problème du sac à dos consiste à essayer de remplir complètement un sac à dos ayant une capacité maximale de `S` kg avec des objets de différents poids non-nuls donnés dans un ensemble `E`.

On représente l'ensemble `E` par une liste d'entiers distincts. Le but est d'écrire une fonction qui détermine s'il existe une solution `F` ou non. Un algorithme possible est le suivant :

- Si $S = 0$ alors il existe une solution ;
- Sinon, pour chaque entier k dans E :
  - s'il existe une solution pour $S - k$ n'utilisant pas k, alors il existe une solution pour s ;
  - sinon, on cherche une solution pour S n'utilisant pas l'entier k.

Questions :

1. Détailler le déroulement de l'algorithme ci-dessus pour $S = 8$ et $E = \{2, 4, 5, 8, 6, 1, 9\}$.

2. Écrire une fonction `sac_a_dos_booleen(somme, entiers, index)` qui renvoie `True` s'il existe une solution au problème pour le poids total somme en utilisant les éléments de entiers à partir de l'indice index, et `False` sinon.
3. Écrire une fonction `sac_a_dos_sol(somme, entiers, index, chemin)` qui renvoie une solution au problème sous la forme d'une liste d'entiers, ou `None` s'il n'en existe aucune.
4. Écrire une fonction `sac_a_dos_liste_sol(somme, entiers, index, chemin)` qui renvoie la liste de toutes les solutions au problème.

SOLUTION

```
def sac_a_dos_booleen(somme, entiers, index=0):
    if somme == 0:
        return True
    if somme < 0 or index >= len(entiers):
        return False
    return sac_a_dos_booleen(somme - entiers[index], entiers, index + 1) or sac_a_dos_booleen(somme, e

def sac_a_dos_sol(somme, entiers, index=0, chemin=[]):
    if somme == 0:
        return chemin
    if somme < 0 or index >= len(entiers):
        return None
    avec = sac_a_dos_sol(somme - entiers[index], entiers, index + 1, chemin + [entiers[index]])
    if avec is not None:
        return avec
    return sac_a_dos_sol(somme, entiers, index + 1, chemin)

def sac_a_dos_liste_sol(somme, entiers, index=0, chemin=[]):
    if somme == 0:
        return [chemin]
    if somme < 0 or index >= len(entiers):
        return []
    avec = sac_a_dos_liste_sol(somme - entiers[index], entiers, index + 1, chemin + [entiers[index]])
    sans = sac_a_dos_liste_sol(somme, entiers, index + 1, chemin)
    return avec + sans
```

Exercice 7. *Monnayeur*

Ce problème est une variante du précédent, dans laquelle il est possible d'utiliser plusieurs fois la même valeur. On dispose de pièces de différentes valeurs en nombre illimité, et on veut rendre la monnaie pour une somme entière $S \geq 0$. La liste des valeurs de pièces disponibles sera fournie sous la forme d'une liste d'entiers pieces (ce n'est pas forcément l'habituelle suite 1, 2, 5, 10, 20, 50, 100, . . .).

Exemples :

```
> > > monnayeur_nb(127,[1,10,97,100])
4
```

1. Écrire une fonction `monnayeur_nb(somme, pieces, index)` qui renvoie le nombre minimum de pièces nécessaires pour atteindre somme en n'utilisant que des pièces de la liste pieces (donnée par ordre décroissant, et contenant forcément 1) à partir de l'indice index, ou `None` s'il n'y a pas de solution.
2. Écrire une fonction `monnayeur_sol(somme, pieces, index)` qui renvoie une solution de longueur minimale sous la forme d'une liste de valeurs de pièces (qui contiendra en général des entiers répétés), ou None s'il n'y a pas de solution.

SOLUTION

```
def monnayeur_nb(somme, pieces, index=0):
    if somme == 0:
        return 0
    if somme < 0 or index == len(pieces):
        return float('inf')
    prendre = 1 + monnayeur_nb(somme - pieces[index], pieces, index)
```

```
    ne_pas_prendre = monnayeur_nb(somme, pieces, index + 1)
    return min(prendre, ne_pas_prendre)

def monnayeur_sol(somme, pieces, index=0):
    if somme == 0:
        return []
    if somme < 0 or index == len(pieces):
        return None
    avec_piece = monnayeur_sol(somme - pieces[index], pieces, index)
    if avec_piece is not None:
        return [pieces[index]] + avec_piece
    sans_piece = monnayeur_sol(somme, pieces, index + 1)
    return sans_piece
```
