

ANNEXE

A

RAPPELS SUR LA NOTION DE COMPLEXITÉ

I - Comment quantifier la complexité?

On dispose de plusieurs façons d'évaluer le coût et l'efficacité d'un algorithme.

- La *complexité spatiale* : on évalue la place que vont occuper dans la mémoire de l'ordinateur toutes les variables utiles à l'algorithme.
- La *complexité temporelle* : on cherche à connaître le temps d'exécution de l'algorithme. C'est essentiellement sur cette information que l'on travaille.

Comme cette durée dépend de la machine sur laquelle on lance le programme, on s'attache à l'évaluation du nombre d'«opérations élémentaires» effectuées par l'ordinateur pour arriver au résultat. Cependant, cette notion dépend du problème considéré : dans un calcul numérique, par exemple, on compte les opérations les plus coûteuses (multiplications, extractions de racines, utilisation du logarithme ou de l'exponentielle); dans un problème de recherche dans une liste, par exemple, on peut compter les références à des emplacements de mémoire ou les permutations d'éléments.

Ces grandeurs dépendent du ou des paramètre(s) d'entrée passé(s) à la fonction ; selon les cas, le temps de calcul peut être très différent et on essaie de tirer les informations importantes des différents cas possibles : la complexité dans le meilleur des cas ; la complexité moyenne (souvent plus dure à évaluer, cela repose sur un calcul probabiliste) ; la complexité dans le pire des cas.

De plus, l'information vraiment importante est le comportement asymptotique de cette durée.

On rappelle que si (u_n) et (v_n) sont deux suites de nombres réels strictement positifs, on dit que :

- la suite (u_n) est **dominée** par la suite (v_n) s'il existe un réel $M > 0$ tel qu'à partir d'un certain rang, $u_n \leq Mv_n$ (autrement dit, la suite (u_n/v_n) est majorée), on note $u_n = O(v_n)$;
- la suite (u_n) **domine** la suite (v_n) si la suite (v_n) est dominée par la suite (u_n) , on note $u_n = \Omega(v_n)$;
- les suites (u_n) et (v_n) sont **de même ordre de grandeur** si chacune est dominée par l'autre, on note $u_n = \Theta(v_n)$;
- les suites (u_n) et (v_n) sont **équivalentes** si le quotient (u_n/v_n) converge vers 1, on note $u_n \sim v_n$.

Les principales classes de complexité sont :

- **complexité constante** : en $O(1)$ – le temps d'exécution est borné indépendamment de la taille n des données;
- **complexité logarithmique** : en $O(\ln(n))$ – les algorithmes de cette classe sont très efficaces puisque l'on double le temps d'exécution en élevant au carré la taille des données (notons que l'on peut remplacer $\ln(n)$ par $\log_2(n)$ ou $\log_{10}(n)$);
- **complexité linéaire** : en $O(n)$ – les algorithmes de cette classe sont efficaces;
- **complexité quasi-linéaire** (ou log-linéaire, ou linéarithmétique) : c'est en $O(n \ln(n))$ (mais pas un $O(n)$) – ces algorithmes sont en pratique souvent aussi efficaces que les algorithmes linéaires (pour s'en convaincre, on peut remarquer que $\ln(10^{15}) \approx 35$);
- **complexité polynomiale** : en $O(n^k)$ avec $k > 1$ – ces algorithmes sont moyennement efficaces pour $k \leq 3$, inefficaces sinon (dans le cas $k = 2$, on parle de *complexité quadratique*);
- **complexité exponentielle** : c'est un $O(a^n)$ avec $a > 1$ – ces algorithmes sont inexploitable sauf pour de très petites valeurs des données.

Comparons les temps de calcul approximatifs correspondant à ces types de complexité en se fondant sur la rapidité d'un ordinateur de bureau standard (source Wikipedia).

nom	ordre	$n = 10$	$n = 50$	$n = 1000$	$n = 1000000$
logarithmique	$\ln(n)$	10 ns	20 ns	30 ns	60 ns
linéaire	n	100 ns	500 ns	10 μ s	10 ms
quasi-linéaire	$n \ln(n)$	100 ns	850 ns	30 μ s	60 ms
quadratique	n^2	1 μ s	25 μ s	10 ms	2,8 h
cubique	n^3	10 μ s	1,25 ms	10 s	316 ans
exponentielle	2^n	10 μ s	130 jours		
factorielle	$n!$	36 ms	10^{48} ans		

II - Des exemples

II.1 - Est-ce que le premier caractère d'une chaîne est un a ?

La fonction suivante a pour entrée une chaîne de caractères et pour sortie un booléen indiquant si le premier caractère est un a.

```
def estCeUnA(ch: str) -> bool:
    premier = ch[0]
    return premier == 'a' or premier == 'A'
```

Quelle que soit la taille n de la chaîne de caractères ch , il y a deux ou trois opérations : la consultation du premier caractère et l'évaluation de booléens. La complexité est donc en $O(1)$.

II.2 - Calcul de factorielle

Le calcul de $n!$ peut être programmé de façon itérative.

```
def factorielle(n: int) -> int:
    """ entrée: n int avec n>=0
        sortie: valeur de n! (int) """
    f = 1
    for k in range(2, n+1):
        f = f * k
    return f
```

Comptons le nombre $C(n)$ d'opérations (arithmétiques, affectations, tests) :

- il y a une première affectation $f=1$;
- si $n \leq 1$ alors c'est tout;
- sinon, il y a ensuite $n - 1$ itérations (k va de 2 à n) et, pour chaque itération, il y a une multiplication et une affectation.

On a donc $C(0) = C(1) = 1$ et, pour $n \geq 2$, $C(n) = 1 + 2(n - 1) = 2n - 1$.

On peut aussi programmer la factorielle de façon récursive.

```
def factorielle_bis(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorielle_bis(n-1)
```

Pour l'appel `factorielle_bis(n)`, il y a un test puis (éventuellement) une multiplication et toutes les opérations liées au calcul de `factorielle_bis(n-1)`. On a donc :

$$C(0) = C(1) = 1 \text{ et : } \forall n \geq 2, C(n) = 2 + C(n-1)$$

d'où :

$$C(0) = C(1) = 1 \text{ et : } \forall n \geq 2, C(n) = 2(n - 1) + 1 = 2n - 1.$$

Que ce soit de manière itérative ou récursive, la complexité est donc linéaire.

II.3 - Calcul d'une somme d'entiers

On écrit une fonction d'argument $n \in \mathbb{N}$ qui renvoie la somme des entiers impairs inférieurs ou égaux à n .

```
def somme(n: int) -> int:
    """ entrée: n int
        énumère les entiers jusqu'à n et, s'ils sont impairs,
        on les somme dans une variable s
        sortie: la somme contenue dans s (int) """
    s = 0
    for k in range(n+1):
        if k%2 == 1:
            s = s + k
    return s
```

Comptons le nombre $C(n)$ d'opérations (arithmétiques, affectations, tests) :

- il y a une première affectation ($s=0$);
- il y a ensuite $n + 1$ itérations (k va de 0 à n) et, pour chaque itération, il y a un test et, éventuellement, une addition et une affectation.

On a donc $1 + (n + 1) \leq C(n) \leq 1 + 3(n + 1)$ i.e. $n + 2 \leq C(n) \leq 3n + 4$.

On a donc une complexité linéaire avec $C(n) = \Theta(n)$.

II.4 - Calcul statistique sur les termes d'une liste

Considérons une fonction `distance` prenant en argument une liste `L` de nombres de type `float` et qui calcule la somme des écarts entre les éléments de `L` et leur moyenne.

```
def distance(L: [float]) -> float:
    """ entrée: liste L de float
        sortie: somme (float) des écarts entre les termes de L et leur moyenne
    """
    m = 0
    for nb in L:
        m = m + nb
    m = m / len(L)
    s = 0
    for nb in L:
        s = s + abs(nb - m)
    return s
```

Notons $C(n)$ le nombre d'opérations dans le cas où la liste passée en argument contient n éléments. Il y a :

- une première affectation pour m ;
- une boucle parcourant les éléments de L avec une addition et une affectation à chaque fois;
- une division et deux affectations pour m et s ;
- une deuxième boucle parcourant les éléments de L avec une addition, une soustraction, une valeur absolue et une affectation à chaque fois.

On a donc $C(n) = 3 + n \times (2 + 4) = 3 + 6n$. La complexité est linéaire.

II.5 - Recherche du maximum dans une liste

La fonction suivante renvoie le maximum de la liste de nombres passée en argument.

```
def recherche_max(L: [float]) -> float:
    M = L[0]
    for x in L:
        if x > M:
            M = x
    return M
```

Si n est la longueur de la liste alors il y a une affectation puis une boucle de n itérations avec, à chaque itération, un test et (éventuellement) une affectation donc le nombre d'opérations vérifie :

$$1 + n \leq C(n) \leq 1 + 2n$$

On a donc une complexité linéaire puisque $C(n) = \Theta(n)$.

Notons que le cas où $C(n) = n + 1$ est celui où la valeur $L[0]$ est la plus grande (*meilleur cas*) et le cas où $C(n) = 2n + 1$ est celui où la liste est classée dans l'ordre strictement croissant et où, à chaque étape, on rencontre un nouveau «candidat maximum» (*pire cas*).

II.6 - Une liste scindée en deux

On considère une fonction `separe` prenant en argument une liste de nombres entiers et qui renvoie un couple de listes : la première formée par les nombres pairs de la liste initiale et la seconde formée par les nombres impairs.

```
def separe(L: List) -> (List, List):
    LP, LI = [], []
    for element in L:
        if element%2 == 0:
            LP.append(element)
        else:
            LI.append(element)
    return LP, LI
```

Si n est la longueur de la liste L alors il y a deux affectations puis une boucle de n itérations avec, à chaque itération, un test et une utilisation de la méthode `append`. La difficulté vient du fait qu'il est alors indispensable de connaître la complexité de ladite méthode.

La complexité de la méthode `append` est subtile (il s'agit d'une complexité dite « $O(1)$ amortie») et, dans les sujets de concours, il est souvent considéré en simplifiant que cette méthode est de coût constant ce qui donnerait alors dans l'exemple une complexité linéaire.

II.7 - Fusion de deux listes triées

La fonction suivante prend en arguments deux listes d'entiers triées et renvoie une liste triée obtenue en réunissant les éléments de l'une et de l'autre.

```
def fusion(L1: [int], L2: [int]) -> [int]:
    """ entrée : L1 et L2 listes d'entiers triées par ordre croissant
        sortie : L liste triée par ordre croissant formée par les éléments de L1 et L2
    """
    L = []
    i, j = 0, 0
    while i < len(L1) and j < len(L2):
        if L1[i] < L2[j]:
            L.append(L1[i])
            i = i+1
        else:
            L.append(L2[j])
            j = j+1
    if i == len(L1):
        for k in range(j, len(L2)):
            L.append(L2[k])
    else:
        for k in range(i, len(L1)):
            L.append(L1[k])
    return L
```

Par exemple, on a :

```
>>> fusion([1, 5, 9], [3, 4, 8, 12, 15, 21, 42])
[1, 3, 4, 5, 8, 9, 12, 15, 21, 42]
```

Déterminons le nombre total d'opérations :

- o on commence par trois affectations ;
- o il y a ensuite une boucle `while` et, à chacune de ces itérations, il y a un test suivi d'un `append`, d'une addition et d'une affectation ;
- o il y a ensuite un test suivi d'une boucle telle qu'à chacune des itérations il y ait un `append` ;
- o le nombre total d'itérations (`while` et `for`) est $\text{len}(L1) + \text{len}(L2)$ puisque chaque itération correspond à l'affectation d'un élément de $L1$ ou $L2$.

Considérons pour simplifier que `append` soit de coût constant. Il y a donc, en notant respectivement p et q les longueurs de $L1$ et $L2$, un nombre d'opérations majoré par :

$$4 + (p + q) \times 4$$

ce qui donne une expression en $O(p + q)$.

II.8 - Plus proches voisins

On considère une fonction `plus_proches_voisins` prenant en argument une liste de n entiers et renvoyant un triplet d'entiers (a, b, c) tel que l'écart minimal entre deux entiers de la liste soit égal à c et soit réalisé entre les éléments d'indices a et b .

```
def plus_proches_voisins(lis: [int]) -> (int, int, int):
    a, b = 0, 1
    val = abs(lis[1] - lis[0])
    for i in range(1, len(lis)):
        for j in range(i):
            if abs(lis[j] - lis[i]) < val:
                val = abs(lis[j] - lis[i])
                a, b = j, i
    return a, b, val
```

Par exemple, on a :

```
>>> plus_proches_voisins([365, 384, 425, 681, 1000, 394, 32, 96, 377, 87])
(1, 8, 7)
```

Notons $C(n)$ le nombre total d'opérations (en négligeant l'accès aux éléments des listes) :

- il y a une différence, une valeur absolue, trois affectations;
- puis une boucle d'indice i avec $n - 1$ itérations qui contient une boucle avec i itérations;
- à chaque itération de cette dernière, il y a deux opérations et un test suivi éventuellement de 5 opérations.

Le nombre total d'itérations est $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$ et le nombre d'opérations par itération est borné donc, globalement, on a $C(n) = O(n^2)$ et la complexité est quadratique.

II.9 - Tri à bulles

Considérons le cas du tri à bulles d'une liste de nombres.

```
def tri_bulles(L: List) -> None:
    """ entrée: liste de nombres (int ou float)
        effet de bord: tri de la liste par ordre croissant
    """
    n = len(L)
    for i in range(n):
        for j in range(0, n-i-1):
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]
```

Il y a n itérations pour la première boucle et l'itération d'indice i induit elle-même $n - i - 1$ itérations, le nombre total d'itérations est donc :

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

Au sein de chacune de ces itérations, il y a un test et, éventuellement, une inversion (ou deux affectations).

La complexité est donc quadratique.

II.10 - Produit matriciel

Considérons deux matrices A et B de $\mathcal{M}_n(\mathbb{R})$. Le produit de A et B est défini pour tout $(i, j) \in \llbracket 1, n \rrbracket^2$ par :

$$(AB)[i, j] = \sum_{k=1}^n A[i, k]B[k, j].$$

La programmation du calcul à l'aide de cette formule induit trois boucles imbriquées avec n itérations chacune (deux pour parcourir tous les emplacements puis une pour calculer la somme). À chaque itération, il y a une multiplication et une addition. La complexité est donc en $O(n^3)$.

II.11 - Exponentiation rapide

La méthode naïve pour calculer x^n (où $x \in \mathbb{R}$ et $n \in \mathbb{N}$) consiste à faire $n - 1$ multiplications ce qui signifie que l'on a une complexité linéaire.

Le principe de l'exponentiation rapide repose sur la constatation suivante :

$$x^n = \begin{cases} x & \text{si } n = 1 \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x \cdot (x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases}$$

Voici une façon de programmer cela de façon récursive :

```
def expor_rec(x: float, n: int) -> float:
    if n == 0: # cas de base
        return 1
    elif n == 1: # autre cas de base
        return x
    elif n%2 == 0:
        return expor_rec(x*x, n//2)
    else:
        return x*expor_rec(x*x, n//2)
```

Notons $C(n)$ le nombre de multiplications effectuées lors de l'appel $\text{expor_rec}(x, n)$.

On a $C(0) = 0$ et $C(1) = 0$. Pour $n \geq 2$, on a :

$$C(n) = \begin{cases} 1 + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{si } n \text{ est pair} \\ 2 + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{si } n \text{ est impair} \end{cases}.$$

On considère la suite $(\widetilde{C}(n))_{n \geq 1}$ donnée par :

$$\begin{cases} \widetilde{C}(1) = 0 \\ \forall n \geq 2, \widetilde{C}(n) = 2 + \widetilde{C}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \end{cases}.$$

Par un raisonnement par récurrence (forte), on obtient aisément la croissance de la suite $(\widetilde{C}(n))_{n \geq 1}$ et le fait que :

$$\forall n \geq 1, C(n) \leq \widetilde{C}(n).$$

Soit $n \geq 2$, il existe un unique entier p tel que l'on ait $2^p \leq n < 2^{p+1}$ (en fait $p = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor$) donc :

$$\widetilde{C}(2^p) \leq \widetilde{C}(n) \leq \widetilde{C}(2^{p+1}).$$

On remarque que, pour tout $m \geq 0$:

$$\widetilde{C}(2^{m+1}) = 2 + \widetilde{C}(2^m),$$

ce qui signifie que la suite $(\widetilde{C}(2^m))_{m \geq 0}$ est arithmétique de raison 2 donc :

$$\forall m \geq 1, \widetilde{C}(2^m) = 2m + \widetilde{C}(2^0) = 2m + 0.$$

On en déduit en revenant à n :

$$2p \leq \widetilde{C}(n) \leq 2p + 2,$$

puis :

$$C(n) \leq 2 \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor + 2$$

donc il existe une constante $K > 0$ telle que l'on ait pour n assez grand :

$$C(n) \leq K \ln(n),$$

ce qui signifie que $C(n) = O(\ln(n))$.

II.12 - Tri fusion

Le principe de ce tri, dans le cas d'une liste L de n entiers, repose sur une approche récursive et sur une stratégie du type *diviser pour régner* :

- on coupe la liste en deux sous-listes de longueurs (à peu près) égales ;
- on trie les deux sous-listes ;
- on fusionne les deux sous-listes triées en une liste triée.

Pour programmer cela, on utilise la fonction de fusion de deux listes triées vue plus haut ainsi que la fonction de tri proprement dite :

```
def triFusion(L: [int]) -> None:
    n = len(L)
    if n <= 1:          # on finit par considérer une liste vide ou à 1 élément
        return L
    else:
        m = n//2
        l1 = triFusion(L[:m])
        l2 = triFusion(L[m:])
        return(fusion(l1, l2))
```

Notons $C(n)$ le nombre (dans le pire des cas) de comparaisons et affectations utiles pour trier ainsi une liste de longueur n . On a alors :

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \text{« complexité de la fusion »},$$

donc, d'après ce qui a été vu :

$$C(n) \leq C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n.$$

Pour simplifier, supposons qu'il s'agisse d'une égalité. On a tout d'abord pour tout entier $k \in \mathbb{N}$:

$$C(2^{k+1}) = 2C(2^k) + 2^{k+1},$$

d'où :

$$\frac{C(2^{k+1})}{2^{k+1}} = \frac{C(2^k)}{2^k} + 1,$$

ce qui conduit pour tout $k \in \mathbb{N}$ à :

$$\frac{C(2^k)}{2^k} = \frac{C(2^0)}{2^0} + k,$$

puis :

$$C(2^k) = 2^k C(1) + k 2^k.$$

Il convient ensuite de vérifier à l'aide d'un raisonnement par récurrence (forte) que la suite $(C(n))_{n \in \mathbb{N}}$ est croissante.

Considérons alors un entier n que l'on encadre entre deux puissances de 2 :

$$2^k \leq n < 2^{k+1} \quad (i.e. \ k = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor).$$

cela conduit alors (du fait de la croissance de la suite $(C(n))_{n \in \mathbb{N}}$) à :

$$C(2^k) \leq C(n) < C(2^{k+1}),$$

soit :

$$2^k C(1) + k 2^k \leq C(n) < 2^{k+1} C(1) + (k+1) 2^{k+1}.$$

Enfin, il existe une constante $c > 0$ telle que l'on puisse écrire :

$$k2^k = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor 2^{\left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor} \leq cn \ln(n).$$

On a donc :

$$C(n) = O(n \ln(n)).$$

III - Quelques exercices

Exercice 1

Au rugby, on peut marquer un drop ou une pénalité ce qui rapporte 3 points, un essai ce qui rapporte 5 points ou un essai transformé ce qui rapporte 7 points. Par conséquent un total de points peut être réalisé de plusieurs façons ; par exemple on peut marquer 15 points avec 3 essais non transformés, ou 1 essai de chaque type et une pénalité, etc.

Écrire une fonction `possibilites(n)` qui donne le nombre de façons d'obtenir le score n (sans distinguer drops et pénalités).

Quelle est la complexité de cette fonction ?

Exercice 2

Déterminer la complexité dans le pire cas puis dans le meilleur cas du tri par sélection.

Exercice 3

Comment pourrait-on améliorer le tri à bulles afin d'éviter de parcourir une fin de liste déjà triée et, en particulier, de façon à obtenir une complexité linéaire dans le meilleur des cas ?

Exercice 4

On cherche à calculer le gain maximum possible à la Bourse sur une action pendant une période de n jours, en ne faisant qu'une opération d'achat ou de vente d'une seule action. On suppose que les cours quotidiens de cette action sont enregistrés dans une liste d'entiers naturels ($a_i \in \mathbb{N}$) de n éléments ($0 \leq i < n$).

Le gain maximum G est :

$$G = \max_{0 \leq i \leq j < n} (a_j - a_i).$$

1. Écrire une fonction Python qui calcule G en suivant textuellement la définition ci-dessus. Quelle est sa complexité ?
2. Pour tout $i \in \llbracket 0, n-1 \rrbracket$, on définit le gain courant G_i comme le gain possible en vendant son action au temps i , c'est-à-dire :

$$G_i = \max_{0 \leq k \leq i} (a_i - a_k).$$

On vérifie que l'on a la relation :

$$G_i = \max\{0 ; G_{i-1} + a_i - a_{i-1}\}.$$

En déduire une fonction qui calcule G avec une complexité linéaire.

Correction des exercices

Exercice 1

```
def possibilites(n: int) -> int:
    p, e, et = n//3, n//5, n//7
    nb = 0
    for i in range(p+1):
        for j in range(e+1):
            for k in range(et+1):
                if 3*i + 5*j + 7*k == n:
                    nb += 1
    return nb

""" Il y a (n//3 + 1) * (n//5 + 1) * (n//7 + 1) itérations
    avec à chaque fois un test et éventuellement une addition et une affectation.
    La complexité est donc en  $O(n^3)$ .
"""
```

Exercice 2

```
def triSelection(L):
    """ entrée : liste L d'éléments comparables (en général int)
        effet de bord : tri de la liste par ordre croissant
        pas de sortie
    """
    for k in range(len(L) - 1):
        ind = k
        for i in range(k+1, len(L)):
            if L[i] < L[ind]:
                ind = i
        L[k], L[ind] = L[ind], L[k]

    """ Notons n la longueur de L.
        La première boucle induit n-1 itérations indexées par k.
        À chacune de ces itérations, il y a 3 affectations plus une boucle.
        La boucle intérieure comporte n-(k+1) itérations
            avec à chaque fois un test et éventuellement une affectation.
        La complexité est donc (dans tous les cas) en  $O(n^2)$ .

        En revanche, le nombre d'échanges est en  $O(n)$ .
        Ce tri est intéressant dans le cas d'éléments faciles à comparer
            mais dont la permutation est coûteuse.
    """
```

```
## Exercice 3

""" Si lors d'un tour de la boucle intérieure il n'y a aucun échange
    alors c'est que la liste est triée.
    On pourrait donc mettre une variable qui teste s'il y a eu au moins un échange.
"""

def tri_bulles_bis(liste):
    """ entrée: liste de nombres (int ou float)
        effet de bord: tri de la liste par ordre croissant
        sortie: None
    """
    n = len(liste)
    i = 0
    test = True
    while test and i < n:
        test = False
        for j in range(n-i-1):
            if liste[j] > liste[j+1]:
                liste[j], liste[j+1] = liste[j+1], liste[j]
                test = True
        i += 1

    """ Dans le meilleur des cas, la liste est déjà triée et on n'effectue
        qu'un seul passage dans la boucle while
        donc la boucle for est effectuée une fois (pour i=0) avec n-1 itérations.
        Donc la complexité est alors O(n).
    """

## Exercice 4

def gain(a):
    g = 0
    for i in range(len(a)):
        for j in range(i+1, len(a)):
            if a[j]-a[i] > g:
                g = a[j] - a[i]
    return g

def gain_bis(a):
    gc = 0 # gain courant
    g = 0 # gain i.e. le max des gains courants
    for i in range(1, len(a)):
        if gc + a[i] - a[i-1] > 0:
            gc = gc + a[i] - a[i-1]
        else:
            gc = 0
        if gc > g:
            g = gc
    return g
```