Tri – Complexité

Ordonner les éléments d'une liste est une activité essentielle en informatique. Par exemple une fois qu'une liste est triée, il est très facile de chercher si elle contient tel ou tel élément. Par définition un algorithme renvoie toujours le résultat attendu, mais certains algorithmes sont plus rapides que d'autres ! Cette efficacité est mesurée par la notion de complexité.

Ce chapitre commence par de la théorie : tout d'abord des rappels sur les suites et l'explication de la notation « grand O ». Ensuite on aborde la notion de complexité qui mesure la performance d'un algorithme. Ceux qui veulent coder peuvent directement s'attaquer aux différents algorithmes de tris présentés. Le bilan est fait dans la dernière activité : comparer les complexités des différents algorithmes de tris.

Cours 1 (Notation « grand O »)**.**
On souhaite comparer deux suites, ou plus exactement leur ordre de grandeur. Par exemple les suites $(n^2)_{n\in\mathbb{N}}$ et $(3n^2)_{n\in\mathbb{N}}$ ont le même ordre de grandeur, mais sont beaucoup plus petites que la suite $(\frac{1}{2}e^n)_{n\in\mathbb{N}}$.
Notation « grand O ».

- On considère $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites de termes strictement positifs.
- On dit que (u_n) est un ***grand O*** de (v_n) si la suite $\left(\frac{u_n}{v_n}\right)$ est bornée.
- Autrement dit il existe une constante réelle $k > 0$ telle que pour tout $n \in \mathbb{N}$:

$$u_n \leqslant k v_n.$$

- *Notation.* On note alors $u_n = O(v_n)$. Il s'agit de la lettre « O » (pour Ordre de grandeur) et pas du chiffre zéro.

Exemples

- Soient $u_n = 3n+1$ et $v_n = 2n-1$. Comme $\frac{u_n}{v_n} \to \frac{3}{2}$ lorsque $n \to +\infty$ alors la suite $\left(\frac{u_n}{v_n}\right)$ est bornée donc $u_n = O(v_n)$.
- $u_n = 2n^2$ et $v_n = e^n$. Comme $\frac{u_n}{v_n} \to 0$ alors la suite $\left(\frac{u_n}{v_n}\right)$ est bornée donc $u_n = O(v_n)$.
- $u_n = \sqrt{n}$ et $v_n = \ln(n)$. Comme $\frac{u_n}{v_n} \to +\infty$ lorsque $n \to +\infty$ alors la suite $\left(\frac{u_n}{v_n}\right)$ n'est pas bornée. (u_n) n'est pas un grand O de (v_n). Par contre dans l'autre sens, on a bien $v_n = O(u_n)$.
- $u_n = O(n)$ signifie qu'il existe $k > 0$ tel que $u_n \leqslant kn$ (pour tout $n \in \mathbb{N}$).
- $u_n = O(1)$ signifie que la suite (u_n) est bornée.

Suites de référence.
On va de préférence comparer une suite (u_n) avec des suites de référence. Voici les suites de référence choisies :

$$\underbrace{\ln(n)}_{\text{croissance logarithmique}} \qquad \underbrace{n \quad n^2 \quad n^3 \quad \cdots}_{\text{croissance polynomiale}} \qquad \underbrace{e^n}_{\text{croissance exponentielle}}$$

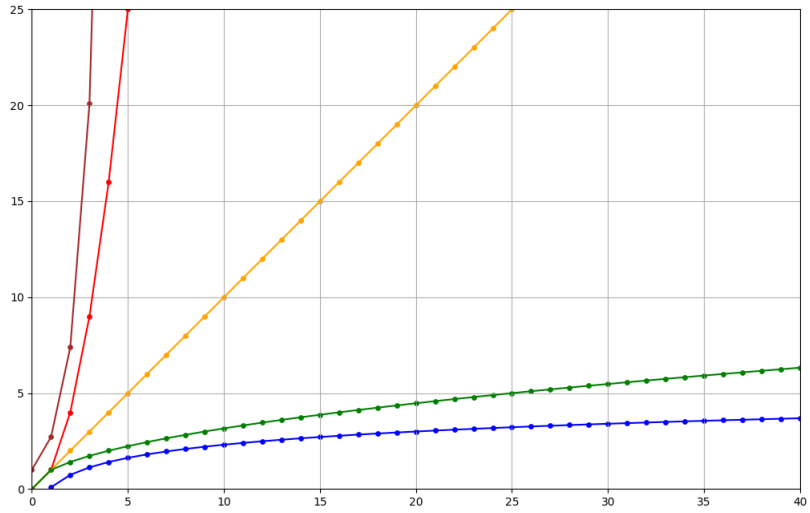
- Les suites sont écrites en respectant l'ordre des O : on a $\ln(n) = O(n)$, $n = O(n^2)$, $n^2 = O(n^3)$, …, $n^3 = O(e^n)$.

- On pourrait intercaler d'autres suites, par exemple $\ln(n) = O(\sqrt{n})$ et $\sqrt{n} = O(n)$. Ou encore $n\ln(n) = O(n^2)$.
- Il est important de savoir visualiser ces suites (voir le graphique de l'activité 1).

Activité 1 (Notation « grand O »)**.**

Objectifs : comparer des suites avec la notation « grand O ».

1. Considère les suites définies par
$$u_n = 1000n^2 \qquad \text{et} \qquad v_n = 0.001\exp(n)$$
Calcule les premiers termes de chaque suite. Penses-tu que $u_n = O(v_n)$ ou bien $v_n = O(u_n)$?
2. Visualise les termes de différentes suites, comme sur le graphique ci-dessous où tu retrouves les termes des suites $\ln(n)$, $\sqrt{n}$, n, n^2, e^n.



Les suites $\ln(n)$, $\sqrt{n}$, n, n^2, e^n.

3. Programme une fonction `est_grand_O(u,v)` qui renvoie « Vrai » si la suite (u_n) est expérimentalement un grand O de (v_n). On dira que (u_n) est expérimentalement un grand O de (v_n) si
$$u_n \leqslant kv_n$$
pour $n \in [10, 1000]$ et $k = 10$. (Bien sûr le choix de ces constantes est arbitraire.)
A-t-on expérimentalement $n^2 = O(2^n)$? Et $n = O(\sqrt{n})$?
Tu peux définir une suite `u` comme une fonction :
```
def u(n): return n**2
```
ou bien
```
u = lambda n: n**2
```
Dans les deux cas on obtient le terme u_n par la commande `u(n)`.

Cours 2 (Complexité d'un algorithme)**.**
On mesure l'efficacité d'un algorithme à l'aide de la complexité.

- Les deux principales caractéristiques qui font qu'un algorithme est bon ou mauvais sont la rapidité d'exécution et l'utilisation de la mémoire. Nous nous limiterons ici à étudier la vitesse d'exécution.

- Comment mesurer la vitesse ? Une durée (en secondes) dépend de chaque ordinateur et n'est pas un indicateur universel.
- Aussi nous définissons de manière informelle la complexité : la ***complexité*** d'un algorithme est le nombre d'opérations élémentaires exécutées.
- Ce que l'on appelle « opération élémentaire » peut varier selon le contexte : pour un calcul cela peut être le nombre de multiplications, pour un tri le nombre de comparaisons…
- La complexité C_n dépend de la taille n des données en entrée (par exemple le nombre de chiffres d'un entier ou bien la longueur de la liste). On obtient ainsi une suite (C_n).
- Les bons algorithmes ont des complexités polynomiales qui sont en $O(n)$ (linéaire), ou en $O(n^2)$ (quadratique) ou bien en $O(n^k)$, $k \in \mathbb{N}^*$ (polynomiale). Les mauvais algorithmes ont des complexités exponentielles, en $O(e^n)$ par exemple.

Multiplication de deux entiers.
On souhaite multiplier deux entiers a et b de n chiffres. Il y a plusieurs méthodes, on les compare en comptant le nombre d'opérations élémentaires : ici des multiplications de petits nombres (entiers à 1 ou 2 chiffres).

Algorithme	Ordre de la complexité
Multiplication d'école	$O(n^2)$
Multiplication de Karatsuba	$O(n^{\log_2(3)}) \simeq O(n^{1.58})$
Transformée de Fourier rapide	$O(n \cdot \ln(n) \cdot \ln(\ln(n)))$

Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n.

Algorithme	$n = 10$	$n = 100$	$n = 1000$
Multiplication d'école	100	10 000	1 000 000
Multiplication de Karatsuba	38	1478	56 870
Transformée de Fourier rapide	19	703	13 350

Plus l'entier n est grand, plus un bon algorithme prend l'avantage.

Recherche dans une liste.
La recherche d'un élément dans une liste non triée nécessite de tester chaque élément de la liste. Si la liste est de longueur n alors il faut $O(n)$ tests. Par contre si la liste est ordonnée alors il existe des algorithmes beaucoup plus efficaces : par exemple la recherche par dichotomie (voir le chapitre « Le mot le plus long »).

Algorithme	Ordre de la complexité
Élément par élément (liste non triée)	$O(n)$
Dichotomie (liste triée)	$O(\log_2(n))$

Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n.

Algorithme	$n = 1000 = 10^3$	$n = 10^6$	$n = 10^9$
Élément par élément	10^3	10^6	10^9
Dichotomie	10	20	30

Problème du voyageur de commerce.
On se donne n villes et les distances entre ces villes. Il s'agit de trouver le plus court chemin qui visite toutes les villes en revenant à la ville de départ. Il n'y pas d'algorithme connu qui soit efficace pour obtenir

la meilleure solution. Un des meilleurs algorithmes a pour complexité $O(n^2 2^n)$. Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n.

Algorithme	$n = 10$	$n = 100$	$n = 1000$
Voyageur de commerce	10^5	10^{34}	10^{307}

On voit que cet algorithme est inutilisable sauf pour de petites valeurs de n.

Cours 3 (Le tri avec `Python`)**.**

- La commande `Python` pour ordonner une liste est `sorted()`. Par exemple avec `liste = [5, 6, 1, 8, 10]`, la commande `sorted(liste)` renvoie la nouvelle liste `[1, 5, 6, 8, 10]` dans laquelle les éléments sont ordonnés du plus petit au plus grand.
  Cela fonctionne aussi avec des chaînes de caractères, pour
  `liste = ['BATEAU', 'ABRIS', 'ARBRE', 'BARBE']`
  alors `sorted(liste)` renvoie `['ABRIS', 'ARBRE', 'BARBE', 'BATEAU']` ordonnée selon l'ordre alphabétique.
- Variante. La méthode `liste.sort()` ne renvoie rien, mais après utilisation de cette méthode, `liste` est ordonnée (on parle de modification en place).
- Pour obtenir un tri dans l'ordre inverse, utilise la commande `sorted(liste, reverse = True)`.
- Variante. `list(reversed(sorted(liste)))`.

Cours 4 (Double affectation avec `Python`)**.**
`Python` permet les affectations multiples, ce qui permet d'échanger facilement le contenu de deux variables.

- **Affectation multiple.**
  ```
  a, b = 3, 4
  ```
  Maintenant `a` vaut 3 et `b` vaut 4.
- **Échange de valeurs.**
  ```
  a, b = b, a
  ```
  Maintenant `a` vaut l'ancien contenu de `b` donc vaut 4 et `b` vaut l'ancien contenu de `a` donc 3.
- **Échange à la main.** Pour échanger deux valeurs sans utiliser la double affectation, il faut introduire une variable temporaire :

```
temp = a
a = b
b = temp
```

Activité 2 (Tri par sélection)**.**

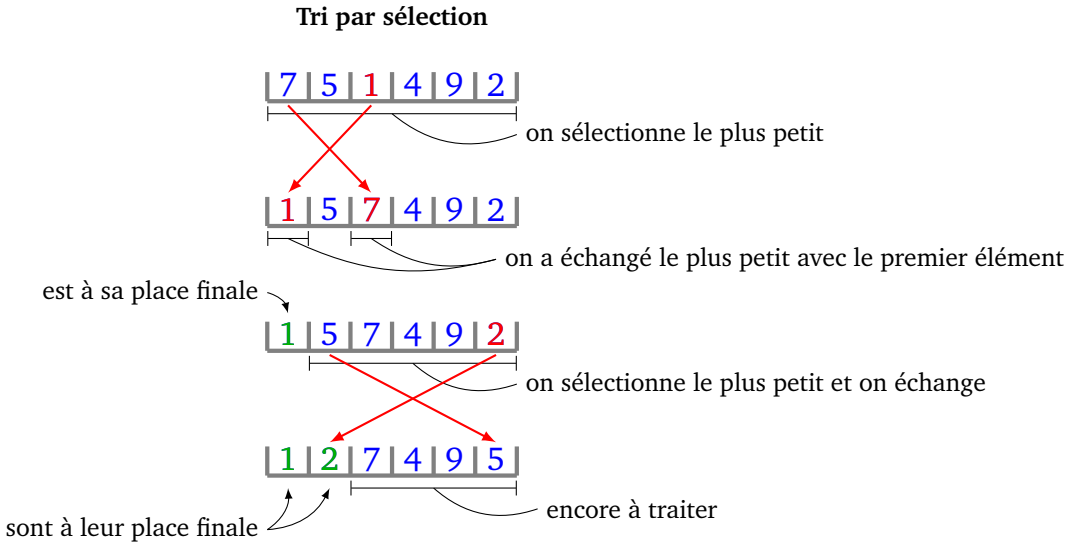
Objectifs : programmer le « tri par sélection » qui est un algorithme très simple.

Il s'agit d'ordonner les éléments d'une liste du plus petit au plus grand. On note n la longueur de la liste. Les éléments sont donc indexés de 0 à $n-1$.

Algorithme.

- — Entrée : une liste de longueur n.
  — Sortie : la liste ordonnée.
- Pour i variant de 0 à $n-1$:
  — *Recherche du plus petit élément après le rang i :*
  — `rg_min` $\leftarrow i$
  — pour j allant de $i+1$ à $n-1$:
  si `liste[j]` $<$ `liste[rg_min]` faire `rg_min` $\leftarrow j$.
  — *Échange.* Échanger l'élément de rang i avec l'élément de rang `rg_min`.
- Renvoyer la liste.

Explications. L'algorithme est très simple : on cherche le plus petit élément de la liste et on le place en tête. Le premier élément est donc à sa place. On recommence avec le reste de la liste : on cherche le plus petit élément que l'on positionne en deuxième place...



Travail à faire. Programme cet algorithme en une fonction `tri_selection(liste)`.
Indications. La fonction ne doit pas modifier la liste passée en paramètre. Pour éviter les désagréments :

- commence par faire une copie de ta liste :

```
cliste = list(liste)
```

- ne travaille qu'avec `cliste`, que tu peux modifier à volonté,
- renvoie `cliste`.

Commentaires. Le principal avantage de cet algorithme est sa simplicité. Sinon il est de complexité $O(n^2)$ ce qui en fait un algorithme de tri lent réservé pour les petites listes.

Activité 3 (Tri par insertion)**.**

Objectifs : programmer le « tri par insertion » qui est un algorithme très simple.

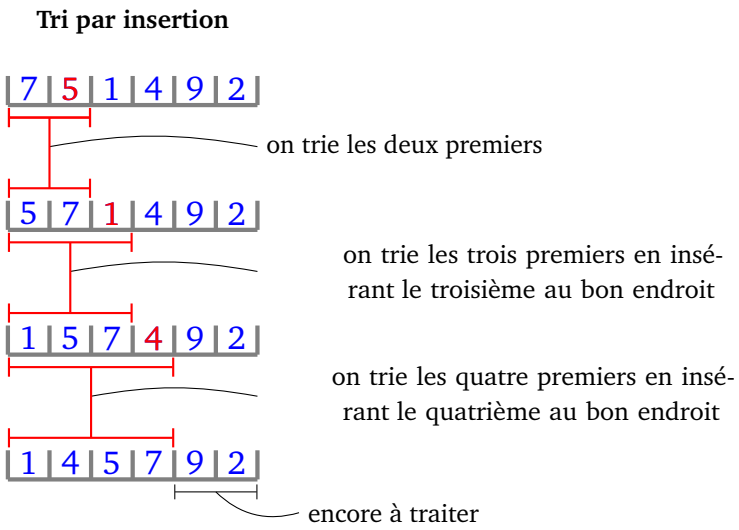
Le tri par insertion est assez naturel : c'est le tri que tu utilises par exemple pour trier un jeu de cartes. Tu prends les deux premières cartes, tu les ordonnes. Tu prends la troisième carte, tu la places au bon

endroit pour obtenir trois cartes bien ordonnées. Tu prends une quatrième carte que tu places au bon endroit pour obtenir quatre cartes bien ordonnées…

Algorithme.

- — Entrée : une liste de longueur n.
  — Sortie : la liste ordonnée.
- Pour i variant de 1 à $n-1$:
  — `el` ← `liste[i]` *(on mémorise l'élément pivot de rang i)*
  — *On décale vers la droite tous les éléments de rang $i-1$ à 0 qui sont plus grands que le pivot :*
  — $j \leftarrow i$
  — Tant que ($j > 0$) et (`liste[j-1]` > `el`) :
    — `liste[j]` ← `liste[j-1]`
    — $j \leftarrow j-1$
  — `liste[j]` ← `el` *(on replace l'élément pivot dans le trou créé par le décalage)*
- Renvoyer la liste.

Explications. L'algorithme est assez simple : on regarde les deux premiers éléments, s'ils sont dans le mauvais sens on les échange (les deux premiers éléments sont maintenant bien ordonnés entre eux). On regarde ensuite les trois premiers éléments, on insère le troisième élément à la bonne place parmi ces trois éléments (qui sont maintenant bien ordonnés entre eux). On recommence avec les quatre premiers éléments : on insère le quatrième élément à la bonne place parmi ces quatre éléments…
Le dernier élément du groupe considéré est appelé pivot, pour l'insérer on décale d'un rang vers la droite tous les élément situés avant lui qui sont plus grands. On obtient donc un trou qui est la place du pivot.



Travail à faire. Programme cet algorithme en une fonction `tri_insertion(liste)`.

Commentaires. C'est un bon algorithme dans la catégorie des algorithmes de tri lents ! Il est de complexité $O(n^2)$, mais sur une liste déjà un peu ordonnée il est efficace. Il est un peu meilleur que le tri par sélection. Il permet aussi d'ordonner des éléments en « temps réels » : on peut commencer à trier le début de la liste sans connaître la fin.

Activité 4 (Tri à bulles)**.**

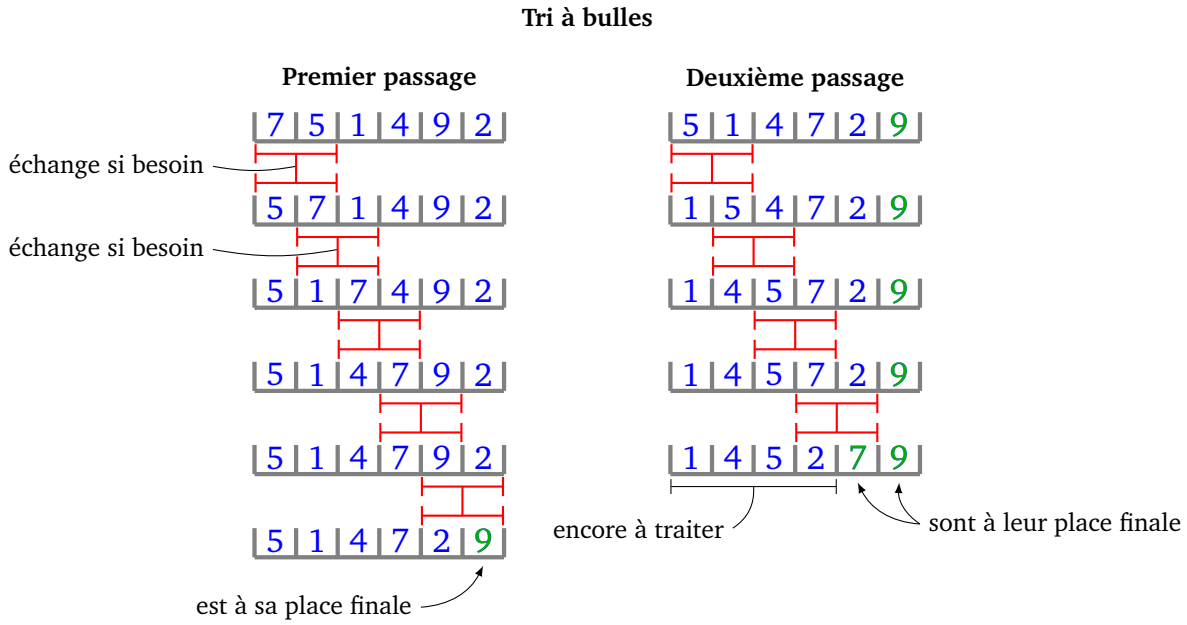
Objectifs : programmer le « tri à bulles ».

Le tri à bulles est très simple à programmer : il s'agit d'échanger deux termes consécutifs s'ils ne sont pas dans le bon ordre. Le nom vient de l'analogie avec les bulles d'eau qui remontent à la surface comme ici les éléments qui viennent se positionner à leur place.

Algorithme.

- — Entrée : une liste de longueur n.
  — Sortie : la liste ordonnée.
- Pour i allant de $n-1$ à 0 :
  Pour j allant de 0 à $i-1$:
  Si `liste[j+1]` $<$ `liste[j]` alors :
  échanger `liste[j]` et `liste[j+1]`.
- Renvoyer la liste.

Explications. L'algorithme est très simple : on compare deux éléments consécutifs et on les échange s'ils sont dans le mauvais ordre. On continue avec les couples suivants jusqu'à la fin de la liste. Au bout du premier passage le dernier élément est définitivement à sa place. On recommence en partant du début avec un second passage, maintenant les deux derniers éléments sont à leur place.



Travail à faire. Programme cet algorithme en une fonction `tri_a_bulles(liste)`.

Commentaires. Le tri à bulles est très simple à programmer, cependant il fait aussi partie des algorithmes de tri lents car sa complexité est en $O(n^2)$.

Activité 5 (Tri fusion)**.**

Objectifs : programmer un tri beaucoup plus efficace : le « tri fusion ». Par contre sa programmation est plus compliquée car l'algorithme est récursif.

Le tri fusion est un tri rapide. Il est basé sur le principe de « diviser pour régner » ce qui fait que sa programmation naturelle se fait par une fonction récursive. Le principe est simple : on divise la liste en deux parties ; on trie la liste de gauche (par un appel récursif) ; on trie la liste de droite (par un autre appel récursif) ; ensuite il faut fusionner ces deux listes en intercalant les termes.
Ce tri se programme à l'aide de deux fonctions : une fonction principale `tri_fusion(liste)` qui trie la liste. Cette fonction nécessite la fonction secondaire `fusion(liste_gauche,liste_droite)` qui fusionne deux listes triées en une seule.

On commence par la fonction principale qui est une fonction récursive.

Algorithme.

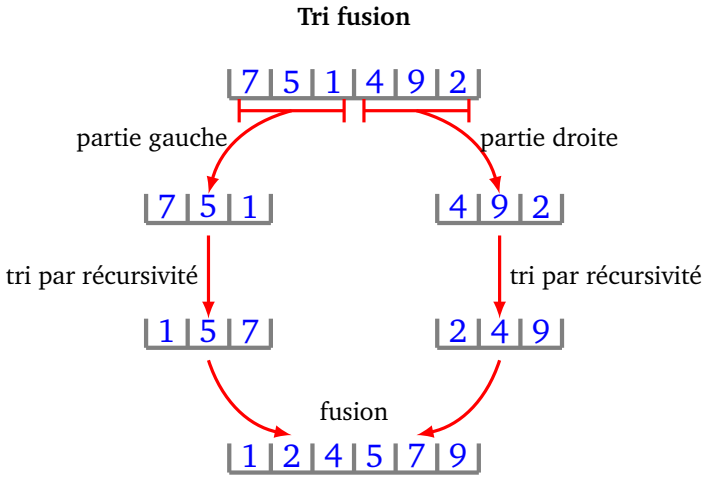
- — Entête : `tri_fusion(liste)`
  — Entrée : une liste de longueur n.
  — Sortie : la liste ordonnée.
  — Action : fusion récursive.
- *Cas terminal.* Si la liste est de longueur 0 ou 1, renvoyer la liste telle quelle.
- *Cas général.*
- Calculer `liste_g = tri_fusion(liste[:n//2])`. On prend les éléments de gauche (de rang $< n//2$) et on les trie par un appel récursif.
- Calculer `liste_d = tri_fusion(liste[n//2:])`. On prend les éléments de droite (de rang $\geqslant n//2$) et on les trie par un appel récursif.
- Renvoyer la liste `fusion(liste_g,liste_d)`.

La fonction précédente nécessite la fonction définie par l'algorithme suivant :

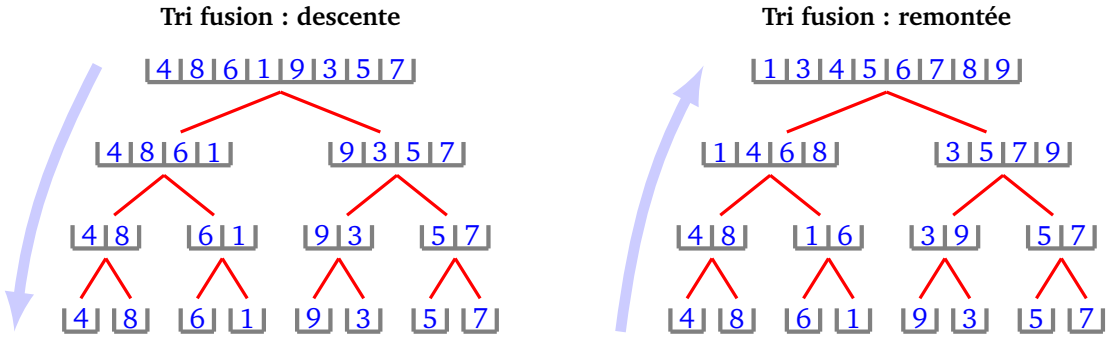
Algorithme.

- — Entête : `fusion(liste_g,liste_d)`
  — Entrée : deux listes ordonnées : `liste_g` de longueur n et `liste_d` de longueur m.
  — Sortie : une liste fusionnée et ordonnée.
- — Un indice i, initialisé à 0, indexe la première liste,
  — un indice j, initialisé à 0, indexe la seconde liste,
  — une liste `liste_fus` est initialisée à la liste vide.
- *Fusion principale.*
  Tant que $(i < n)$ et $(j < m)$, faire :
  — si `liste_g[i]` $<$ `liste_d[j]` ajouter `liste_g[i]` à `liste_fus` et incrémenter i,
  — sinon ajouter `liste_d[j]` à `liste_fus` et incrémenter j.
- *S'il reste des termes.*
  — Tant que $i < n$, ajouter `liste_g[i]` à `liste_fus` et incrémenter i.
  — Tant que $j < m$, ajouter `liste_d[j]` à `liste_fus` et incrémenter j.
- Renvoyer `liste_fus`.

Explications. La seconde fonction `fusion()` regroupe deux listes ordonnées en une seule liste ordonnée. La fonction `tri_fusion(liste)` est très simple, comme nous l'avons expliqué précédemment : la liste est découpée en une partie droite et une partie gauche. Grâce à deux appels récursifs chacune de ces sous-parties est triée. Il ne reste plus qu'à fusionner ces deux listes.



Bien sûr, comme toute fonction récursive, c'est assez difficile d'appréhender l'enchaînement complet des instructions.
Pourquoi cet algorithme est-il plus performant que les précédents ? Ce n'est pas facile à expliquer ! Le point-clé se passe lors la fusion : quand on fusionne deux listes ordonnées de longueur n on effectue des comparaisons `liste_g[i]` $<$ `liste_d[j]`, mais on ne compare pas tous les éléments entre eux (ce qui donnerait n^2 comparaisons) mais seulement pour quelques couples (i, j) ce qui donne $2n$ comparaisons.



Travail à faire. Programme ces algorithmes en deux fonctions `fusion(liste_g,liste_d)` et `tri_fusion(liste)`.

Commentaires. Le tri fusion est un tri rapide : sa complexité est $O(n\ln(n))$ ce qui est beaucoup mieux que les algorithmes précédents et est asymptotiquement optimal. Il est conceptuellement simple à comprendre et à programmer si on connaît la récursivité.

Activité 6 (Complexité des algorithmes de tri).

Objectifs : comparer expérimentalement les complexités des algorithmes de tri.

Pour les quatre algorithmes de tri que tu as programmés, modifie tes fonctions afin qu'elles renvoient en plus de la liste triée le nombre de comparaisons effectuées entre deux éléments de la liste. Chaque test du type « `liste[i]` $<$ `liste[j]` » compte pour une comparaison.

- Pour les trois premiers algorithmes, c'est assez facile ; pour le dernier c'est plus compliqué (voir ci-dessous).
- Compare le nombre de comparaisons effectuées :

  - quand la liste est une liste d'éléments tirés au hasard ;
  - quand la liste est déjà triée ;
  - quand la liste est déjà triée mais en sens inverse.
- On note n la longueur de la liste. Compare la complexité avec n^2 (ou mieux $n^2/2$) pour les algorithmes lents et $n\ln(n)$ (ou mieux $n\log_2(n)$) pour l'algorithme de tri fusion.

Indications pour le tri fusion. Il faut commencer par modifier la fonction `fusion()` en une fonction `fusion_complexite()` qui renvoie en plus de la liste, le nombre de comparaisons effectuées lors de cette étape. Il faut ensuite modifier la fonction `tri_fusion()` en une fonction `tri_fusion_complexite()` qui renvoie en plus le nombre de comparaisons. Pour cela il faut compter le nombre de comparaisons venant du tri fusion de la partie gauche de la liste, le nombre de comparaisons venant du tri fusion de la partie droite de la liste, et enfin le nombre de comparaisons venant de la fusion. Il faut finalement renvoyer la somme de ces trois nombres.

Une jolie activité consiste à visualiser pas à pas le tri d'une liste pour chaque algorithme. Voir les pages Wikipédia *pour un tel exemple d'animation.*