Ch 2 : Complexité des algorithmes-Exercices
Eléments de correction

Exercice 1 (Exponentiation rapide-Version itérative)
On désire comparer ici deux algorithmes d'exponentiation prenant en entrée un flottant x et un entier positif n pour calculer x^n.

1. On commence par un algorithme naïf.

```
def expo(x,n):
        p=1
        for i in range(n):
                p=p*x
        return p
```

   Quel est le nombre de multiplications effectuées par cet algorithme ? Donner un équivalent simple en l'infini.

2. On donne ici une version récursive de l'algorithme d'exponentiation rapide. Cette version est basée sur l'idée suivante : pour tout réel x et pour tout entier naturel $n \geq 1$,

   a. $x^n = (x^2)^{\frac{n}{2}}$ si n est pair
   b. $x^n = (x^2)^{\frac{n-1}{2}} \times x$ si n est impair.

   On peut donc calculer x^n au moyen de l'algorithme `exp_rapide_rec`.

```
def expo_rapide_rec(x,n):
        if n==0:
                return 1
        elif n%2==0:
                return expo_rapide_rec(x**2,n//2)
        else return x*expo_rapide_rec(x**2,n//2)
```

   On note $T(n)$ le nombre de multiplications effectuées pour calculer x^n.

   a. Montrer que pour tout entier naturel n, la fonction effectue au plus récursivement $\log_2(n)$ appels.
   b. En déduire une majoration asymptotique de $T(n)$.

Exercice 2 (Exponentiation rapide-Version récursive)
On donne maintenant une version itérative de l'algorithme d'exponentiation rapide, dont on désire montrer la correction et calculer la complexité.

```
def expo_rapide(x,n):
        e=1
        while n>0:
                if n%2==1:
                        e=e*x
                x=x*x
                n=n//2
        return e
```

1. Faire fonctionner cet algorithme « à la main » pour $x = 2$ et $n = 13$. Indiquer les valeurs successives prises par e, x et n. Vérifier que le résultat obtenu est correct.
2. Proposer une spécification d'entrée de l'algorithme et montrer qu'il termine.
3. On souhaite montrer la correction de l'algorithme. On note $\overline{b_{k-1}\dots b_1 b_0}^2$ la décomposition de n en base deux, c'est-à-dire les entiers $(b_i)_{0\leq i\leq k-1}$ tels que pour tout i compris entre 0 et $k-2$, $b_i \in \{0;1\}$, $b_{k-1} = 1$ et $n = b_{k-1}2^{k-1} + \dots b_1 2^1 + b_0 2^0$.

   On note e_i, x_i et n_i les valeurs respectives des variables e, x et n à la fin du $i^{\text{ème}}$ passage dans la boucle.

   Montrer qu'à la fin de la boucle, l'invariant suivant est vérifié (par convention, on considérera qu'une somme ne contenant aucun élément est égale à 0).

$$\begin{aligned} P(i) \quad : \quad & e_i = x^{b_{i-1}2^{i-1}+\dots+b_1 2^1+b_0 2^0} \\ & \text{et } x_i = x^{2^i} \\ & \text{et } n_i = b_{k-1}2^{k-1-i} + \dots b_{i+1}2^1 + b_i 2^0 \end{aligned}$$

4. Préciser la valeur finale de n lors de la terminaison de la boucle, et le nombre de passages effectués. En déduire la correction de l'algorithme.
5. Donner une majoration du nombre de passages dans la boucle effectués en fonction de n, puis du nombre de multilplications effectuées.

Exercice 3
On rappelle ci-dessous l'algorithme de recherche dichotomique d'un entier *elt* dans un tableau de nombres entiers **triés** T.

```
def recherche(T,elt):
        a,b=0,len(T)-1
        c=(a+b)//2
        while b-a>=0 and T[c]!=elt:
                if T[c]<elt:
                        a=c+1
                else b=c-1
                c=(a+b)//2
        return b>=a
```

On désire estimer le nombre d'accès $T(n)$ au tableau effectués dans le pire des cas pour tester la présence d'un élément dans un tableau de taille n.

1. Montrer que le programme termine en utilisant un variant de boucle adapté.
2. Combien de comparaisons sont effectuées au pire à chaque passage dans la boucle ?
3. Donner une majoration du nombre de passages dans la boucle, et en déduire une majoration asymptotique de la complexité.

Exercice 4
On donne ici une version améliorée de l'algorithme de tri bulle, prévue pour s'arrêter dès lors qu'il n'y a plus d'échange à réaliser.

```
def tri_bulle_opti(T):
        i=0
        trie=False
        while i<len(T) and trie==False:
                trie=True
                for j in range(len(T)-1-i):
                        if T[j]>T[j+1]:
                                T[j],T[j+1]=T[j+1],T[j]
                                trie=False
                i=i+1
```

1. Proposer des spécifications d'entrées et de sorties pour l'algorithme.
2. Appliquer cette algorithme à la suite $[3, 1, 5, 4, 2]$ en précisant les valeurs prises par la suite à chaque modification.
3. Justifier que l'algorithme termine.
4. Proposer un invariant vérifié en fin de la deuxième boucle, puis un invariant vérifié en fin de première boucle. En déduire la correction de l'algorithme.
5. On chercher à estimer maintenant la complexité de l'algorithme. Pour cela, on note $T(n)$ le nombre de tests effectués pour trier un tableau de taille n.
   Calculer $T(n)$ dans le meilleur et le pire des cas.

Exercice 5
Une fonction de hachage est une fonction f définie sur l'ensemble des chaînes de caractères et associant à chaque chaîne *str* un entier $f(str)$.

On s'intéresse ici au problème inverse de la fonction de hachage, à savoir : un entier v étant donné, existe-t-il une chaîne *str* telle que $f(str) = v$?

On se place ici dans la cas simplifié où les chaînes ne sont composées que des caractères 0 et 1. On propose l'algorithme récursif suivant, qui prend en entrée la fonction f, un entier v, un entier n, une chaîne de caractère *endstr*, et répond à la question suivante : existe-t-il une chaîne *str* formée de n premiers caractères suivis par *endstr* telle que $f(str) = v$. L'algorithme retourne une chaîne qui convient s'il y en a une, et *False* sinon.

```
def InvEnd(f, v, n, endstr):
    if n == 0:
        return endstr if f(endstr) == v else False
    else:
        result = InvEnd(f, v, n - 1, '0' + endstr)
        if result!=False:
                        return result
        else:
                return InvEnd(f, v, n - 1, '1' + endstr)
```

Pour déterminer s'il existe une chaîne de caractères n ayant pour image v par f, on appelle ensuite la fonction suivante.

```
def InvHach(f, v, n):
        return InvEnd(f, v, n, "")
```

On note $T(n)$ le nombre d'opérations effectuées dans le pire des cas par un appel de $InvHach(f, v, n)$.

1. Montrer que pour tout entier $n \geq 1$, $T(n) = 2T(n-1) + b$ où b est une constante.
2. En déduire une estimation asymptotique de $T(n)$.
   Indication On pourra montrer que la suite U définie par $U(n) = T(n) + b$ est une suite géométrique.

Corrigé 1 **1.** L'algorithme effectue une multiplication à chaque itération de la boucle, et la boucle s'exécute n fois. Ainsi, le nombre total de multiplications effectuées est exactement n. Un équivalent asymptotique est donc $O(n)$.

2. **a.** A chaque appel, l'eposant passé en argument est divisé par au moins 2. On a déduit que cet exposant est majoré par $\frac{n_0}{2^p}$ ou n_0 est la valeur initiale de l'exposant et p le nombre d'appel effectué.
Si $\frac{n_0}{2^p} < 1$, l'algorithme s'arrête ; $\frac{n_0}{2^p} < 1$ étant équivalent à $p > \log_2(n_0)$, l'algorithme esffecteu au plus $\log_2(n_0) + 1$ appel.

b. A chaque appel, on effectue au plus 2 multiplications : le nombre de multiplication est donc en $O(\ln n)$.

Corrigé 2 **1.** On obtient les valeurs suivantes à la fin de chaque boucle.

Étape	n	x	e
Init	13	2	1
1	6	4	2
2	3	16	2
3	1	256	32
4	0	65536	512

Résultat obtenu : $2^{13} = 8192$.

2. L'entier n est positif et décroit strictement à chaque passage dans la boucle, donc c'est un variant de boucle et l'algorithme termine.
3. L'initialisation de l'invariant ne pose pas de problème en calculant la valeur des différentes sommes pour $i = 0$.
   Il reste à pourver que si l'invariant est vrai pour un entier i, il reste vrai poir $i+1$ ce qui ce fait en remarquant que n impair équivait à $b_i = 1$.
4. La boule s'arrête lorsque $n = 0$, ce qui est le cas pour $i = k$. La première ligne de l'invariant affirme alors que le résultat retourné, e, est égal à 2^n.

5. Le nombre k étant le nombre de chiffre de l'écriture de n en base 2, il est inférieur à $\log_2(n)+1$. On effectue au plus deux multiplications à chaque étape, d'où une complexité en $O(\ln n)$.

Corrigé 3 **1.** La variable $b-a$ diminue strictement à chaque itération et est positive : c'est un variant de boucle et l'algorithme termine.

2. On effectue au plus une comparaison de $b-a$ avec 0 et deux comparaisons de $T[c]$ avec elt à chaque itération. Le nombre de comparaisons peut être borné de manière constante.

3. A chaque passage dans la boucle, on cherche si elt est présent dans le sous tableau $T[a:b+1]$. La dimension de ce dernier tableau est divisé par au moins deux à chaque passage. Elle est donc inférieure ou égale à $\frac{n}{2^p}$, où n est la dimension initiale et p est le nombre de passages dans la boucle.

Si $\frac{n}{2^p}<1$, le tableau $T[a:b+1]$ est vide et l'algorithme s'arrête. On effectue donc au plus $\log_2(n)+1$ passages dans la boucle.

Le nombre de tests effectués à chaque passage pouvant être borné par une constane, on a une complexité en $O(\ln n)$.

Corrigé 4 **1.** — Précondition : T tableaux de nombres entiers ou flottants

— Postcondition : T trié en ordre croissant.
De manière plus formelle, $\forall i\in[\![0,len(T)-2]\!], T[i]\leq T[i+1]$.

2. On écrit les valeurs successives de la liste à chaque modification pour les passages successifs dans la boucle `while`.

— Premier passage : [1,3,5,4,2], [1,3,4,5,2], [1,3,4,2,5]
— Deuxième passage : [1,3,2,4,5]
— Troisième passage : [1,2,3,4,5]

Au cours du quatrième passage, la liste n'est pas modifiée et la variable `trie` garde la valeur `True` : le programme s'arrête.

3. Un variant de boucle pour la boucle `while` est $len(T)-i$, donc l'algorithme termine.

4. — Invariant pour la boucle `for` : À la fin de chaque itération de `for`, $T[j]$ est plus grand que tous les éléments de $T[0:j]$

Initialisation La difficulté est que pour une boucle `for`, l'indice j est incrémenté en début de boucle, et non défini avant le premier passage dans la boucle. On peut supposer par convention qu'avant ce premier passage, $j=-1$. Le tableau $T[0:j]$ est alors vide et l'invariant vérifié.

Hérédité Supposons l'invariant vérifié à la fin de la boucle d'indice j : . Au début de la boucle suivante, j prend la valeur $j+1$ et si $T[j]<T[j+1]$, ces deux éléments sont inversés, si bien que le plus grand des deux devient l'élément du tableau d'indice $j+1$. On en déduite que $T[j+1]$ est plus grand que tous les éléments de $T[0:j+1]$ et l'invariant est vérifié à l'indice $j+1$.

— Invariant pour la boucle `while` : à la fin de la boucle d'indice i, $T[len(T)-i:]$ est ordonné et contient les i plus grand élément du tableau initial.

Initialisation Avant le premier passage dans la boucle, $i=0$ et le tableau $T[len(T)-i:]$ est vide. L'invariant est évidemment vérifié.

Hérédité Supposons l'invariant vérifié pour i. Lors du passage suivant dans la boucle `while`, la boucle `for`, d'après l'invariant de fin de cette boucle, place à l'indice $len(T)-1-i$ le plus grand élément de $T[0,len(T)-1-i]$. Cet élément est par ailleurs plus petit que tous les éléments de $T[len(T)-i:]$ d'après l'invariant de la boucle `while` à l'indice i. On en déduit que le tableau $T[len(T)-1-i:]$ est ordonné et contient les $i+1$ plus grand élément du tableau initial, c'est-à-dire que l'invariant est vérifié à l'indice $i+1$.

5. Complexité dans le pire des cas (tableau trié dans l'ordre inverse) Chaque passage de la boucle `for` effectue une comparaison (et un échange) dans la liste. On a $n-1-i$ passage dans cette boucle.

Le nombre total d'échanges est donné par la somme des termes d'une suite arithmétique :

$$\sum_{i=0}^{n-1} n-1-i=\frac{n(n-1)}{2}$$

La complexité est donc une complexité quadratique en $O(n^2)$.

Complexité dans le meilleur des cas (tableau déjà trié) On effectue un seul passage dans la boucle `while`. La boucle `for` effectue $n-1$ tests d'où une complexité linéaire en $O(n)$.

Corrigé 5 **1.** A chaque appel, l'algorithme effectue un nombre fini, majorable indépendamment de n, d'opérations (comparaisons, concaténations de liste), et au pire deux appels de T pour un troisième argument égal à $n-1$ et les chaînes obtenues en ajoutant au début de *endstr* les caractères 0 et 1. On en déduite qu'il existe un entier b tel que $T(n) = 2T(n-1) + b$.

2. On a $U(n) = T(n) + b = 2T(n-1) + 2b = 2U(n)$ donc la suite U est géométrique de raison 2, et pour tout entier n, $U(n) = 2^n U(0)$. On en déduite que $T(n) \sim 2^n$, donc qu'on a une complexité exponentielle.

3. Exponentielle, $O(2^n)$, confirmée par le fait que chaque appel produit deux sous-appels.