Chapitre 1

Algorithmique, diviser pour régner

Sommaire

1.1 Principe général

Un algorithme de type *diviser pour régner* se décompose en trois parties :

1. **Diviser** : On divise le problème initial en plusieurs sous-problèmes plus petits que le problème initial.
2. **Résolution récursive des sous-problème** : On résoud chacun des sous-problèmes. La méthode est particulièrement efficace si on peut le faire de manière récursive. Il faut pour cela que les sous-problèmes soient des instances du problème initial avec de plus petits paramètres.
3. **Régner** : On peut reconstituer la solution du problème initial avec les solutions des différents sous-problèmes.

Condition sur l'appel récursif. L'étape crutiale de cette méthode de recherche est l'étape de *division*. Pour être efficace, cette stratégie doit *diviser* le problème. Pour être plus précis, si on applique un algorithme de type diviser pour régner à une donnée de taille N, alors les appels récursifs doivent se faire sur des données de tailles $\frac{N}{k}$, avec k un entier trictement plus grand que 1.

Précisions sur les algorithmes. Les algorithmes de type diviser pour régner sont basés sur basés sur la récursivité. En pratique, ils seront implémentés en Python à l'aide de fonctions *récursives*. Pour chacun des exemples qui vont suivre, les algorithmes seront écrits en pseudo-code, sous la forme de fonctions récursives. L'emploi des fonctions récursives étant peu efficace en général (comparé à des fonctions itératives), il existe souvent des versions itératives des exemples que nous allons présenter, mais qui ne seront pas forcément présentées ici.

1.2 Exemples d'algorithmes

1.2.1 Recherche dichotomique (révision de première)

Rappel du problème. On se donne une liste triée, et on veut pouvoir déterminer si un élément se trouve ou non dans la liste. On peut spécifier le problème de la manière suivante :

- **Données :** Une liste `lst` d'éléments, et un élément `elt`.
- **Précondition :** La liste `lst` doit être triée selon l'ordre croissant. Autrement dit, on doit avoir `lst[i] <= lst[j]` pour tout couple valide d'indices `i` et `j`, tel que `i < j`. L'élément `elt` doit être comparable avec tous les éléments de `lst`.
- **Résultat :** `True` si `elt` est un élément de `lst`, et `False` sinon.
- **Postcondition :** Aucunes.

Diviser pour régner. On commence notre recherche par le *milieu de la liste*. Si l'élément central de `lst` est `elt`, alors on renvoie `True`. Sinon, on peut utiliser le fait que la liste est ordonnée pour éliminer une moitié des éléments de `lst`. On applique ensuite récursivement la recherche dichotomique sur la moitié restante des éléments.

Le cas minimum assurant l'arrêt des appels récursif est le cas de la liste vide : si la liste `lst` est vide, elle ne contient pas l'élément `elt`, et on renvoie `False`. Le pseudo-code de cet algorithme est donné par la fonction rechercheDichoRec (page 3).

Fonction rechercheDichoRec(`elt`, `lst`)

Entrées :
- `elt`, un élément comparable avec tous les éléments de `lst`
- `lst`, une liste d'éléments triés par ordre croissants

Sorties :
- `True` si `elt` est un élément de `lst`
- `False` dans le cas contraire

```
1  début
2      si lst est vide alors
3          retourner False
4      sinon
5          k = len(lst) // 2 (On cherche l'indice de l'élément central)
6          si elt = lst[k] alors
7              retourner True
8          sinon si elt < lst[k] alors
9              retourner rechercheDichoRec(elt, lst[:k])
10         sinon
11             retourner rechercheDichoRec(elt, lst[k+1:])
```

Preuve de terminaison. L'appel récursif se fait sur une liste de taille strictement plus petite que la taille de la liste passée en entrée. La condition d'arrêt portant sur la taille de la liste (on vérifie si la liste est vide), les appels récursifs vont se terminer.

Preuve de correction. Si la liste est vide, notre algorithme renvoie clairement `False`. De même, lors du premier test, notre algorithme vérifie si l'élément d'indice `k` est égale à `elt`. Si c'est le cas, on renvoie `True`.

Si `lst[k]` n'est pas égal à `lst`, on teste si `elt < lst[k]`. Si c'est le cas, alors il est inutile de tester les éléments d'indices plus grands que `k`. En effet, si `i` est un indice valide de `lst`, et que `k < i`, alors on a `elt < lst[k] <= lst[i]`, puisque les éléments de la liste sont rangés en ordre croissant.

Nous ne devons alors que tester les éléments d'indice `i < k`. C'est ce que fait l'appel récursif.

Un raisonnement tout à fait symétrique s'applique si `elt > lst[k]`.

Complexité. Nous allons évaluer la complexité de cette fonction par le *nombre de comparaisons* qu'elle devra faire entre `elt` est les éléments de la liste `lst` pour se terminer. On notera C la fonction de complexité, et $C(n)$ le nombre de comparaisons nécessaires à la fonction rechercheDichoRec pour terminer sur une liste de n éléments. peut déjà noter que :

- Si la liste `lst` est vide, la fonction rechercheDichoRec n'exécute aucune comparaison. On a donc $C(0) = 0$.
- Si la liste `lst` n'est pas vide, le fonction rechercheDichoRec fait une seule comparaison, puis un appel récursif à la fonction sur une liste de taille $\frac{n}{2}$ dans le pire des cas. On a donc la relation de récurrence suivante : $C(n) = C\left(\frac{n}{2}\right) + 1$, dans le pire des cas.

Il nous faut donc résoudre le système suivant :

$$C(n) = \begin{cases} 0 & \text{si } n = 0 \\ C\left(\frac{n}{2}\right) + 1 & \text{si } n > 0 \end{cases}$$

On commence par un **cas particulier** : si n est une **puissance de 2**, c'est à dire si n est de la forme $n = 2^k$, avec k un entier naturel. Dans ce cas, on peut résoudre le système par "extension" de la formule :

$$C(n) = C(2^k) = C(2^{k-1}) + 1 = C(2^{k-2}) + 2 = \cdots = C(0) + k = k$$

Dans le cas de $n = 2^k$, on a donc $C(n) = k = \log_2(n)$.

On peut maintenant généraliser pour d'autres valeurs de n. Si on a une liste de n éléments, on peut toujours encadrer n par deux puissances de 2 : on pourra toujours trouver un nombre entier naturel k tel que $2^k < n < 2^{k+1}$.

Par croissance de la fonction de complexité, on en déduit :

$$C(2^k) \leq C(n) \leq C(2^{k+1}) \quad \Longleftrightarrow \quad k \leq C(n) \leq k+1$$

Or on a $k = \lfloor \log_2(n) \rfloor$. On en déduit l'inégalité :

$$C(n) \leq \lfloor \log_2(n) \rfloor + 1 \leq \log_2(n) + 1$$

La fonction de complexité est donc de classe $O(\log_2(n))$.

Une implémentation. L'algorithme utilise des tranches pour gérer les sous-listes de `lst`. Le code est disponible dans le listing 1

```
1  def rechercheDichoRec(elt, lst):
2      if len(lst) == 0:
3          return False
4      else:
5          k = len(lst) // 2
6          if lst[k] == elt:
7              return True
8          elif elt < lst[k]:
9              return rechercheDichoRec(elt, lst[:k])
10         else:
11             return rechercheDichoRec(elt, lst[k+1:])
```

Listing 1: Recherche dichotomique

1.2.2 Exponentiation rapide

Présentation du problème. Soit x un nombre (entier ou décimal), et n un entier positif ou nul. Nous voulons implémenter une fonction qui calcule x^n, c'est à dire qui respecte les spécifications suivantes :

- **Entrées :** Un nombre x (entier ou flottant), et un entier positif n.
- **Préconditions :** Aucune sur x. n doit être positif ou nul.
- **Sortie :** x^n.
- **Postcondition :** Aucune.

Un algorithme naïf. La méthode naïve de résolution de ce problème repose sur la définition de la puissance, que nous rappelons ci-dessous :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \underbrace{x \times x \times \ldots \times x}_{n-1 \text{ multiplications}} & \text{sinon} \end{cases}$$

La méthode naïve consiste donc à multiplier le nombre x avec lui-même $n-1$ fois lorsque $n > 0$, et de renvoyer 1 lorsque $n = 0$. Le pseudo-code de cet algorithme est donné dans le listing 1 (page 5).

Algorithme 1 : Exponentiation naïve 1

Data :
- Un nombre x (entier ou flottant).
- Un entier naturel n.

Result : x^n

```
1 début
2     si n = 0 alors
3         retourner 1
4     sinon
5         res = 1
6         pour k variant de 0 à n − 1 faire
7             res = res ×x
8         retourner res
```

Si l'on se penche sur cette méthode naïve, on peut se rendre compte que l'on peut se passer du test *si ... alors*. En effet, si on exécute le bloc d'instruction correspondant au **sinon** avec la valeur $n = 0$, on écrit la valeur 1 dans la variable res, puis on *n'exécute aucun tour de la boucle pour* (k ne peut pas varier entre 0 et -1). On peut donc réécrire une version légèrement améliorée de cet algorithme, présente dans le listing 2 (page 5).

Algorithme 2 : Exponentiation naïve 2

Data :
- Un nombre x (entier ou flottant).
- Un entier naturel n.

Result : x^n

```
1 début
2     res = 1
3     pour k variant de 0 à n − 1 faire
4         res = res ×x
5     retourner res
```

Complexité en nombre de multiplications de l'algorithme naïf. Pour calculer x^n, cet algorithme exécute exactement n multiplications. La fonction de complexité de l'algorithme naïf (en nombre de multiplications) est donc $C(n) = n = O(n)$. La complexité de l'algorithme naïf est *linéaire*.

Diviser pour régner. Il est possible d'adapter cet algorithme à l'aide d'une approche de type diviser pour régner. Elle se base sur l'observation suivante :

- Si $n = 0$, il suffit de renvoyer 1.

- Si n est un entier positif *pair*, on peut diviser en deux le calcul de x^n :

$$x^n = \underbrace{x \times x \times \ldots \times x}_{a} \times \underbrace{x \times x \times \ldots \times x}_{a} = a \times a$$

- Même chose si n est un entier *impair*, il suffit de rajouter une multiplication par x en plus.

$$x^n = \underbrace{x \times x \times \ldots \times x}_{a} \times \underbrace{x \times x \times \ldots \times x}_{a} \times x = a \times a \times x$$

Avec cette décomposition, il ne faut calculer le facteur a qu'une seule fois. On peut ensuite élever a au carré pour calculer x^n. Ce faisant, on économise la moitié des multiplications. Les différentes décompositions sont résumée ci-dessous :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^{\frac{n}{2}})^2 & \text{si } n > 0 \text{ est pair} \\ (x^{\frac{n-1}{2}})^2 \times x & \text{si } n > 0 \text{ est impair} \end{cases}$$

Cette formule nécessite de calculer $x^{\frac{n}{2}}$ ou $x^{\frac{n-1}{2}}$, ce qui peut se faire récursivement (on pourra vérifier que les exposants sont entiers, et donc que la précondition sur n est vérifiée). Il s'agit bien d'une stratégie de type diviser pour régner, car la taille des appels récursifs se font sur un exposant n deux fois plus petit que l'exposant en entrée. L'algorithme basé sur cette observation est l'algorithme d'*exponentiation rapide*. Il est donné dans la figure 3

Algorithme 3 : Exponentiation rapide 1

Data :
- Un nombre x (entier ou flottant).
- Un entier naturel n.

Result : x^n

```
1 début
2     si n = 0 alors
3         retourner 1
4     sinon si n est pair alors
5         a = Exp_rapide(x, n/2)
6         retourner a × a
7     sinon
8         a = Exp_rapide(x, (n-1)/2)
9         retourner a × a × x
```

Complexité de la méthode diviser pour régner. On la compte encore en nombre de multiplications. Nous allons différencier le pire du meilleur des cas.

Dans tous les cas, si $n = 0$, il n'y a aucune multiplications. On peut donc écrire $C(0) = 0$, avec C la fonction de complexité de l'algorithme.

Le pire des cas se présente lorsque n est impair. On peut constater que l'on doit faire *deux* multiplications, en plus de l'appel récursif. La fonction de complexité dans le pire des cas est donc :

$$C(n) = \begin{cases} 0 & \text{si } n = 0 \\ C\left(\frac{n}{2}\right) + 2 & \text{sinon} \end{cases}$$

La résolution est identique à celle de la recherche dichotomique. On montrer d'abord que $C(2^k) = k$. Puis on pourra majorer la fonction de complexité dans tous les cas pour montrer que la complexité de l'algorithme d'exponentiation rapide est en $O(\log_2(n))$.

Implémentation. Le code python, reprenant l'algorithme 3 (page 6) est donné ci-dessous :

```
def expRapideRec(x, n):
    if n == 0:
        return 1
    else:
        if n%2 == 0:
            a = expRapideRec(x, n // 2)
            return a * a
        else:
            a = expRapideRec(x, (n-1) // 2)
            return a * a * x
```

Listing 2: Exponentiation rapide

1.2.3 Le tri fusion (Merge sort)

L'algorithme

Description de l'algorithme. Le tri fusion est une méthode de tri de type diviser pour régner. Les étapes de l'algorithme sont les suivantes. La figure 1.1 résume ces différentes étapes. Les listes représentées en *rouge* ne sont *pas triée*. Les listes représentées en *vert* sont *triées*.

1. **Diviser** : On coupe la liste initiale en deux parties égales.
2. **Résolution récursive** : on utilise le tri fusion récursivement pour trier les deux sous-listes.
3. **Régner** : On **fusionne** les deux listes triées pour fabriquer une grande liste. Cette étape de **fusion** est la plus importante.

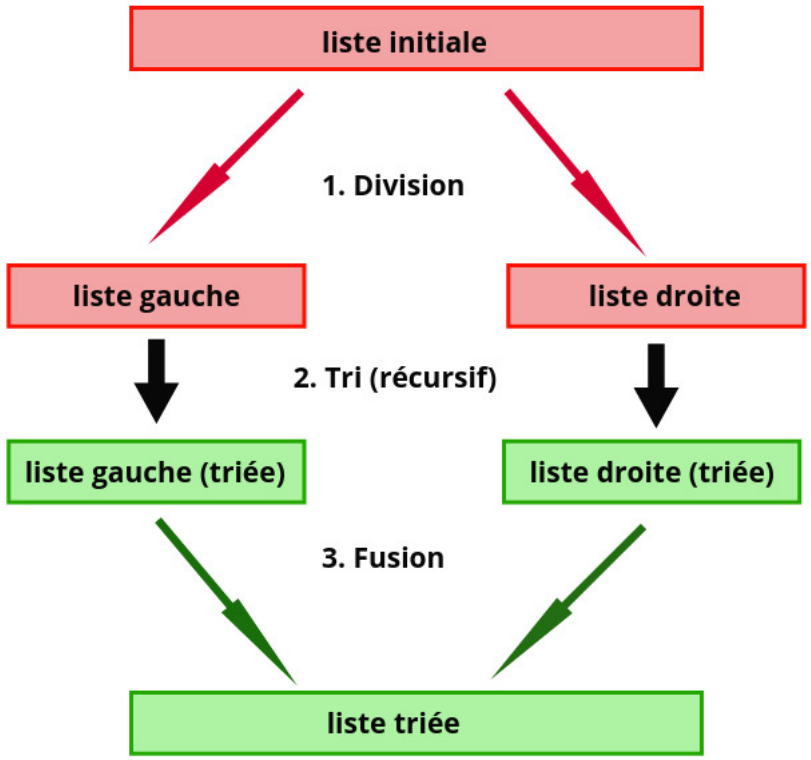

FIGURE 1.1 – Principe de fonctionnement récursif du tri-fusion

Cette description sommaire s'accompagne des spécifications de l'algorithme, qui sont les mêmes que pour n'importe quel algorithme de tri. On les rappelle ci-dessous :

- **Données :** Une liste `lst`.
- **Préconditions :** Les éléments de `lst` sont comparables deux à deux.
- **Résultat :** Une liste `liste_triee`.
- **Postcondition :** Les éléments de `liste_triee` sont exactement ceux de `lst`, rangés par ordre croissant.

Exemple d'exécution de l'algorithme. La figure 1.2 (page 8) nous montre le fonctionnement de l'algorithme sur la liste `[38, 27, 43, 3, 9, 82, 10]`. Les transitions en *rouge* correspondent aux étapes de *division*. Les transitions en *vert* correspondent aux étapes de *fusion*.

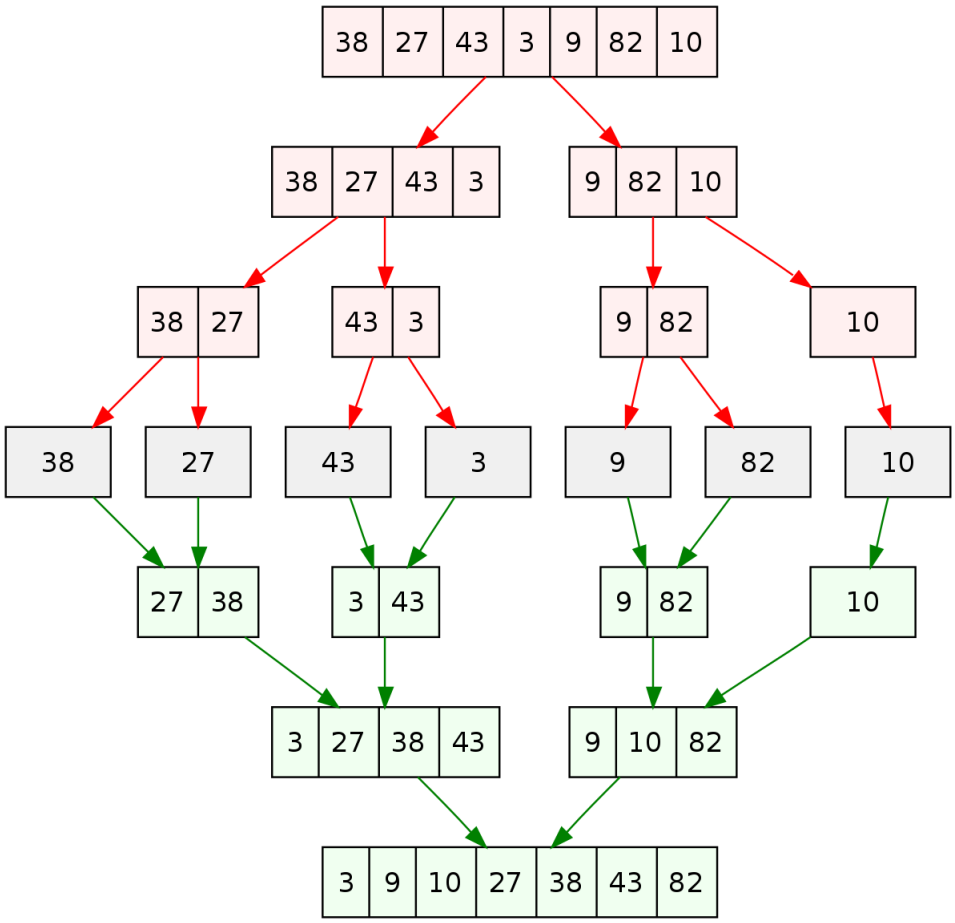

FIGURE 1.2 – Exemple d'exécution du tri-fusion

Pseudo-code du tri fusion. Nous allons commencer à définir les objets sur lesquels nous allons appliquer l'algorithme. Comme nous travaillons en Python, nous utiliserons des listes. Pour le moment, nous les utilisons en toute généralité. Nous détaillerons dans la partie sur l'implémentation la manière dont nous nous servirons de ces listes. Ceci dépendra de comment nous comptons réaliser le tri, si on le réalise *en place*, ou non.

Il faut également remarquer que le tri fusion est un algorithme de tri **récursif**. Nous devons donc définir une condition d'arrêt pour l'algorithme. Cette condition est assez simple :

- Si la liste à trier est *vide*, ou si elle ne contient qu'*un seul élément*, alors elle est déjà triée.
- Si ce n'est pas le cas, on applique la procédure de division/fusion vue plus haut.

Algorithme 4 : Tri fusion

Data : Une liste `lst` de n éléments
Result : Une liste `liste_triee`

```
1  début
2      si lst est vide, ou contient un élément alors
3          retourner lst
4      sinon
5          Créer une liste_gauche, qui contient la moitié des éléments de lst
6          Créer une liste_droite, qui contient l'autre moitié des éléments de lst
7          liste_gauche = tri_fusion(liste_gauche)
8          liste_droite = tri_fusion(liste_droite)
9          liste_triee = fusion(liste_gauche, liste_droite)
10         retourner liste_triee
```

Preuve de terminaison du tri fusion. Si la liste `lst` à trier est vide, ou si elle est de taille 1, l'algorithme termine (il se contente de renvoyer la liste originale). Si la liste `lst` contient n éléments, avec $n > 1$, alors il y a deux appels récursifs dans l'algorithme. Comme on sépare la liste de n éléments en deux listes *non vides*, ces appels se font sur des listes *strictement plus petites* que la liste `lst` de départ. Nous sommes donc assurés que ces appels récursifs terminent.

Preuve de correction du tri fusion. D'après le pseudo-code de l'algorithme, les listes `liste_gauche` et `liste_droite` contiennent à elles-deux tous les éléments de `lst`.

D'après les spécifications du tri fusion, nous sommes assurés que ces listes ont été ordonnées, sans perte ni gain d'élément par les deux appels récursifs du tri fusion.

Enfin, d'après les spécifications de la fonction de fusion, comme les deux listes `liste_gauche` et `liste_droite` sont ordonnées par ordre croissant (on respecte la précondition de la fonction de fusion), on peut être assuré que la `liste_triee` est la réunion des deux listes droites et gauches, avec ses éléments ordonnées en ordre croissant.

Complexité du tri fusion. Nous allons calculer la complexité en terme de nombre de tests nécessaires pour accomplir le tri. On note N la taille de la liste donnée en entrée et $C(N)$ la fonction de complexité du tri fusion.

Nous allons commencer par nous interresser aux listes dont la taille est une puissance de 2 (*i.e* $N = 2^p$, avec p un entier naturel quelconque). Si la liste en entrée ne contient qu'un seul élément ($N = 1$), alors on ne fait qu'un seul test (Le premier *si*, qui teste la taille de la liste en entrée). On a donc clairement $C(1) = 1$. Même chose pour $C(0) = 1$, dans le cas d'une liste vide.

Maintenant, on verra à la partie suivante que l'étape de fusion nécessite $N-1$ tests pour une entrée de taille N. On a donc clairement la relation de récurrence ci-dessous :

$$C(N) = \begin{cases} 1 & \text{si } N = 1 \\ 2C(\frac{N}{2}) + N & \text{sinon} \end{cases}$$

L'étape de fusion

L'étape de fusion. Il s'agit de la partie centrale de l'algorithme de tri-fusion. On peut en donner les spécifications suivantes :

- **Données :** Deux listes `lst1` et `lst2`.
- **Préconditions :** Les listes `lst1` et `lst2` sont rangées par ordre croissant.
- **Résultat :** Une liste `liste_triee`.
- **Postcondition :** La liste `liste_triee` est la réunion des éléments de `lst1` et `lst2`, rangés par ordre croissant.

La réalisation de la fusion se fait par un parcours simultané des listes `lst1` et `lst2`. On compare les plus petits éléments de ces listes entre eux. On récupère le plus petit des deux, pour l'écrire à la suite de la liste triée. On recommence l'opération jusqu'à remplir complètement la liste triée.

Fonction fusion(lst_gauche, lst_droite)

Entrées :
`lst_gauche`, une liste de p éléments rangés par ordre croissant
`lst_droite`, une liste de q éléments rangés par ordre croissant

Sorties :
`liste_triee`, fusion des listes `lst_gauche` et `lst_droite`

```
1  début
2      Créer une liste liste_triee, de taille p + q
3      Créer une variable i, égale à 0
4      Créer une variable j, égale à 0
5      pour k variant de 0 à p + q − 1 faire
6          si i = p alors la liste_gauche est épuisée
7              liste_triee[k] = liste_droite[j]
8              j += 1
9          sinon si j = q alors la liste_droite est épuisée
10             liste_triee[k] = liste_gauche[i]
11             i += 1
12         sinon si liste_gauche[i] < liste_droite[j] alors
13             liste_triee[k] = liste_gauche[i]
14             i += 1
15         sinon
16             liste_triee[k] = liste_droite[j]
17             j += 1
18     retourner liste_triee
```

Preuve de terminaison. C'est un algorithme qui utilise une boucle bornée, qui se traduira par un `for`.

Preuve de correction. Work in progress...

Une implémentation

Choix d'implémentation. Le tri fusion sera implémenté comme une fonction, prenant en entrée une liste `lst`, et renvoyant en sortie une liste `liste_triee`. Le tri ne modifie pas la liste `lst`. Le code de la fonction `tri_fusion` est présenté dans le listing 3.

```
1  def tri_fusion(lst):
2      # Sortie de récursion
3      if len(lst) <= 1:
4          return lst
5
6      # 1. création des listes gauches et droites
7      n = len(lst) // 2
8      lst_gauche = lst[:n]
9      lst_droite = lst[n:]
10
11     # 2. Résolution récursive du problème
12     lst_gauche = tri_fusion(lst_gauche)
13     lst_droite = tri_fusion(lst_droite)
14
15     # 3. Fusion des listes gauches et droites triées
16     return fusion(lst_gauche, lst_droite)
```

Listing 3: Tri fusion

La fonction de fusion. L'étape de fusion de l'algorithme sera elle aussi codée comme une fonction, prenant en entrée deux listes triées `lst1` et `lst2`, et renvoyant la fusion des deux listes en sortie. Les listes passées en argument ne sont pas modifiées par l'algorithme. Le code est présenté dans le listing 4.

1.2.4 Tri rapide (Quick sort)

L'algorithme

Spécifications. Ce sont les mêmes que pour les tris.

- **Entrée :** Une liste `lst` d'éléments (la liste peut être vide).
- **Précondition :** Les éléments de la liste sont comparables deux à deux.
- **Sortie :** Une liste contenant exactement les mêmes éléments que `lst`.
- **Postcondition :** La liste en sortie contient exactement les mêmes éléments que `lst`, rangés par ordre croissant.

Description de l'algorithme. Il repose sur la notion de pivot. On choisit un élément de la liste à trier comme *pivot*, puis on sépare la liste en deux groupes

1. **Diviser** : Il y a deux étapes.
   a. On choisit un pivot dans la liste à trier (n'importe quel élément fonctionne, en pratique ce choix peut influer sur les performances de l'algorithme).
   b. On constitue deux nouvelles listes, à partir de la liste à trier :

```python
def fusion(lst1, lst2):
    # 1. On crée la liste triée en avance
    liste_triee = [0] * (len(lst1) + len(lst2))
    i = 0 # indice de lst1
    j = 0 # indice de lst2

    # 2. On remplit la liste au fur et à mesure.
    for k in range(len(liste_triee)):
        if i == len(lst1): # lst1 est épuisée
            liste_triee[k] = lst2[j]
            j += 1
        elif j == len(lst2): #lst2 est épuisée
            liste_triee[k] = lst1[i]
            i += 1
        elif lst1[i] < lst2[j]:
            liste_triee[k] = lst1[i]
            i += 1
        else:
            liste_triee[k] = lst2[j]
            j += 1

    return liste_triee
```

Listing 4: La fonction de fusion

- une *liste gauche*, qui contient tous les éléments *inférieurs ou égaux* au pivot.
- Une *liste droite*, qui contient tous les éléments *strictement supérieurs* au pivot.

2. **Résolution récursive** : On trie les listes gauches et droites de manière récursive.
3. **Régner** : La liste finale est la concaténation de la liste gauche (triée), du pivot et de la liste droite (triée).

Ces éléments sont repris dans l'algorithme 5. Cet algorithme est volontairement peu détaillé, car la manière dont nous allons l'implémenter va dépendre de certains choix (fait-on un tri en place ou non ? Comment choisit-on le pivot ? Quelle structure de données utilise-t-on pour représenter les listes *etc.*).

Preuve de terminaison. Dans le pire des cas, les appels récursifs se font avec une liste de taille $n-1$. C'est une liste strictement plus petites que la liste en entrée. La condition d'arrêt étant la taille de la liste en entrée, cet algorithme termine.

Preuve de correction. WORK IN PROGRESS

Implémentation avec des compréhensions de listes

Choix d'implémentation. Dans cette version, on utilise les listes de Python, pour bénéficier du mécanisme de compréhension de liste. Le pivot sera la premier élément de la liste. L'algorithme sera implémenté sous forme d'une fonction `tri_rapide`, dont les spécifications sont les suivantes :

- **Paramètres :** Une liste Python `lst`, respectant les préconditions de l'algorithme.
- **Sortie :** Une liste triée reprenant tous les éléments

Algorithme 5 : Tri rapide

Données : Une liste `lst` de n éléments
Résultat : Une liste `liste_triee`

```
1  début
2      si lst est vide, ou contient un seul élément alors
3          retourner lst
4      sinon
5          Choisir un pivot dans lst (n'importe quel élément peut faire l'affaire)
6          Créer une liste_gauche, contenant tous les éléments de lst inférieurs ou égaux au pivot
             (pivot exclu)
7          Créer une liste_droite, contenant tous les éléments de lst strictement supérieurs au
             pivot
8          Trier récursivement liste_gauche
9          Trier récursivement liste_droite
10         liste_triee est la concaténation de liste_gauche, du pivot et de liste_droite
11         retourner liste_triee
```

Le code python est montré dans le listing 5. L'utilisation de listes en compréhension rend la traduction de l'algorithme très facile.

```python
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst

    else:
        # Choix du pivot et construction des listes droites et gauches
        pivot = lst[0]   # On choisit le premier élément
        lgauche = [lst[k] for k in range(1, len(lst)) if lst[k] <= pivot]
        ldroite = [lst[k] for k in range(1, len(lst)) if lst[k] > pivot]

        # Résolution récursive
        lgauche = tri_rapide(lgauche)
        ldroite = tri_rapide(ldroite)

        # Renvoi des morceaux
        return lgauche + [pivot] + ldroite
```

Listing 5: Tri rapide

Implémentation d'une version en place

Choix d'implémentation. Faire un tri *en place* signifie que l'on modifie la liste passée en paramètre. La liste initiale `lst` est donc perdue, remplacée par sa version triée. Nous allons implémenter cette version avec une *procédure* Python. C'est à dire que nous allons écrire une fonction sans valeur de retour, mais qui va modifier l'état de la liste `lst`, passée en paramètre.

Comme nous modifierons à chaque fois la liste `lst`, cette procédure aura `lst` comme paramètre pour chacun des appels récursifs. Comme les appels se feront seulement sur des portions de la liste `lst`, on

rajoute deux paramètres en plus pour indiquer exactement la portion de `lst` que l'on souhaite trier. Pour résumer, les paramètres de notre procédure `tri_rapide_enPlace` sont :

- `lst` : la liste sur laquelle on fait un tri.
- `debut` : Un entier (`int`).
- `fin` : Un entier (`int`).

L'appel `tri_rapide_enPlace(lst, debut, fin)` trie les éléments de `lst` entre les indices `debut` et `fin`. Si les indices ne sont pas valables (s'ils provoquent un dépassement, ou si `fin < debut`), la procédure laissera la liste `lst` inchangée.

L'implémentation en Python demande l'utilisation d'une fonction auxiliaire, que l'on va appeler `partition`.

La fonction de partition. Le rôle de cette fonction est de réaliser les instructions des lignes 5, 6 et 7 de l'algorithme 5 (page 13). La mission de cette fonction est de préparer la liste aux appels récursif du tri rapide, c'est à dire qu'elle devra :

- Choisir un pivot
- Déplacer les éléments de la liste pour placer tous ceux qui sont inférieurs au pivot à sa gauche, et ceux qui sont supérieurs à sa droite.
- renvoyer la position du pivot, pour pouvoir exécuter les appels récursifs.

Comme il s'agit d'un pré-traitement, cette fonction prendra en entrée les mêmes paramètres que la fonction `tri_rapide_enPlace` :

- **Entrée :**
  - `lst`, la liste de départ
  - `debut`, un entier (`int`)
  - `fin`, un entier (`int`)
- **Précondition :** Les indices `debut` et `fin` doivent être valides pour la liste `str`. Autrement dit :
  - `0 <= debut < fin < len(lst)`
  - `fin - debut > 1`

  Il est particulièrement important que ces conditions soient vérifiées. Le pré-traitement avant les appel récursifs
- **Sortie :** La position (l'indice) `p` du pivot dans la partition réalisée.
- **Postcondition :** La fonction de partition a placé tous les éléments plus petits à gauche du pivot, et plus grands à droite (entre les indices `debut` et `fin`). Autrement dit, on a :
  - `lst[debut] <= lst[k] <= lst[p]` si `debut <= k < p`.
  - `lst[p] < lst[k] <= lst[fin]` si `p < k <= fin`.

  Il est possible que `p` soit égal à `debut` ou à `fin`.

L'algorithme de la fonction de partition. WORK IN PROGRESS

Preuve de correction et de terminaison. La fonction de partition termine de manière évidente (c'est une boucle `for`).

WORK IN PROGRESS pour la preuve de correction.

```python
def partition(lst, debut, fin):

    pivot = lst[fin]
    i = debut     # indice de gauche
    j = fin - 1   # indice de droite

    for k in range(fin - debut):
        if lst[i] <= pivot:
            i += 1
        elif lst[j] > pivot:
            j -= 1
        else:
            lst[i], lst[j] = lst[j], lst[i]
            i += 1

    # Mise en place du pivot et renvoi de sa position
    lst[fin], lst[i] = lst[i], lst[fin]
    return i
```

Listing 6: Fonction de partition

Implémentation du tri rapide en place. Avec la fonction de partition, le tri rapide est relativement facile à mettre en place. Les lignes 5, 6 et 7 de l'algorithme 5 (page 13) sont prises en charge par la fonction de partition. Cette fonction nous informe également de la position du pivot, ce qui nous permet de réaliser les appels successifs en connaissant exactement les bornes des sous-listes à trier.

```python
def tri_rapide_enPlace(lst, debut, fin):
    if fin - debut <= 0: # lst de taille 1 ou moins, ou indices non valables
        pass
    else:
        indice_pivot = partition(lst, debut, fin)

        # Résolution récursive
        tri_rapide_enPlace(lst, debut, indice_pivot - 1)
        tri_rapide_enPlace(lst, indice_pivot + 1, fin)
```

Listing 7: Tri rapide en place

1.3 Ressources

- Le site de Wikipedia, parceque... pourquoi pas ? C'est une bonne *première* source...
- Une intro de cours sur le site de developpement-informatique.com.
- Un cours de supInfo international university sur la récursivité et diviser pour régner.
- Des slides de cours de l'INRIA
- Un cours/exos de classe prépa MPSI (attention, ils utilisent le langage OCaml ! C'est des oufs !)
- Et un autre TP de classe prépa... Toujours du Ocaml...
- Et voilà le site dédié au cours de l'exemple plus haut. Il y a plein de trucs, c'est cool.