Complexité algorithmique

Avant-propos :

Cette feuille d'exercices a pour objectif d'introduire la notion de complexité en algorithmique. Nous nous limiterons à l'exposition de quelques principes généraux et à des calculs de complexité (exacte ou par comparaison asymptotique i.e. notations O et Θ).

1 Principes

Au cours de ce TD, nous imaginerons, implémenterons et manipulerons des algorithmes, c'est-à-dire des séquences d'opérations ayant pour but la résolution d'un problème via le calcul de données de sortie en fonction de certaines entrées.

A savoir : Algorithme

Une définition possible d'un algorithme est la suivante : un algorithme consiste en une transformation d'un nombre fini d'entrées en un nombre fini de sorties par le biais d'une suite finie d'opérations. Dans cette définition généraliste, une recette de cuisine peut être considérée comme un algorithme.

Cependant nous nous intéresserons dans ce cours à des algorithmes numériques non probabilistes, c'est-à-dire exprimés de manières non ambiguë, comportant uniquement des opérations de nature déterministe, et manipulant des données numériques (nombres ou structures de données plus ou moins complexes construites par induction).

Afin d'exprimer ces algorithmes dans un langage universellement compréhensible et généralisable, nous utiliserons une syntaxe de pseudo-code (qui est interchangeable avec n'importe quel autre pseudo-code et uniquement question de convention).

A savoir : Pseudo-Code

Un pseudo-code est une syntaxe permettant de construire des représentations non ambiguës d'algorithmes. Un pseudo-code n'est pas un langage de programmation ; c'est une syntaxe qui se veut en fait indépendante de tout langage de programmation et dont le but est de pouvoir servir de schéma directeur pour l'implémentation dudit algorithme dans n'importe quel langage de programmation adapté au problème donné.

Aussi, le pseudo-code peut se rapprocher davantage du langage naturel, facilitant ainsi son écriture et sa compréhension. De plus, il a l'avantage de s'abstraire de toute complication ou difficulté d'expression liées à un langage d'implémentation donné.

Il n'y a pas de convention quant à la syntaxe à adopter pour écrire du pseudo-code ; en pratique on préfère utiliser des symboles mathématiques divers pour les opérations simples ($=$ ou $\leftarrow$ pour l'assignation, $=$ ou $==$ pour le test d'égalité, $\neq$ ou $!=$ pour le test d'inégalité etc.), des mots anglais comme mots clés pour désigner des concepts propres à l'algorithmique (boucles avec **while**, **for**, et **break** pour s'en extraire, **if**, **then**, **else** pour les conditionnelles, **return** pour signifier la terminaison de l'algorithme avec renvoi éventuel de données etc.

Voici un exemple de pseudo-code tel que nous allons l'utiliser dans la suite :

```
1: procedure F1(n ∈ ℕ⁺)
2:     for i ∈ [1, n] by 1 do
3:         for j ∈ [i, n] by 1 do
4:             Afficher ("Salut")
```

Notez bien que cela n'est pas directement du code Python. On peut voir cela dès la première ligne : en Python, pour définir une nouvelle fonction, on n'utilise pas le mot-clé "procedure" mais le mot-clé `def`.

Un algorithme résout un problème en utilisant une série d'opérations sur un ensemble de données qui peuvent être crées, modifiées, ou détruites et qui sont stockées tant qu'elle ne sont pas détruites. Toutefois à résolution

égale, un algorithme peut être plus "performant" qu'un autre ; cette "performance" se mesure par la quantité de "ressources" qui est utilisée, définie par la complexité en espace et la complexité en temps.

A savoir : Complexité des algorithmes

La complexité en temps correspond au nombre d'opérations élémentaires (assignations, tests d'ordonnancement, parcours de structures, opérations arithmétiques (additions, soustractions etc.)) nécessaires pour que l'algorithme complète sa tâche. Cette complexité est calculée récursivement pour des algorithmes qui en appellent d'autres.

La complexité en espace correspond au nombre maximum de données stockées simultanément par l'algorithme au cours de son exécution. Ceci corresponds donc à "l'espace mémoire" requis pour faire fonctionner l'algorithme correctement.

La complexité (dans les 2 cas) est généralement donnée en fonction d'une estimation entière de la taille des données d'entrées (notée n). Dans ce cours, on la notera $T : \mathbb{N} \to \mathbb{N}$. Le nombre $T(n)$ représente la complexité (en temps ou en espace, selon le cas) pour une entrée de taille n.

La complexité (dans les 2 cas) peut s'exprimer en moyenne, dans le pire des cas (vis-à-vis d'une propriété des données d'entrées ayant été identifiée comme donnant un maximum global de complexité à taille fixe) ou bien dans le meilleur des cas (vis-à-vis d'une propriété des données d'entrées ayant été identifiée comme donnant un minimum global de complexité à taille fixe).

La complexité (dans les 2 cas) peut être donnée comme une fonction exacte de la taille ou bien asymptotiquement.

La complexité est généralement interprétée à partir du pseudo-code d'un algorithme.

Cette "performance" algorithmique qu'est la complexité se retrouve aussi traduite dans les "performances" des implémentations concrètes correspondantes. Considérons par exemple 2 algorithmes résolvant le même problème mais de complexité inégales. Alors, étant donné un même langage de programmation et une même compétence du développeur, on aura presque systématiquement, pour une même donnée d'entrée:

- Un temps de calcul plus long pour l'implémentation correspondant à l'algorithme ayant la pire complexité en temps
- Une utilisation de la mémoire vive (RAM) accrue pour l'implémentation correspondant à l'algorithme ayant la pire complexité en espace

En pratique, pour calculer la complexité en temps d'un algorithme, on utilise l'une des deux méthodes suivantes :

Algorithme itératif. Si l'algorithme est itératif, on calcule $T(n)$ en comptant le nombre de fois que l'on passe dans chaque boucle. Un passage dans une boucle correspond à une somme. Deux boucles imbriquées correspondent à deux sommes imbriquées, etc. Par exemple :

```
1 def iteratif(n):
2     for i in range(0, n):
3         for j in range(0, 2*n):
4             for k in range(n, 5*n):
5                 print("Un passage dans la boucle compte pour 1")
```

On écrit

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{2n-1} \sum_{k=n}^{5n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{2n-1} 4n = \sum_{i=0}^{n-1} 2n \times 4n = n \times 2n \times 4n = 8n^3$$

Algorithme récursif. Si l'algorithme est récursif, on calcule $T(n)$ en utilisant des relations de récurrence. Les cas de base de l'algorithme correspondent à l'initialisation de la suite (i.e. souvent la valeur de $T(0)$, voire $T(1)$), et les cas récursifs de l'algorithme correspondent à la relation de récurrence. Par exemple :

```
1 def recursif(n):
2     if n == 0:
```

```
        return 0
    else:
        return (recursif(n-1) + n)*3
```

On écrit $T(0) = 0$ (aucune opération n'est effectuée dans le cas de base puisqu'on renvoie juste 0), et $T(n+1) = T(n) + 2$ (calculer le résultat pour $n+1$ requiert de calculer le résultat pour n ainsi que de faire deux opérations élémentaires, une addition et une multiplication). On en déduit facilement que $T(n) = 2n$ pour tout $n \in \mathbb{N}$.

2 Compréhension d'algorithmes

Exercice 1 : Algorithme mystère 😎

Nous étudierons l'algorithme donné ci-dessous.

procedure MYSTERE($n \in \mathbb{N}$)
  if $n == 0$ **then**
    return 0
  else
    return MYSTERE($n-1$) + $2n-1$

Question 1: Implémentez l'algorithme en Python et testez-le.

Question 2: Que fait cet algorithme ?

```
def mystere(n):
    if n == 0:
        return 0
    else :
        return mystere(n-1) + 2*n-1

def test_mystere(n):
    for i in range(n):
        print(mystere(i))

test_mystere(10)
```

On obtient la suite suivante dans la console : $0, 1, 4, 9, 16, 25, 36, 49, 64$. Cet algorithme permet de calculer le carré d'un entier. Cela peut se prouver formellement par récurrence:

- $mystere(0) = 0 = 0^2$ donc le résultat est vrai au rang 0.
- $mystere(1) = 1 = 1^2$ donc le résultat vrai au rang 1.

Supposons maintenant, pour $n \in \mathbb{N}$ que $mystere(n) = n^2$. Alors:

$$\begin{aligned} mystere(n+1) &= mystere(n) + 2 * (n+1) - 1 \\ &= n^2 + 2 * n + 1 \\ &= (n+1)^2 \end{aligned}$$

Donc $mystere(n) = n^2$ pour tout $n \in \mathbb{N}$.

Exercice 2 : Algorithme mystère 2 😀

Considérons l'algorithme donné ci-dessous.

```
1:  procedure MYSTERE(T ∈ Tableau)
2:      n ← T.length
3:      B ← [True] * n                    ▷ Creates a list of size n filled with True
4:      for i ∈ [1, n] by 1 do
5:          for j ∈ [i + 1, n] by 1 do
6:              if T[i] > T[j] then
7:                  B[j] ← False
8:              if T[j] > T[i] then
9:                  B[i] ← False
10:     k ← 1
11:     while not(B[k]) do
12:         k → k + 1
13:     return k, B
```

Question 1: Détaillez l'exécution de cet algorithme sur l'instance suivante : $[1, -4, 4, 2, -4, 4, 1, 3]$.

Question 2: Implémentez l'algorithme suivant en Python et testez-le. Vérifiez notamment la réponse de la question précédente.

Question 3: Que fait cet algorithme ?

Détails de l'exécution en considérant les indices en Python (premier indice 0).

```
i: 0 j: 1 B: [True, False, True, True, True, True, True, True]
i: 0 j: 2 B: [False, False, True, True, True, True, True, True]
i: 0 j: 3 B: [False, False, True, True, True, True, True, True]
i: 0 j: 4 B: [False, False, True, True, False, True, True, True]
i: 0 j: 5 B: [False, False, True, True, False, True, True, True]
i: 0 j: 6 B: [False, False, True, True, False, True, True, True]
i: 0 j: 7 B: [False, False, True, True, False, True, True, True]
i: 1 j: 2 B: [False, False, True, True, False, True, True, True]
i: 1 j: 3 B: [False, False, True, True, False, True, True, True]
i: 1 j: 4 B: [False, False, True, True, False, True, True, True]
i: 1 j: 5 B: [False, False, True, True, False, True, True, True]
i: 1 j: 6 B: [False, False, True, True, False, True, True, True]
i: 1 j: 7 B: [False, False, True, True, False, True, True, True]
i: 2 j: 3 B: [False, False, True, False, False, True, True, True]
i: 2 j: 4 B: [False, False, True, False, False, True, True, True]
i: 2 j: 5 B: [False, False, True, False, False, True, True, True]
i: 2 j: 6 B: [False, False, True, False, False, True, False, True]
i: 2 j: 7 B: [False, False, True, False, False, True, False, False]
i: 3 j: 4 B: [False, False, True, False, False, True, False, False]
i: 3 j: 5 B: [False, False, True, False, False, True, False, False]
i: 3 j: 6 B: [False, False, True, False, False, True, False, False]
i: 3 j: 7 B: [False, False, True, False, False, True, False, False]
i: 4 j: 5 B: [False, False, True, False, False, True, False, False]
i: 4 j: 6 B: [False, False, True, False, False, True, False, False]
i: 4 j: 7 B: [False, False, True, False, False, True, False, False]
```

```
i: 5 j: 6 B: [False, False, True, False, False, True, False, False]
i: 5 j: 7 B: [False, False, True, False, False, True, False, False]
i: 6 j: 7 B: [False, False, True, False, False, True, False, False]
```

```python
def mystere(l):
    n = len(l)
    B = [True]*n
    for i in range(n):
        for j in range(i+1,n):
            if l[i] > l[j]:
                B[j] = False
            if l[j] > l[i]:
                B[i] = False
    k = 0
    while not(B[k]):
        k = k+1
    return k,B
```

$print(mystere([1,-4,4,2,-4,4,1,3]))$ retourne $(2,[False,False,True,False,False,True,False,False])$

Cet algorithme trouve le premier indice du plus grand élément d'une liste d'entrée et il retourne cet indice, ainsi qu'une liste de booléens indiquant la présence ou non de ce plus grand élément aux indices correspondants dans la liste d'entrée.

Le principe est d'organiser un tournoi entre deux éléments consécutifs et d'éliminer les éléments qui perdent le tournoi en mettant leur valeur correspondante dans B à False. A la fin du tournoi, il existe au moins un indice k tel que $T[k]$ n'a jamais été battu. Il s'agit des indices correspondant au plus grand élément; on recherche ensuite simplement le premier d'entre eux.

3 Analyse d'algorithmes

Exercice 3 : Algorithme F1 😎

On considère l'algorithme ci-dessous.

1: **procedure** $F1(n \in \mathbb{N}^+)$
2: **for** $i \in [1,n]$ **by** 1 **do**
3: **for** $j \in [i,n]$ **by** 1 **do**
4: Afficher ("Salut")

Calculez (comme fonction de n) le nombre de fois que la chaîne de caractères "Salut" est affichée. Vous pouvez tester pour des petites valeurs de n. En déduire la complexité en notation asymptotique.

```python
def f1(n):
    for i in range(0,n+1):
        for j in range(i,n+1):
            print('salut')

f1(3)
```

La chaine de caractère "Salut" est affichée $\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ fois. La complexité est donc en $O(n^2)$: en clair l'ordre de grandeur du nombre d'impressions est en n^2.
Définition : $f(n) \in O(g(n))$ ssi

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}$$

$$\forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)$$

Posons $c = 1$ et on remarque que $\frac{n(n+1)}{2} \leq n^2,\ \forall n \geq 1$

On peut aussi définir $f(n) = \Theta(g(n))$ ssi $f(n) = O(g(n))$ **et** $g(n) = O(f(n))$

Exercice 4 : Algorithme F2

On considère:

1: **procedure** $F2(n \in \mathbb{N}^+)$
2: **while** $i \in [1, n]$ **do**
3: Afficher ("Salut") ;
4: $i \leftarrow 2i$

Calculez (comme fonction de n) le nombre de fois que la chaîne de caractères "Salut" est affichée. Vous pouvez tester pour des petites valeurs de n. En déduire la complexité en notation asymptotique.

```
1 def f2(n):
2     i = 1
3     while i < n+1:
4         print('salut')
5         i = 2*i
```

La première boucle se fait avec $i = 1$
La k-ième boucle se fait avec $i = 2^{k-1}$
On a donc k_{max} boucles de telle sorte que:

$$n \leq 2^{k_{max}-1} < n + 1$$

Et donc:

$$log_2(n) + 1 \leq k_{max} < log_2(n + 1) + 1$$

On a donc une complexité de l'ordre de $O(log_2(n))$

Exercice 5 : Algorithme F3

Considérons:

1: **procedure** $F3(n \in \mathbb{N}^+)$
2: $i \leftarrow 1$
3: **while** $i \in [1, n]$ **by** 1 **do**
4: **for** $j \in [1, i]$ **by** 1 **do**
5: Afficher ("Salut")
6: $i \leftarrow 2 * i$

Calculez (comme fonction de n) le nombre de fois que la chaîne de caractère "Salut" est affichée. Vous pouvez tester pour des petites valeurs de n. En déduire la complexité en notation asymptotique.

```
def f3(n):
    i = 1
    while i < n+1:
        for j in range(0,i+1):
            print('salut')
        i = 2*i
```

Cela est équivalent à avoir:

```
k=0
while k < ln(n+1)+1:
    for j in range(0,2^{k-1}):
        print('salut')
    k = k+1
```

Ainsi on a $\sum_{k=1}^{ln(n+1)+1} 2^{k-1} = \sum_{k=0}^{ln(n+1)} 2^k$
Ce qui par la formule des sommes de puissances vaut

$$2^{ln(n+1)+1} - 1 = 2 * (n+1) - 1 = 2n + 1$$

On a donc un complexité en $O(n)$.

Exercice 6 : Apprendre par l'exemple 😎

Il est relativement simple de contrôler le pas d'une boucle **while**. Ci-dessous est fourni le pseudo-code d'un algorithme permettant de calculer la puissance de 2 immédiatement supérieure à un nombre n donné.

1: **procedure** PROC($n \in \mathbb{N}$)
2: $i \leftarrow 1$
3: **while** $i < n$ **do**
4: $i \leftarrow 2 * i$
5: **return** i

On peut alors contrôler la complexité en temps par le nombre de pas nécessaires à la complétion de la boucle **while**. Ceci est fait ci-dessous par l'ajout de la variable c.

1: **procedure** PROC($n \in \mathbb{N}$)
2: $c \leftarrow 0$
3: $i \leftarrow 1$
4: **while** $i < n$ **do**
5: $c \leftarrow c + 1$
6: $i \leftarrow 2 * i$
7: **return** i, c

Le code Python correspondant à l'algorithme est le suivant :

```
def proc(n):
    c = 0
    i = 1
    while i < n:
            c = c + 1
            i = 2 * i
```

```
7       return(i,c)
```

On peut aussi contrôler le pas de la boucle de cette manière :

1: **procedure** PROC($n \in \mathbb{N}^+$)
2: $c \leftarrow 0$
3: **for** $i \in [1, n]$ **by** $i = 2i$ **do**
4: $c \leftarrow c + 1$
5: **return** $2 * i, c$

La construction Python usuelle pour le parcours entier $range(start, stop[, step])$ permet d'incrémenter les indices d'une boucle de la valeur prise par *step* (en l'absence de l'argument *step*, les indices sont incrémentés de 1 par défaut). Cependant *step* définit un pas fixe alors que notre algorithme nécessite un pas variable.

Ainsi, transcrire un tel algorithme en Python n'est pas direct car on parcourt une liste ($[1, n]$) pas à pas avec un pas variable. La fonction usuelle Python "$range(0, n, p)$" n'est donc pas adaptée ici.

Une façon de faire est de coder une fonction auxiliaire (*iter*) qui prend en entrée une fonction f, un entier n et une liste l et qui retourne la liste des images des éléments de l par f, strictement inférieures à n. Par exemple, si f est la fonction $x \mapsto x^2$, $n = 9$ et $l = [1, 2, 3]$, alors $iter(f, n, l)$ est $[1, 4]$.

Nous proposons l'implémentation récursive suivante:

```
1  import math
2
3  def iter (f, borne, l ):
4      if l == []:
5          return []
6      elif f(l[0]) > borne :
7          return []
8      else:
9          return [f(l[0])] + iter(f,borne,l[1:])
10 # >>> iter(deux_puis, 20, [1,2,3,4,5])
11 # [2.0, 4.0, 8.0, 16.0]
12 # iter applique f aux elements de l et tronque la liste avec la borne
```

Pour avoir un pas de $i = 2i$, c'est à dire une itération par multiplication par 2, il suffit d'utiliser la fonction auxiliaire *tousles2i* et la fonction Python *range* de la façon suivante :

```
1  def deux_puis(n):
2      return math.pow(2,n)
3
4  def from_to_by(i,n,f):
5      return iter(f, n ,range(i,n) )
6  #>>> from_to_by(0,12,deux_puis)
7  #[1.0, 2.0, 4.0, 8.0]
8  #appliquer f de i a (n -1),
9  # en s'arretant si f(x) est superieur a n
10
11 def proc(n):
12     c = 0
13     for i in from_to_by(0,n,deux_puis):
14         print(i,'',c)
15         c = c + i
16     return c
17 #>>> proc(12)
18 #1.0  0
19 #2.0  1.0
```

```
#4.0  3.0
#8.0  7.0
#15.0
# les indices de la boucle suivent la progression
# 1, 2, 4, 8
```

On voit ici qu'il y a parfois des contorsions pour passer de la version pseudo-code vers du code Python : la progression des indices d'une boucle en puissance de 2 ne se transcrit ainsi pas naturellement en Python.

Exercice 7 : Algorithme F4

Considérons:

procedure $F4(n \in \mathbb{N}^+)$
  for $j \in [1, n]$ **by** 1 **do**
    $i \leftarrow 1$
    while $i \in [1, j]$ **do**
      Afficher ("Salut")
      $i \leftarrow 2 * i$

Calculez (comme fonction de n) le nombre de fois que la chaîne de caractères "Salut" est affichée. Vous pouvez tester pour des petites valeurs de n. En déduire la complexité en notation asymptotique.

```
def f4(n):
    for j in range(0,n):
        i = 1
        while i <= j:
            print('salut')
            i = 2*i
```

C'est équivalent à:

```
for j in range(0,n):
    k=0
    while k <= ln(j):
        print('salut')
        k += 1
```

Et donc on a:

$$\sum_{j=0}^{n-1} \sum_{k=0}^{log_2(j)} 1 = \sum_{j=1}^{n} log_2(j) + 1 = log_2(\prod_{j=1}^{n} j) + n = log_2(n!) + n$$

Or selon la formule de Stirling:

$$n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$$

D'où une complexité

$$\sim log_2(\sqrt{2\pi n}) + n * log_2(n/e) + n \sim n * log_2(n)$$

On a donc une complexité en $O(n * log_2(n))$

4 Commencer à optimiser sa complexité

Exercice 8 : Tri par sélection

Le principe de tri par sélection est le suivant :

1. Rechercher l'élément le plus petit de la liste.
2. Le placer en début de la liste ;
3. Puis recommencer avec la liste privée du plus petit élément.

Question 1. Illustrer le principe du tri par sélection sur la liste [3,19,7,1,5]

Question 2. Écrire l'algorithme de tri par sélection en pseudo-code et l'implémenter en Python.

Pensez à prendre l'habitude dès maintenant de décomposer le problème en fonctions. Par exemple, votre fonction implémentant le tri par sélection pourrait faire appel à une fonction échange (l,i,j) permettant l'échange des deux éléments aux indices i et j dans une liste l.

Question 3. Calculer la complexité de l'algorithme dans le pire et dans le meilleure des cas (sans la prouver). Est ce que celle-ci dépend de l'ordre du tableau initial ?

Question 1. $[3,19,7,1,5] \to [1,3,19,7,5] \to [1]+[3,19,7,5] \to [1]+[3,19,7,5] \to [1]+[3]+[19,7,5] \to [1]+[3]+[5,19,7] \to [1]+[3]+[5]+[19,7] \to [1]+[3]+[5]+[7,19] \to [1]+[3]+[5]+[7]+[19]$

Question 2.

```
1: procedure Indice_min(l, i ∈ ℕ⁺)
2:     n = len(l)
3:     c = i
4:     for j ← i + 1 to n do
5:         if l[j] < l[c] then
6:             c ← j
7:     return c
8:
9: procedure Echange(l, i, p )
10:    aux ← l[i]
11:    l[i] ← l[p]
12:    l[p] ← aux
13:
14: procedure TriS(l)
15:    n = len(l)
16:    for i ← 1 to n − 1 do
17:        p ← Indice_min(l, i)
18:        if not p == i then
19:            Echange(l, i, p)
```

```python
def indice_min(l,i):
    n = len(l)
    c = i
    for j in range(i , n):
        if l[j] < l[c]:
            c = j
    return c
```

```
9
10 def Echange(l , i ,p):
11     aux = l[i]
12     l[i] = l[p]
13     l[p] = aux
14
15 def TriS(l):
16     n = len(l)
17     for i in range(0,n-1):
18         p = indice_min(l,i)
19         if not p == i:
20             Echange(l,i,p)
21     return l
22
23 print(TriS([8,3,2,3,5,4,6]))
```

Question 3. La fonction $Indice_{min}$ est appelée n fois. Et la complexité de $indice_{min}(l,i)$ est de complexité $n-i$ pour i allant de 1 à n. La complexité est donc en $O(\sum_{i=1}^{n} i) = O(\frac{n(n+1)}{2}) = O(n^2)$

Exercice 9 : Tri Fusion

À partir de deux listes triées, on peut facilement construire une liste triée comportant les éléments issus de ces deux listes (leur *fusion*). Le principe de l'algorithme de tri fusion repose sur cette observation : le plus petit élément de la liste à construire est soit le plus petit élément de la première liste, soit le plus petit élément de la deuxième liste. Ainsi, on peut construire la liste élément par élément en retirant tantôt le premier élément de la première liste, tantôt le premier élément de la deuxième liste (en fait, le plus petit des deux, à supposer qu'aucune des deux listes ne soit vide, sinon la réponse est immédiate).
L'algorithme du tri fusion est le suivant :

1. Si la liste n'a qu'un élément (ou aucun), elle est déjà triée.
2. Sinon, séparer la liste en deux parties à peu près égales.
3. Trier récursivement les deux parties avec l'algorithme de tri fusion.
4. Fusionner les deux listes triées en une seule liste triée.

Question 1. Illustrer le principe du tri fusion sur la liste [3,19,7,1]

Question 2. Écrire en pseudo-code puis en Python une fonction fusion qui prend en entrée deux listes l1 et l2 supposées triées et renvoie une liste l contenant les mêmes éléments que l1 et l2 mais rangés par ordre croissant.

Question 3. Écrire en pseudo-code puis en Python la fonction récursive de tri qui, si la liste contient 0 ou 1 élément, la renvoie telle quelle (puisqu'elle est déjà triée) et sinon renvoie le résultat de la fusion de sa partie gauche triée et de sa partie droite triée.

Question 4. En vous aidant de vos réponses aux deux questions précédentes, écrire en pseudo-code puis en Python l'algorithme de tri fusion.

Question 5. Quelle est la complexité dans le pire des cas de cette algorithme ?

Question 1. $[19,3,7,1] \to [19,3]+[7,1] \to [19]+[3]+[7]+[1] \to [3,19]+[1,7] \to [1,3,7,19]$

Question 2.

```python
def merge(l_1, l_2):
    if l_1[0] <= l_2[0]:
        return [l_1[0]] + merge(l_1[1:],l_2)
    else:
        return [l_2[0]] + merge(l_1, l_2[1:])


def triF(l):
    n = len(l)
    if len(l) == 0 or len(l) == 1:
        return l
    else:
        return merge(triF(l[0:(int(n/2))]), triF(l[(int(n / 2) ):n]))









print(triF([1, 2, 4, 3, 2, 8, 9, 3, 7 , 3 ]))
```

Question 3. La fonction *merge* a une complexité linéaire . Le nombre d'instructions élémentaires de la fonction triF est $\sum_{i=0}^{log_2 n} 2^i * |merge(l_{r_i}, l_{t_i}|$, où l_{r_i} et l_{r_j} sont des listes de taille $\frac{n}{2^i}$ donc la complexité est en $O(nlog(n))$.

Exercice 10 : Comparaison

Utiliser `import time` dans Python pour vous servir des méthodes sur le temps de calcul. La commande $t = time.process\_time()$ permet d'obtenir le temps à un moment donné du programme. Pour savoir le temps de calcul d'un algorithme, tapez $t_0 = time.process\_time()$ avant l'algo puis $t_1 = time.process\_time()$ à la fin, puis affichez la valeur de $t_1 - t_2$ à l'écran.

- La fonction $randint(0, x)$ vous permet de générer un nombre entier aléatoire dans l'intervalle $[0, x]$. Créez une liste aléatoire de 1000 entiers allant de 0 à 500.
- Comparez les temps de calcul de vos deux algorithmes de tri sur cette liste.

Question 1 :

```python

import random

def list(n):
    l=[]
    for i in range(0,n):
        l += [random.randint(0, 500)]
    return l

print(list(7))
```

Question 2 :

```
1
2  import time
3
4  d = list(1000)
5
6  t0 = time.process_time()
7  triS(d)
8  t1 = time.process_time()
9
10 print(t1 - t0)
11
12 t2 = time.process_time()
13 triF(d)
14 t3 = time.process_time()
15
16 print(t3 - t2)
```