Chapitre 3
Corrigé des exercices

Tris par comparaison

Exercice 1

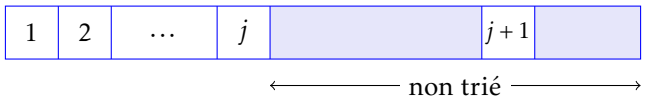
```
def minimum(t, j):
    if j == len(t)-1:
        return j
    i = minimum(t, j+1)
    if t[j] < t[i]:
        return j
    else:
        return i

def select_sort(t, j=0):
    if j < len(t)-1:
        i = minimum(t, j)
        if i > j:
            t[i], t[j] = t[j], t[i]
        select_sort(t, j+1)
```

```
def insere(t, j):
    if j > 0 and t[j] < t[j-1]:
        t[j-1], t[j] = t[j], t[j-1]
        insere(t, j-1)

def insertion_sort(t, j=1):
    if j < len(t):
        insere(t, j)
        insertion_sort(t, j+1)
```

Exercice 2 Observons la situation après avoir déjà trouvé les j plus petits éléments et déterminé l'endroit où se trouve l'élément $j+1$:



La partie non triée correspond à une permutation de $\mathfrak{S}(\llbracket j+1, n \rrbracket)$. Si on les suppose toutes équiprobables, la probabilité pour que $j+1$ soit situé à son endroit définitif et ne nécessite donc pas de permutation est égale à $\dfrac{(n-j-1)!}{(n-j)!} = \dfrac{1}{n-j}$. Le nombre de permutation moyen est donc égal à :

$$\sum_{j=0}^{n-2}\left(1-\frac{1}{n-j}\right) = n - \sum_{k=1}^{n}\frac{1}{k} = n - \ln n - \gamma + o(1).$$

Exercice 3 Lorsqu'on parcourt le tableau (de la gauche vers la droite) en permutant deux éléments consécutifs à chaque fois que l'élément le plus petit se trouve à droite du plus grand, on est assuré en fin de parcours d'avoir placé le plus grand élément du tableau à sa place définitive, en ayant effectué $n-1$ comparaisons et au plus $n-1$ permutations.
Il reste à réitérer le procédé sur le tableau privé de sa dernière case pour obtenir l'algorithme de tri bulle, ainsi nommé car les éléments les plus grands du tableau se dirigent vers leur place définitive à l'image des bulles d'air qui remontent à la surface d'un liquide.
On rédige une fonction de « remontée » des bulles (jusqu'au niveau k) :

```
def remonte(t, k):
    for j in range(k-1):
        if t[j] > t[j+1]:
            t[j], t[j+1] = t[j+1], t[j]
```

puis la fonction de tri proprement dit :

```
def bubble_sort(t):
    for k in range(len(t), 0, -1):
        remonte(t, k)
```

Notons c_n le nombre de comparaisons effectuées, et p_n le nombre de permutations de deux éléments. Nous avons $c_n = n-1+c_{n-1}$ et $p_{n-1} \leqslant p_n \leqslant p_{n-1}+n-1$, donc $c_n = \dfrac{n(n-1)}{2}$ et $0 \leqslant p_n \leqslant \dfrac{n(n-1)}{2}$.
Il s'agit donc d'un algorithme de coût quadratique, le pire des cas ayant lieu lorsque le tableau est trié à l'envers, car alors $c_n = p_n = \dfrac{n(n-1)}{2}$.

Remarque. Il est possible d'améliorer légèrement cet algorithme en observant que si lors d'une étape de remontée aucune permutation n'est effectuée, c'est que le tableau est trié, et qu'on peut donc s'arrêter là.
Pour exploiter cette remarque on peut par exemple faire en sorte que la fonction `remonte` renvoie une valeur booléenne indiquant si aucune permutation n'a été réalisée. Ceci conduit à cette seconde version :

```
def remonte(t, k):
    b = False
    for j in range(k-1):
        if t[j] > t[j+1]:
            t[j], t[j+1] = t[j+1], t[j]
            b = True
    return b

def bubble_sort(t):
    b, k = True, len(t)
    while b and k > 0:
        b = remonte(t, k)
        k -= 1
```

Avec cette nouvelle version, le nombre d'échanges est inchangé mais le nombre de comparaisons n'est plus systématiquement quadratique. Dans le cas d'un tableau déjà trié, par exemple, cet algorithme n'effectue que $n-1$ comparaisons et aucune permutation.

On peut aussi observer que le nombre de permutations effectuées reste identique dans chacune des deux versions, très précisément égal au nombre d'inversions de la permutation initiale (c'est à dire le nombre de couples (i,j) tel que $i<j$ et $t[j]<t[i]$), au moins dans les cas où toutes les valeurs du tableau sont distinctes.
Il n'est pas difficile de prouver que le nombre moyens d'inversions est égal à $\dfrac{n(n-1)}{4}$, ce qui assure un coût quadratique en moyenne.

Exercice 4 La version itérative de l'algorithme de tri fusion consiste à fusionner les cases deux par deux, puis quatre par quatre, huit par huit, etc. On peut utiliser la fonction `fusion` du cours, et la fonction de tri à proprement parler devient :

```
def merge_sort(t):
    n = len(t)
    aux = [None] * n
    p = 1
    while p < n:
        q = n // p
        for k in range(0, q-1, 2):
            fusion(t, k*p, (k+1)*p, (k+2)*p, aux)
            t[k*p:(k+2)*p] = aux[k*p:(k+2)*p]
        if q % 2 == 1:
            k = q - 1
            fusion(t, k*p, (k+1)*p, n, aux)
            t[k*p:n] = aux[k*p:n]
        p *= 2
```

Prouvons la validité de cet algorithme en supposant qu'après $p-1$ étapes on ait $n = pq+r$ avec $0 \leqslant r < p$ et que le tableau t soit la concaténation de q tableaux triés de longueur p et d'un tableau de longueur r. Deux cas sont à envisager :

- si $q = 2q'$ est pair, les tableaux sont fusionnés deux par deux pour former des tableaux triés de longueur $2p$ et le dernier de longueur r n'est pas fusionné ;
- si $q = 2q' - 1$ est impair, les tableaux sont fusionnés deux par deux pour former des tableaux triés de longueur $2p$ et le dernier de longueur p est fusionné avec celui de longueur r pour former un tableau trié de longueur $p + r$.

En posant $p' = 2p$ et $r' = r$ ou $t' = p + r$ suivant la parité de q nous pouvons affirmer que $n = p'q' + r'$ avec $0 \leqslant r' < p'$ et le tableau t est maintenant constitué de q' tableaux triés de longueur $2p$ et d'un tableau trié de longueur r'.
L'algorithme se termine lorsque $p > n$ car alors $q = 0$ et $r = n$.

Exercice 5

a) Lors du tri par insertion d'un tableau déjà 2-trié, le nombre maximal d'échanges est obtenu lorsque les termes d'indices impairs sont tous inférieurs aux termes d'indices pairs. Dans cette situation, le nombre d'échanges effectués est égal à $1 + 2 + \cdots + \left\lceil \frac{n}{2} \right\rceil = \frac{p(p+1)}{2}$ avec $p = \left\lceil \frac{n}{2} \right\rceil$.
Le tableau trié prend alors la forme : $(a_1, a_3, a_5, \ldots, a_0, a_2, a_4, \ldots)$.

b) Partons d'un tableau trié, par exemple $(1, 2, 3, 4)$. Pour que le nombre d'échanges par le 1-tri soit maximal, il faut qu'à l'étape précédente on soit en présence du tableau $(3, 1, 4, 2)$. De même, pour que le nombre d'échanges par le 2-tri soit maximal, il faut qu'à l'étape précédente on soit en présence du tableau $(4, 2, 3, 1)$.
Ce dernier tableau nécessite donc 5 échanges pour être trié par le tri de Shell : à la première étape (le 2-tri), 2 échanges le transforment en $(3, 1, 4, 2)$, à la seconde étape (le 1-tri), 3 échanges en transforment en $(1, 2, 3, 4)$.

C'est la même chose pour $n = 8$ en partant de $(1, 2, 3, 4, 5, 6, 7, 8)$. Avant le 1-tri, il faut avoir $(5, 1, 6, 2, 7, 3, 8, 4)$, avant le 2-tri, $(7, 3, 5, 1, 8, 4, 6, 2)$, et avant le 4-tri, $(8, 4, 6, 2, 7, 3, 5, 1)$. Avec ce dernier tableau, le tri de Shell effectue 20 échanges.

c) Le tri de Shell étudié ici revient à trier récursivement les termes pairs et impairs séparément puis à appliquer le 1-tri (le tri par insertion) au tableau obtenu. D'après la première question, le nombre maximal d'échanges effectués vérifie la relation : $C(2^p) = 2C(2^{p-1}) + \frac{2^{p-1}(2^{p-1}+1)}{2}$, soit : $u_p = 2u_{p-1} + 2^{p-2}(2^{p-1} + 1)$.
Écrivons : $\frac{u_p}{2^p} = \frac{u_{p-1}}{2^{p-1}} + \frac{2^{p-1}+1}{4}$ de manière à réaliser un télescopage. Sachant que $u_0 = 0$ on obtient : $u_p = 2^p \sum_{k=0}^{p-1} \frac{2^k + 1}{4} = 2^{p-2}(2^p - 1 + p) = \frac{2^p(2^p - 1)}{4} + p2^{p-2}$.

Lorsque $n = 2^p$, le nombre d'échanges effectués est égal dans le pire des cas à $\frac{n(n-1)}{4} + \frac{n \log n}{4}$. Le coût reste quadratique, mais asymptotiquement le nombre d'échanges est deux fois moindre que pour le tri par insertion. Cela n'en reste pas moins mauvais, et d'autres choix pour la suite h_p conduisent à de meilleurs résultats. Par exemple, la suite définie par $h_p = 2^p - 1$ conduit à un coût dans la pire des cas en $\Theta(n^{3/2})$ (Hibbard, 1963). Mieux, la suite des nombres de la forme $2^a 3^b$ conduit à un coût en $\Theta(n \log^2 n)$ (Pratt, 1971). Mais la recherche de la meilleure suite de valeurs pour h_p reste à l'heure actuelle un problème ouvert.

d) On écrit tout d'abord une fonction `insertion` qui réalise le h-tri d'un tableau.

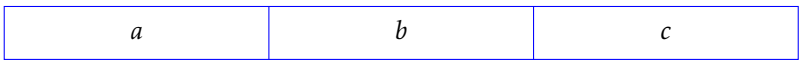
```
def insertion(t, h):
    for j in range(h, len(t)):
        k = j
        while k >= h and t[k] < t[k-h]:
            t[k-h], t[k] = t[k], t[k-h]
            k -= 1
```

Le tri proprement dit consiste essentiellement à calculer la plus grande valeur h_p de la suite qui vérifie $h_p \leqslant n < h_{p+1}$ puis à appliquer la fonction précédente avec $h_p, h_{p-1}, \ldots, h_1 = 1$.

```
def shell_sort(t):
    h = 1
    while 2 * h + 1 < len(t):
        h = 2 * h + 1
    while h > 0:
        insertion(t, h)
        h = (h - 1) // 2
```

Exercice 6 Montrons par récurrence forte sur $n = j - i \geqslant 2$ que `tri(t, i, j)` trie correctement le tableau $t[i : j]$.

- Si $n = 2$ l'algorithme réalise au plus une permutation pour trier le tableau à deux cases et ne fait pas d'appel récursif.
- Si $n \geqslant 3$ on suppose le résultat acquis jusqu'au rang $n-1$.
  L'algorithme scinde la tableau en trois parties : $a = t[i : i+k]$, $b = t[i+k : j-k]$, $c = t[j-k : j]$.



  On a $|a| = |c| = k$ et $|b| = j - i - 2k$ avec $k = \left\lfloor \frac{j-i}{3} \right\rfloor$ donc $|b| \geqslant |a| = |c|$.

  Le tableau $a + b$ est tout d'abord trié ; ceci assure que dans b se trouvent désormais (au moins) les k plus grands éléments de $a + b$;

  Le tableau $b+c$ est ensuite trié ; ceci assure que les k plus grands éléments de $a+b+c$ se trouvent maintenant triés dans c.

  Le dernier appel récursif trie enfin les $j - i - k$ derniers éléments dans $a + b$, ce qui assure qu'après ces trois appels récursifs le tableau t est trié.

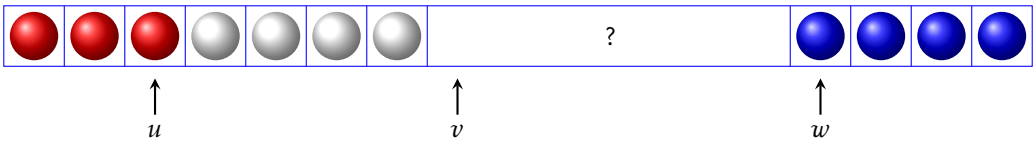
Si $C(n)$ désigne le coût temporel de cet algorithme lorsque $n = |t|$, on dispose de la relation : $C(n) = 3C(n-k)+\Theta(1)$ avec $k = \lfloor n/3 \rfloor$, soit $C(n) = 3C(\lceil 2n/3 \rceil) + \Theta(1)$.
Supposons avoir trouvé une valeur de α et une constante M telles que $C(n') \geqslant Mn'^{\alpha}$ pour tout $n < n'$. Alors :

$$C(n) \geqslant 3M\left\lceil \frac{2n}{3} \right\rceil^{\alpha} + \Theta(1) \geqslant 3M\left(\frac{2}{3}\right)^{\alpha} n^{\alpha}.$$

En choisissant α vérifiant $3\left(\frac{2}{3}\right)^{\alpha} = 1$ nous avons prouvé que $C(n) = \Omega(n^{\alpha})$. Sachant que $\alpha = \dfrac{\ln 3}{\ln 3 - \ln 2} \approx 2{,}71$ cet algorithme a un coût plus que quadratique, c'est le plus médiocre que nous ayons rencontré !

Segmentation et tri rapide

Exercice 7 Le principe est semblable au principe de segmentation du tri rapide : on parcours le tableau en ayant pour invariants les entiers u, v, w définis par le shéma ci-dessous :



Tant que $v < w$, on regarde la couleur de $t[v]$:

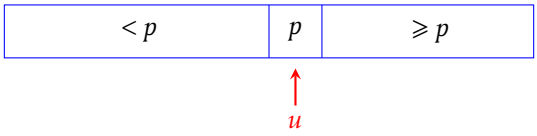
- si celle-ci est rouge, on permute cet élément avec celui d'indice $u+1$, et on incrémente u et v ;
- si celle-ci est blanche, on incrémente v ;
- si celle-ci est rouge, on permute cet élément avec celui d'indice $w-1$, et on décrémente w.

Traduit en Python, cela donne :

```
def drapeau(t):
    u, v, w = -1, 0, len(t)
    while v < w:
        c = t[v].couleur
        if c == 'rouge':
            t[u+1], t[v] = t[v], t[u+1]
            u += 1
            v += 1
        elif c == 'bleu':
            t[v], t[w-1] = t[w-1], t[v]
            w -= 1
        else:
            v += 1
```

La terminaison de cette fonction est assurée par le fait que la valeur de $w - v$ décroit d'une unité à chaque itération.

Exercice 8 Observons la situation après la segmentation par un pivot p :



- Si $k \leqslant u$, le k^{e} plus petit élément du tableau est à chercher entre les indices 0 et $u-1$;
- si $k = u+1$, le k^{e} plus petit élément du tableau est l'élément p qui a été pris pour pivot ;
- si $k > u+1$, le k^{e} plus petit élément du tableau est aussi le $(k-u-1)^{\text{e}}$ élément du sous-tableau délimité par les indices $u+1$ et $n-1$.

Ceci conduit à la fonction suivante (utilisant la fonction `segmente` du cours) :

```
def kieme(t, k, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    u = segmente(t, i, j)
    if k <= u - i:
        return kieme(t, k, i, u)
    elif k > u - i + 1:
        return kieme(t, k-u+i-1, u+1, j)
    else:
        return t[u]
```

Le calcul de l'élément médian consiste à appliquer la fonction précédente avec $k = \lceil n/2 \rceil$:

```
def median(t):
    k = (len(t) + 1) // 2
    return kieme(t, k)
```

Après avoir trié un tableau on peut déterminer en temps linéaire l'élément médian de ce dernier ; l'algorithme de tri rapide permet donc d'avoir un algorithme de coût semi-linéaire en moyenne (et quadratique dans le pire des cas) pour calculer le médian. L'algorithme ci-dessus ne présente donc d'intérêt que s'il permet de faire mieux.
Notons $\mathrm{C}(n)$ le coût *en moyenne* du calcul du k^{e} élément de t. Le coût de la segmentation étant linéaire, on dispose de la relation :

$$\mathrm{C}(n) = \frac{1}{n}\sum_{i=1}^{n} \max\Big(\mathrm{C}(i-1), \mathrm{C}(n-i)\Big) + \Theta(n).$$

Supposons l'existence d'une constante M telle que $\mathrm{C}(i) \leqslant \mathrm{M}i$ pour $i < n$. Alors :

$$\mathrm{C}(n) \leqslant \frac{\mathrm{M}}{n}\sum_{i=1}^{n} \max(i-1, n-i) + \Theta(n) \leqslant (\text{calcul}) \leqslant \frac{3}{4}\mathrm{M}n + \Theta(n).$$

Il suffit donc de choisir M assez grand pour que $C(n) \leqslant Mn$, ce qui prouve que $C(n) = O(n)$. En moyenne, cet algorithme calcule l'élément médian en temps linéaire.

Remarque. En choisissant adroitement le pivot dans cet algorithme, il est possible de modifier cet algorithme pour obtenir un algorithme de coût linéaire dans le pire des cas.

Exercice 9

a) Trier 3 valeurs demande 3 comparaisons, donc $u_p = c_{3^p}$ vérifie les relations : $u_0 = 0$ et $u_p = 3.3^{p-1} + u_{p-1}$, qui se résolvent en : $u_p = \frac{3}{2}(3^p - 1)$. Lorsque n est une puissance de 3, nous avons donc $c_n = \frac{3}{2}(n-1)$.

b) Notons m_n le nombre d'éléments plus petits que le résultat donné par cet algorithme. Cette suite vérifie les relations : $m_3 = 1$ et $m_{3^p} \geqslant 2m_{3^{p-1}}$, donc $m_{3^p} \geqslant 2^p = 3^{p\log_3 2}$. On en déduit que si n est une puissance de 3, $m_n \geqslant n^\alpha$, avec $\alpha = \log_3 2$.
On montre de même qu'il y a au moins n^α éléments plus grands, ce qui prouve que cet algorithme détermine un élément α-pseudomédian avec $\alpha = \log_3 2$.

c) L'algorithme de tri rapide se révèle d'autant moins performant que la segmentation produit une partition déséquilibrée du tableau. Calculer un élément α-pseudomédian et s'en servir comme pivot de la segmentation permet de s'assurer que chacun des deux sous-tableaux contiendra au moins n^α éléments.

Tris de coûts linéaires

Exercice 10 Posons $k = \lceil p/2 \rceil$, et segmentons (par exemple par une méthode analogue à l'algorithme du drapeau tricolore) le tableau de manière à obtenir :

$< k$	$= k$	$> k$

On peut observer que les éléments de la partie centrale sont à leurs places définitives. Les deux autres parties ont des éléments dans $[\![1, k-1]\!]$ et $[\![k+1, p]\!]$; on leur applique récursivement la segmentation décrite.

Le nombre $c(n,p)$ de comparaisons effectuées vérifie la relation :

$$c(n,p) = n + c(n_1, k-1) + c(n_2, p-k) \quad \text{avec} \quad n_1 + n_2 \leqslant n.$$

Montrons alors par récurrence forte sur p que $\forall n \in \mathbb{N}^*$, $c(n,p) \leqslant n \log p$.

- C'est clair si $p = 1$ car $c(n,1) = 0$.
- Si $p \geqslant 2$, supposons le résultat acquis jusqu'au rang $p-1$.
  Alors $c(n,p) \leqslant n + n_1 \log(k-1) + n_2 \log(p-k) \leqslant n + (n_1+n_2)\log(p/2) \leqslant n\Big(1 + \log(p/2)\Big) = n \log p$.

L'entier p étant fixé, on peut observer que cet algorithme de tri a un coût linéaire, mais ceci ne contredit pas l'étude générale du coût des algorithmes de tri que nous avons faite, car cet algorithme ne procède pas par comparaison entre deux éléments du tableau, mais par comparaison à des valeurs fixées dans $[\![1, p]\!]$.

Exercice 11 Ce problème est aussi connu sous le nom de tri par inversion de préfixe, puisqu'il revient à trier un tableau en s'autorisant uniquement de prendre l'image miroir d'un préfixe quelconque.

a) Lorsqu'on dispose au moins de trois crêpes, glissons la spatule sous la plus grande et retournons-la. Cette crêpe se retrouve au sommet de la pile. Glissons ensuite la spatule sous le tas et retournons-la. La crêpe la plus grande se trouve maintenant en bas de la pile. Il suffit de réitérer ce procédé récursif pour trier les crêpes.
Lorsqu'on dispose de deux crêpes, à l'évidence une seule manipulation est nécessaire, donc le nombre $C(n)$ d'opérations à effectuer vérifie : $C(n) \leqslant C(n-1) + 2$ pour $n \geqslant 3$ et $C(2) \leqslant 1$, ce qui conduit à $C(N) \leqslant 2n - 3$.

Remarque. Des méthodes plus efficaces permettent de majorer $C(n)$ par $18n/11$ pour $n \geqslant 2$ (valeur obtenue en 2008) mais la valeur optimale de $C(n)$ reste à l'heure actuelle inconnue.

b) Il est facile de répondre par l'affirmative, en ajoutant une éventuelle opération supplémentaire à l'algorithme précédent : si, lorsque la crêpe la plus grosse se trouve au sommet de la pile, la face intacte est visible, retourner cette seule crêpe à l'aide de la spatule.
Ceci nous assure que le nombre maximal de retournement par cette méthode vérifie la relation : $C(n) \leqslant C(n-1)+3$. Sachant que $C(1) \leqslant 1$ on en déduit : $C(n) \leqslant 3n - 2$.