Série 10 - Corrigé

Sauf si spécifié autrement, tous les algorithmes demandés sont à écrire sous forme d'une fonction Python.

Exercice 1

(a) Simulez à la main le parcours de `tri_par_selection` sur la liste ci-dessous en affichant le contenu de la liste à la fin de chaque itération de la boucle `for` extérieure. Vérifiez votre réponse en appelant `tri_par_selection` sur la liste en entrée dans un Jupyter Notebook (et en insérant une instruction `print` dans le code).
`[7, 2, 13, -5, -3, 17, 8, 24, -11, 1]`

(b) Même question pour `tri_par_insertion`.

(c) On considère l'algorithme de fusion vu en cours (slide 19). Qu'affichent les appels suivants à `fusion` ?

(i)
```
L = [1, 3, 5, 7, 2, 4, 6, 8]
fusion(L, 2, 3, 5)
print(L)
```

(ii)
```
L = [5, 1, 5, 6, 1, 2, 4, 3]
fusion(L, 2, 3, 6)
print(L)
```

(iii)
```
L = [1, 3, 5, 7, 1, 2, 3, 4]
fusion(L, 3, 3, 7)
print(L)
```

(d) Le fichier `tri_par_fusion.ipynb` contient le code de l'algorithme de tri par fusion vu en cours, avec des instructions `print` ajoutées de façon à montrer l'état de la liste à différents points de l'exécution. Réfléchissez à l'affichage produit par l'exécution des instructions suivantes :
`L = [4, 3, 2, 1]`
`tri_par_fusion(L, 0, len(L)-1)`
et vérifiez votre réponse en exécutant le code. Testez votre compréhension de l'algorithme de tri par fusion en exécutant le code donné pour divers choix de la liste `L`.

Solution.

(a) On modifie le code de `tri_par_selection` en rajoutant une instruction `print` et un appel à `tri_par_selection` avec cette liste en argument :

```
def tri_par_selection(L):
    '''
    Entree: liste L de nombres
    Trie L
    '''
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1, n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[m_index], L[i] = L[i], L[m_index]
        print(L)

tri_par_selection([7, 2, 13, -5, -3, 17, 8, 24, -11, 1])
```

Ce code affiche l'état de la liste à la fin de chaque itération :

```
[-11, 2, 13, -5, -3, 17, 8, 24, 7, 1]
[-11, -5, 13, 2, -3, 17, 8, 24, 7, 1]
[-11, -5, -3, 2, 13, 17, 8, 24, 7, 1]
[-11, -5, -3, 1, 13, 17, 8, 24, 7, 2]
[-11, -5, -3, 1, 2, 17, 8, 24, 7, 13]
[-11, -5, -3, 1, 2, 7, 8, 24, 17, 13]
[-11, -5, -3, 1, 2, 7, 8, 24, 17, 13]
[-11, -5, -3, 1, 2, 7, 8, 13, 17, 24]
[-11, -5, -3, 1, 2, 7, 8, 13, 17, 24]
[-11, -5, -3, 1, 2, 7, 8, 13, 17, 24]
```

(b) On modifie de même le code de `tri_par_insertion` :

```
def tri_par_insertion(L):
    '''
    Entree: liste L de nombres
    Sortie: liste L triee
    '''
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
        print(L)

tri_par_insertion([7, 2, 13, -5, -3, 17, 8, 24, -11, 1])
```

L'état de la liste est affiché à la fin de chaque itération de la boucle `for` extérieure :

```
[7, 2, 13, -5, -3, 17, 8, 24, -11, 1]
[2, 7, 13, -5, -3, 17, 8, 24, -11, 1]
[2, 7, 13, -5, -3, 17, 8, 24, -11, 1]
[-5, 2, 7, 13, -3, 17, 8, 24, -11, 1]
[-5, -3, 2, 7, 13, 17, 8, 24, -11, 1]
[-5, -3, 2, 7, 13, 17, 8, 24, -11, 1]
[-5, -3, 2, 7, 8, 13, 17, 24, -11, 1]
[-5, -3, 2, 7, 8, 13, 17, 24, -11, 1]
[-11, -5, -3, 2, 7, 8, 13, 17, 24, 1]
[-11, -5, -3, 1, 2, 7, 8, 13, 17, 24]
```

(c) Exécutez les appels !

Exercice 2

Nous avons analysé en cours le temps de parcours de `tri_par_insertion` au pire des cas, qui correspond au cas où la liste est triée par ordre décroissant. A quoi correspond le meilleur des cas pour `tri_par_insertion` ? Donner l'ordre du temps de parcours en notation $\Theta(\cdot)$ dans ce cas.

Solution.

Le temps de parcours est le plus petit possible lorsqu'on ne rentre jamais dans la boucle `while`. Cela a lieu si pour tout `i`, `L[i]` $\geq$ `L[i-1]`, ce qui correspond

au cas où la liste est déjà triée. Dans ce cas le temps de parcours sera $\Theta(n)$.

Exercice 3

Récrivez l'algorithme `fusion` vu en cours sans utiliser les valeurs sentinelles `float(''inf'')`.

Solution.

L'algorithme de fusion ci-dessous, au lieu d'utiliser des valeurs sentinelles, effectue la vérification suivante avant de comparer le prochain élément de `L1` avec le prochain élément de `L2` : si une des sous-listes a déjà été entièrement copiée (par exemple si l'index `i1` a atteint la valeur `n1`, ce qui signifie que tous les éléments de `L1` ont été copiés dans `L`), alors on se contente de copier toutes les valeurs restantes de l'autre liste aux indices restants de `L`.

```
def fusion2(L, bas, milieu, haut):
    L1 = L[bas:milieu+1]
    L2 = L[milieu+1:haut+1]
    n1 = len(L1)
    n2 = len(L2)

    i1 = i2 = 0

    for i in range(bas, haut+1):
        if i1 == n1:
            L[i] = L2[i2]
            i2 += 1
        elif i2 == n2:
            L[i] = L1[i1]
            i1 += 1
        elif L1[i1] < L2[i2]:
            L[i] = L1[i1]
            i1 += 1
        else:
            L[i] = L2[i2]
            i2 += 1
```

Exercice 4

Le **tri à bulles** (bubble sort) fonctionne de la manière suivante : Il prend en entrée une liste `L` de taille n. Il parcourt la liste n fois. A chaque parcours, dès qu'il rencontre deux éléments adjacents de la liste qui sont dans le mauvais ordre, c'est-à-dire `L[i], L[i+1]` tels que `L[i] > L[i+1]`, il les échange.

On appelle cet algorithme tri à bulles car à chaque parcours de la liste, le plus grand élément (restant) se déplace vers la fin de la liste comme une bulle qui remonte à la surface de l'eau.

(a) Observez une simulation du tri à bulles sur l'un des sites proposés en cours, par exemple https://visualgo.net/bn/sorting, pour comprendre exactement comment fonctionne l'algorithme.

(b) Donnez l'algorithme du tri à bulles en Python. N'oubliez pas de tester votre code sur de petites listes.

(c) Formulez et prouvez un invariant de boucle pour la boucle extérieure de l'algorithme.

(d) Quel est le temps de parcours du tri à bulles ?

Solution.

(b) Voici le code Python qui implémente l'algorithme du tri à bulles :

```
def tri_a_bulles(L):
    '''
    Entree: liste L de taille n
    Trie L
    '''
    n = len(L)
    for i in range(n):
        for j in range(n-i-1):
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]
```

On remarque qu'à l'itération `i` de la boucle extérieure, la boucle intérieure parcourt uniquement les éléments `L[:n-i-1]` (en changeant potentiellement le `n-i-1`ème). C'est parce que à cette itération, les `i` plus grands éléments sont déjà fixés à leur bonne position à la fin de la liste (voir l'invariant de boucle au point (c)).

(c) Au début de l'itération `i` de la boucle extérieure, `L[n-i:]` contient les `i` plus grands éléments de la liste, triés.
On donne ci-dessous la preuve de cet invariant :

— **Initialisation** : au début de l'itération `0`, donc avant le début de la boucle, `L[n:]` est la liste vide (qui contient les `0` plus grands éléments de la liste).

— **Maintenance** : On suppose qu'au début de l'itération `i` de la boucle extérieure, `L[n-i:]` contient les `i` plus grands éléments de la liste, triés. Or l'itération `i` place le `i`ème plus grand élément à l'index `n-i`. Pour le prouver, il faudrait prouver l'invariant de boucle suivant pour la boucle intérieure à l'itération `i` :

Au début de l'itération `j`, `L[j]` est plus grand que tous les éléments de `L[0:j]`.
La terminaison de cet invariant donne qu'à la sortie de la boucle intérieure, `L[n-i-1]` contient le `i+1`ème plus grand élément de la liste.

— **Terminaison** : A la sortie de la boucle **for**, càd à l'itération `n` (qui n'aura pas lieu), `L[0:]` (qui correspond à toute la liste), contient les `n` plus grands éléments de la liste (donc tous les éléments de la liste), triés.

(d) Les deux boucles `for` imbriquées donnent un temps de parcours $\Theta(n^2)$.

Exercice 5

Etant donnés un nombre x et une liste `L` de n nombres tous distincts, on veut trouver un algorithme pour déterminer s'il existe deux éléments de `L` dont la somme vaut `x`. L'algorithme doit retourner des valeurs distinctes `L[i]` et `L[j]` telles que `L[i] + L[j] = x`. S'il n'existe pas de tels éléments, il retourne `None`.
Par exemple, pour l'entrée `10, [1, 2, 3, 5, 7, 8]`, l'algorithme peut retourner `(2, 8)` ou `(3, 7)`, mais pas `(5, 5)`.

(a) Donnez un algorithme naïf de temps de parcours quadratique qui résoud ce problème.

(b) Donnez un algorithme qui résoud ce problème en temps $\Theta(n \log_2(n))$. Vous pouvez faire appel à des algorithmes vus en cours.
Indice : [1]

(c) On suppose maintenant que la liste `L` est déjà triée. Donner un algorithme qui résoud ce problème en temps linéaire en n.
Indice : [2]

Solution.

Vous trouverez sur la page Moodle le fichier `somme_deux.ipynb` qui réunit tous les algorithmes présentés ci-dessous.

(a) L'algorithme ci-dessous trouve la paire d'éléments spécifiée :

1. Commencez par trier la liste. Puis répétez une certaine opération n fois.
2. Mettez un doigt au début de la liste et un doigt à la fin...

```python
def somme_deux_naif(x, L):
    '''
    Entree: liste L de nombres distincts, nombre x
    Sortie: L[i], L[j] distincts tq L[i] + L[j] = x
            None si aucuns tels L[i], L[j] n'existent
    '''
    n = len(L)
    for i in range(n):
        for j in range(i+1, n):
            if L[i] + L[j] == x:
                return L[i], L[j]
```

Pour une liste de taille n en entrée, `somme_deux_naif` a temps de parcours $\Theta(n^2)$.

(b) L'algorithme ci-dessous fait appel à `tri_par_fusion` pour trier la liste `L`. Ensuite, pour chaque élément `L[i]` de la liste, il fait appel à `recherche_binaire` pour trouver `x - L[i]` dans la liste, pour un temps de parcours total de $\Theta(n \log_2 n)$.

```python
def somme_deux(x, L):
    '''
    Entree: liste L de nombres distincts, nombre x
    Sortie: L[i], L[j] distincts tq L[i] + L[j] = x
            None si aucuns tels L[i], L[j] n'existent
    '''
    n = len(L)
    tri_par_fusion(L, 0, n-1)

    for i in range(n):
        if x - L[i] == L[i]:
            return None #pour ne pas retourner (L[i],L[i])
        j = recherche_binaire(L, x - L[i])
        if j != None:
            return L[i], L[j]
```

(c) L'algorithme ci-dessous suppose que la liste est triée. Il maintient deux indices, `i` qui commence au début de la liste et `j` à la fin. A chaque itération de la boucle il calcule `L[i] + L[j]`. Si cette somme est plus petite que `x`, il augmente l'index `i` ; si elle est plus grande que `x` il décrémente l'index `j`.

```
def somme_deux_lineaire(x, L):
    '''
    Entree: liste L triee de nombres distincts, nombre x
    Sortie: L[i], L[j] distincts tq L[i] + L[j] = x
            None si aucuns tels L[i], L[j] n'existent
    '''
    i = 0
    j = len(L) - 1

    while i < j:
        if L[i] + L[j] == x:
            return L[i], L[j]

        if L[i] + L[j] < x:
            i += 1
        else:
            j -= 1
```

La boucle `while` termine bien car la valeur de `j-i`, qui est égale à $n-1$ avant la boucle (où n est la taille de la liste), diminue de 1 à chaque itération.

Si l'algorithme retourne une paire `(i,j)`, elle est bien telle que `L[i] + L[j] = x`. L'algorithme est correct aussi lorsqu'il retourne `None`, par l'invariant suivant :

A chaque itération de la boucle `while`, les éléments d'index plus petit que `i` ou plus grand que `j` ne peuvent pas faire partie d'une paire solution.

Pour comprendre cela, raisonnons sur la première itération de la boucle, lorsque `i` indexe le premier (le plus petit) élément, et `j` le dernier (le plus grand) élément. A ce stade l'ensemble des éléments d'index plus petit que `i` ou plus grand que `j` est vide. Si `L[0] + L[n-1] < x`, l'élément `L[0]` ne peut pas faire partie d'une paire solution : en effet, en lui ajoutant le plus grand élément possible, on a toujours une somme trop petite. Dans ce cas on incrémente `i`. Et si `L[0] + L[n-1]`

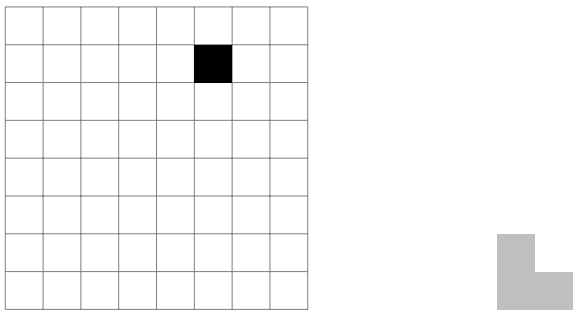
`> x`, `L[n-1]` ne peut pas faire partie d'une paire solution, puisqu'en lui ajoutant le plus petit élément possible on a toujours une somme plus grande. Dans ce cas, on décrémente `j`.

De même, on considère n'importe quelle itération de la boucle avec certaines valeurs de `i` et de `j`, et on suppose que les éléments d'index plus petit que `i` ou plus grand que `j` ne peuvent pas faire partie d'une paire solution. Si `L[i] + L[j] < x`, alors `L[i]` ne peut pas faire partie d'une paire solution puisqu'en lui ajoutant le plus grand élément permis on a toujours une somme trop petite, et dans ce cas on incrémente `i`. On raisonne de même sur `j`.
Lorsque les indices `i` et `j` se croisent, tous les éléments de la liste ont été éliminés et donc il n'est possible de former aucune paire qui somme à `x`.

Exercice 6 (facultatif)

Pour $n \geq 1$, on vous donne

- un tableau de dimensions $2^n \times 2^n$ avec une case manquante (qui peut être n'importe quelle case)
- et un nombre suffisant de **trominos** : ce sont des pièces qui couvrent trois cases d'un tableau formant un angle droit.



Un tableau 8×8 avec la case manquante indiquée en noir ; un tromino

Donnez un algorithme (en français) qui permet de paver le tableau complètement et exactement avec des trominos (les trominos ne se chevauchent pas, ne dépassent pas, la case manquante n'est pas couverte, et aucune autre case ne reste découverte).

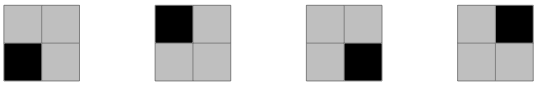
Indice : [3]

3. Divisez pour régner.

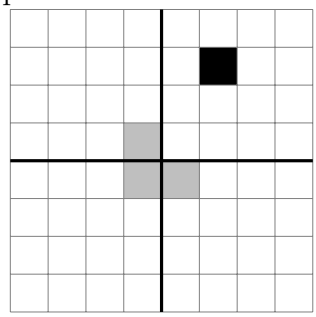
Solution.

On pave le tableau récursivement avec des trominos.

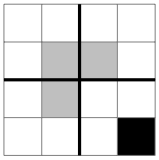
— Pour $n = 1$, le tableau est 2×2. Il a quatre cases, dont une manquante. On couvre simplement les trois cases restantes avec un tromino. Les quatre configurations possibles sont présentées ci-dessous.



— Pour $n > 1$, on divise le tableau en quatre quadrants de dimensions $2^{n-1} \times 2^{n-1}$. Un de ces quatre quadrants contient la case manquante ; on peut le paver récursivement. Mais que faire des trois autres ?
La solution est de poser un tromino au milieu du tableau, là où les quatre quadrants se rencontrent, de façon à couvrir une case de chacun des trois quadrants qui n'ont pas la case manquante, comme dans la figure ci-dessous. Maintenant on peut considérer la case couverte par le tromino comme une case manquante dans chacun de ces trois quadrants, qu'on peut paver récursivement.



Par exemple, pour paver récursivement le quadrant en haut à gauche dans la figure ci-dessus, on le divise de nouveau en quatre quadrants, et on place un tromino de manière appropriée à la jointure des quatre quadrants, comme ci-dessous.



Maintenant chacun des quatre quadrants ainsi obtenus est un tableau 2×2 avec une case manquante ; il suffit d'y placer un tromino selon une des quatre configurations de base.