Méthode : « Diviser pour régner »

Objectifs :

- Décomposition d'un problème en sous problèmes ;
- Prolongement du travail sur la récursivité ;
- Tri fusion d'une liste chaînée

Durée estimée : 4 h avec les exercices

I - Introduction et premiers exemples

La méthode dite « **diviser pour régner** » se décompose ainsi en trois phases :

- **Diviser :** on divise les données initiales en plusieurs **sous-parties**.
- **Régner :** on résout **récursivement** chacun des sous-problèmes associés (ou on les résout directement si leur taille est assez petite).
- **Combiner :** on **combine** les différents résultats obtenus pour obtenir une solution au **problème initial**.

Cela va prendre tout son sens sur les exemples à venir, qui vont nous permettre de nous familiariser avec ce principe général.

Pour chacun d'eux on présentera en détail les trois phases : diviser, régner et combiner.

I - 1. Calcul du maximum d'un tableau de nombres

En adoptant le paradigme « diviser pour régner », l'idée pour résoudre cette question est de calculer récursivement le maximum de la première moitié du tableau et celui de la seconde, puis de les comparer. Le plus grand des deux sera le maximum de tout le tableau. La condition d'arrêt à la récursivité sera l'obtention d'un tableau à un seul élément, son maximum étant bien sûr la valeur de cet élément.

Voici donc les trois étapes de la résolution de ce problème via cette méthode :

- **Diviser** le tableau en deux sous-tableaux en le « coupant » par la moitié.
- **Calculer récursivement** le maximum de chacun des sous-tableaux. Arrêter la récursion lorsque les tableaux n'ont plus qu'un seul élément.
- **Retourner** le plus grand des deux maximums précédents.

Voici donc l'algorithme :

```
Définition maximum_rec(tab, debut, fin):
    Si debut = fin alors
        Renvoyer tab[debut]
    FinSi
    milieu ← (debut + fin) // 2
    x ← maximum_rec(tab, debut, milieu)
    y ← maximum_rec(tab, milieu + 1, fin)
    Si x > y alors
        Renvoyer x
    Sinon
        Renvoyer y
    FinSi
FinDéfinition
```

```
Définition maximum(tab):
    maximum_rec(tab, 0, longueur(tab) −1)
FinDéfinition
```

On peut montrer de manière générale, que pour un tableau de taille 2^n, il y a $2^{n+1}-2$ appels récursifs, ou encore que pour un tableau de taille n il y en aura au plus $2n-2$.

Il est facile de se persuader que ce programme termine toujours :

- Dans le premier appel récursif, la valeur de la variable *debut* ne change pas alors que celle de la valeur *fin* diminue strictement en étant toujours supérieur ou égale à *debut*, ainsi la condition *debut* = *fin* sera atteinte ;
- Dans le second appel récursif, c'est la valeur de la variable *fin* qui ne change pas alors que celle de la variable *debut* augmente strictement en étant toujours inférieure ou égale à *fin*, la condition sera encore atteinte.

I - 2. Recherche dichotomique d'un élément dans un tableau trié

Rappelons que l'idée principale de la recherche dichotomique consiste à délimiter une portion du tableau `tab` dans laquelle la valeur `v` peut encore se trouver, avec deux indices `g` et `d`.

Voici une illustration de la situation à chaque étape :

	0	g	d	
tab	éléments < v	...	éléments > v	

On compare la valeur au centre de cet intervalle avec la valeur `v` et, selon le cas, on signale qu'on a trouvé la valeur `v` ou on se déplace vers l moitié gauche ou la moitié droite. Il s'agit bien là d'une instance de l'idée « diviser pour régner », car on réduit le problème de la recherche dans l'intervalle `[g;d]` à celui de la recherche dans un intervalle plus petit.

Là encore la condition d'arrêt à la récursivité sera l'obtention d'une liste à un seul élément. Voici donc les trois étapes de la résolution de ce problème via la méthode « diviser pour régner » :

- Diviser la liste en deux sous-listes de même taille (à un élément près) en la « coupant » par la moitié.
- Rechercher récursivement la présence de l'élément recherché dans la » bonne« des deux sous-listes après l'avoir comparé à l'élément situé au milieu de la liste.
- Pas de résultats à combiner puisque l'on ne « travaille » que sur l'une des deux sous-listes.

```
Définition dichotomie_rec(tab, v, g, d):
    Si g > d alors
        Renvoyer Rien
    FinSi
    milieu ← (g + d) // 2
    Si tab[milieu] < v alors
        Renvoyer dichotomie_rec(tab, v, milieu + 1, d)
    Sinon Si t[milieu] > v alors
        Renvoyer dichotomie_rec(tab, v, g, milieu - 1)
    Sinon
        Renvoyer milieu
    FinSi
FinDéfinition
```

```
Définition dichotomie(tab, v):
    Renvoyer dichotomie_rec(tab, v, 0,
                            longueur(t) - 1)
FinDéfinition
```

La fonction *dichotomie* renvoie une position de `v` dans le tableau `tab`, supposé trié, et `Rien` si elle ne s'y trouve pas.

Ce programme termine toujours par le même argument que celui utilisé en première avec la boucle `while` : la valeur entière de la quantité `d - g` diminue à chaque appel récursif. Ainsi, si les appels ne se terminent pas par l'envoyer de l'indice de la valeur dans la liste, à l'étape suivant la quantité `d - g` sera négative et la fonction renverra `Rien`.

On peut également se persuader qu'il n'y a pas de risque d'obtenir l'erreur `RecursionError` à cause d'un trop grand nombre d'appels récursifs. En effet, la taille de l'intervalle étant divisé par deux à chaque étape, il faudrait un tableau de plus de 2^{1000} éléments pour que la fonction *dichotomie* soit appelée plus de 1000 fois. Or, la mémoire d'un ordinateur n'autorise aujourd'hui que les tableaux de quelques milliards d'éléments, c'est-à-dire de l'ordre de 2^{30}.

II - Tri fusion

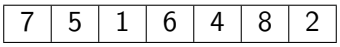
Afin de réinvestir le travail sur les **listes chaînées** et proposer des algorithmes plus généraux et implémentable dans d'autres langages, nous allons considérer, dans cette partie, le tri d'une liste chaînée.

Le tri fusion consiste à trier récursivement les deux moitiés de la liste, puis à fusionner ces deux sous-listes triées en une seule. La condition d'arrêt à la récursivité sera l'obtention d'une liste à un seul élément, car une telle liste est évidemment déjà triée.

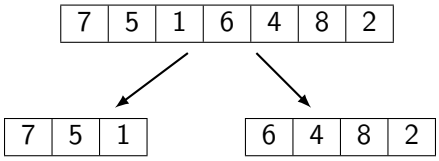
Voici donc les trois étapes (diviser, régner et combiner) de cet algorithme :

- Diviser la liste en deux sous-listes de même taille (à un élément près) en la « coupant » par la moitié.
- Trier récursivement chacune de ces deux sous-listes. Arrêter la récursion lorsque la liste est vide ou n'a plus qu'un seul élément.
- Fusionner les deux sous-listes triées en une seule.

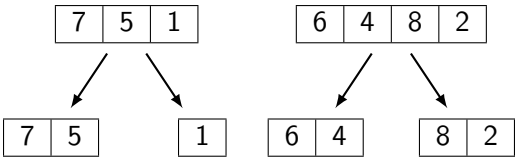
Voici un exemple du déroulement du tri fusion. On considère la liste suivante de sept entiers :



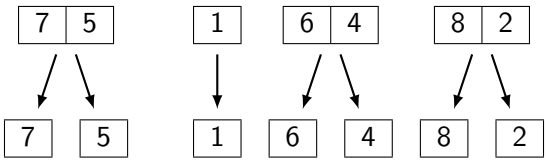
On la scinde en deux sous-listes en la **coupant** par la moitié (la première liste contient un élément de moins que le seconde dans le cas d'une liste de départ contenant un nombre impair d'éléments) :



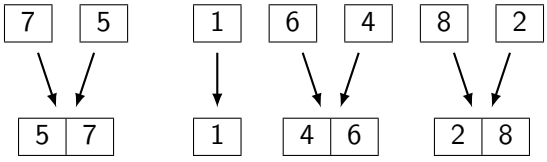
Sous-listes que l'on scinde à leur tour :



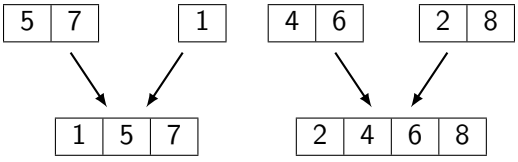
Sous-listes que l'on scinde à leur tour :



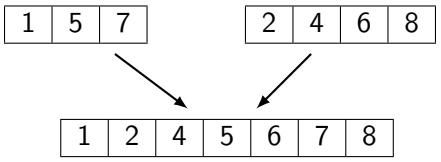
Ces sous-listes sont triées car elles n'ont qu'un élément. On va maintenant les **fusionner** deux par deux en de nouvelles sous-listes triées :



De nouveau une étape de fusion :

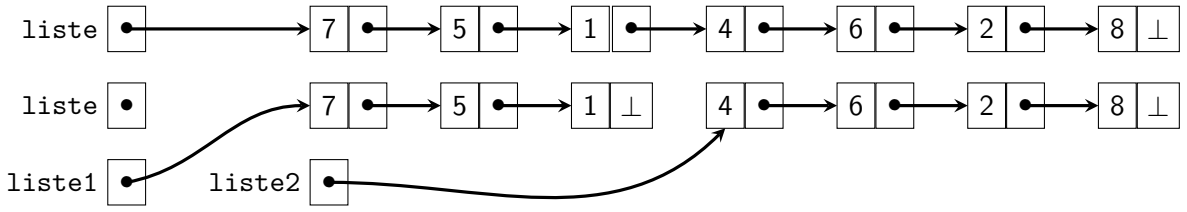


Une dernière fusion :



II - 1. Première étape : scinder la liste

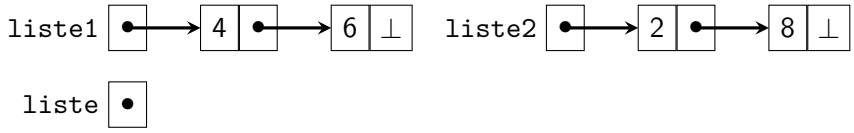
Plutôt que d'écrire l'algorithme, voici un schéma décrivant la fonction à écrire pour réaliser l'opération. Notre fonction prendra en paramètre une liste chaînée `liste` et reverra deux liste `liste1` et `liste2` de même taille, à un élément près :



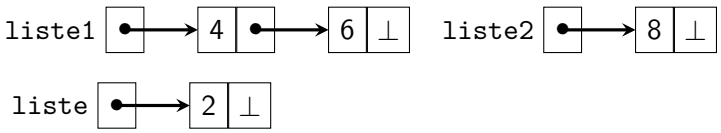
Ainsi, la tête de `liste1` est le premier maillon de `liste` que l'on parcourt jusqu'à sa moitié. On récupère l'adresse du maillon suivant celui du « milieu », sur lequel on vient de s'arrêter qui sera le tête de `liste2`. On termine par remplacer dans le maillon du milieu l'adresse du suivant par le symbole signifiant la fin de la liste.

II - 2. Deuxième étape : la fusion de deux listes triées

Nous devons commencer par vérifier qu'aucune des deux listes n'est pas vide, sinon nous renvoyons celle qui ne l'est pas comme résultat de la fusion. Reprenons l'exemple suivant pour la suite de l'illustration. À la première étape, nous avons deux listes non vides et une liste vide `liste` qui contiendra la liste à renvoyer :

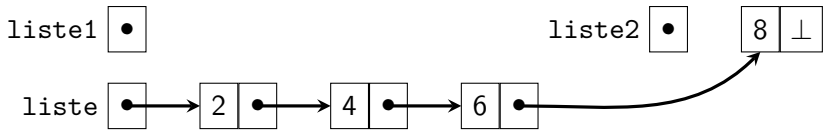


On compare les têtes de chacune des listes, nous supprimons la plus petite valeur de la liste la contenant et l'ajoutons à la suite de la nouvelle liste tant que les deux listes ne sont pas vides :



Puis, on recommence avec les nouvelles têtes des deux listes que l'on ajoute comme nouveau maillon de la liste fusionnée :

Lorsque l'une des deux listes de départ est vide, nous terminons en ajoutant la queue de la liste non vide à la fin de la liste à renvoyer :



II - 3. Le tri fusion

Rappelons le principe :

- Si la liste à trier est vide ou n'a qu'un seul élément, il n'y a rien à faire et nous renvoyons cette liste ;
- Sinon, on **scinde** la liste en deux listes ;
- Nous renvoyons la **fusion** de chacune des deux liste auxquelles on a appliqué le tri fusion récursivement.

Remarque : La fonction `tri_fusion` bien que récursive, ne conduira pas jamais à l'erreur `RecusionError`.

En effet, la taille de la liste étant divisée par deux à chaque fois, il faudrait une liste de plus de 2^{1000} éléments pour conduire à une erreur. Comme nous l'avons déjà indiqué, la mémoire de notre machine ne nous permet pas de construire une liste aussi grande.

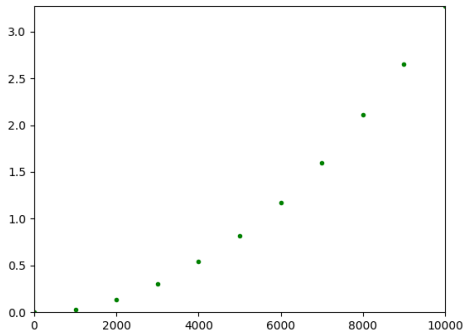
II - 4. Efficacité du tri fusion

Voici un tableau comparatifs des temps observé (sur le serveur NSI) par méthodes de tri vues en première (le tri par sélection et le tri par insertion) avec le tri fusion :

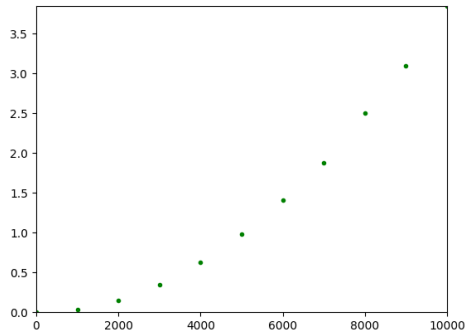
Taille `n`	1 000	2 000	4 000	8 000	16 000	32 000	64 000
tri par sélection	$0,04$ s	$0,15$ s	$0,57$ s	$2,30$ s	$9,93$ s		
tri par insertion	$0,05$ s	$0,19$ s	$0,72$ s	$2,88$ s	$11,40$ s		
tri fusion	$0,01$ s	$0,02$ s	$0,05$ s	$0,10$ s	$0,25$ s	$0,48$ s	$1,04$ s

On peut constater que les performances sont bien meilleurs avec le tri fusion.

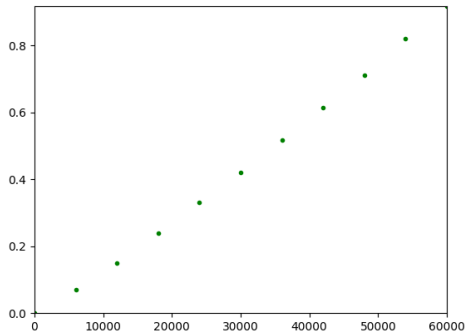
Rappelons que dans le pire des cas, les tris par sélection et par insertion peuvent prendre un temps **quadratique**, c'est-à-dire proportionnel à n^2 où n est le nombre d'éléments à trier. Ainsi, chaque fois que le nombre d'éléments à trier double, le temps est multiplié par quatre. Le tri fusion, en revanche, demande un temps qui est proportionnel à $n \log_2 n$, où $\log_2$ désigne la fonction **logarithme** de base 2. Le logarithme est une fonction qui croît relativement lentement $2^{30} \simeq 10^9$ et $\log_2(2^{30}) = 30$. Cela veut dire que, lorsque le nombre d'éléments à trier double, le coût du tri fusion fait un peu plus que doubler, mais pas beaucoup plus.



Tri par sélection



Tri par insertion



Tri fusion