TP 18 d'informatique - MPSI - 29/04/2025 - Algorithmes de tri

Dans ce TP, on s'intéresse à des algorithmes de tris. Étant donnée une liste `L`, il s'agit de réorganiser `L` pour obtenir une liste dont les éléments sont rangés par ordre croissant. Par exemple en Python, les commandes `sorted(L)` et `L.sort()` trient la liste `L`.

Vocabulaire 1. L'ordre utilisé habituellement sur les chaînes de caractères et les tuples s'appelle ***l'ordre lexicographique***. Cet ordre suit la même logique que l'ordre alphabétique :

```
L1 = ["maison", "chien", "python", "chat"]
print(sorted(L1))  # Affiche ['chat', 'chien', 'maison', 'python']
L2 = [(1,8), (1,2,3), (2,0), (1,2)]
print(sorted(L2))  # Affiche [(1,2), (1,2,3), (1,8), (2,0)]
```

Remarque 2. Les énoncés de ce TP s'intéressent uniquement à des listes d'entiers. Toutefois, la plupart des fonctions écrites pourront être utilisées avec des listes de flottants, des listes de couples, des listes de chaînes de caractères ...

Vocabulaire 3. ***Un tri en place*** est un algorithme de tri dont la complexité spatiale est en $\mathcal{O}(1)$. Ainsi, si on note `L` la liste à trier, cette définition implique que :

→ Un tri en place doit nécessairement modifier `L` et ne peut pas créer de nouvelle liste pour y copier les éléments de `L`. En effet, créer une nouvelle liste nécessite une complexité spatiale au moins linéaire.

→ Toutes les variables locales doivent utiliser une quantité de mémoire constante (interdiction de créer une liste, un dictionnaire ou une chaîne de caractères).

```
L1 = [5,7,2,6,3]
T1 = sorted(L1)  # sorted n'est pas un tri en place: une autre liste T1 est créée
print(L1)        # Affiche [5,7,2,6,3]
print(T1)        # Affiche [2,3,5,6,7]

L2 = [5,7,2,6,3]
L2.sort()        # .sort est un tri en place: il renvoie None
print(L2)        # Affiche [2,3,5,6,7]. La liste initiale n'est plus accessible.
```

Vocabulaire 4. ***Un tri par comparaisons*** choisit la manière de trier `L` en se basant uniquement sur le résultat de comparaisons entre éléments de la liste. Par exemple, le tri par insertion est un tri par comparaisons (exercice 2), mais ce n'est pas le cas du tri par comptage (exercice 4). Un tri par comparaisons permet de trier n'importe quel type d'objets sur lequel les opérateurs de comparaisons (`<`, `<=`, ...) sont définis. À l'inverse, le tri par comptage de l'exercice 4 ne permet de trier que des entiers.

Proposition 5 (admise)**.** *Le temps d'exécution d'un tri par comparaisons est au moins en $\mathcal{O}(n.\log(n))$ où n est la taille de la liste. Cette borne est atteinte pour le tri fusion (exercice 9) et peut être battue par un tri qui n'est pas par comparaisons.*

Vocabulaire 6. Un ***tri stable*** est un tri dans lequel l'ordre de deux éléments égaux est le même dans la liste initiale et dans la liste triée. Par exemple, supposons disposer d'une liste contenant des noms d'élèves associés à une note :

```
[("Léa", 15), ("Grégoire", 11), ("Hugo", 17), ("Hélène", 17), ("Clément", 14)]
```

Lorsqu'on trie la liste en fonction de la note, un tri stable renverra nécessairement :

```
[("Grégoire", 11), ("Clément", 14), ("Léa", 15), ("Hugo", 17), ("Hélène", 17)]
```

Un tri qui n'est pas stable peut potentiellement échanger les élèves `"Hugo"` et `"Hélène"` :

```
[("Grégoire", 11), ("Clément", 14), ("Léa", 15), ("Hélène", 17), ("Hugo", 17)]
```

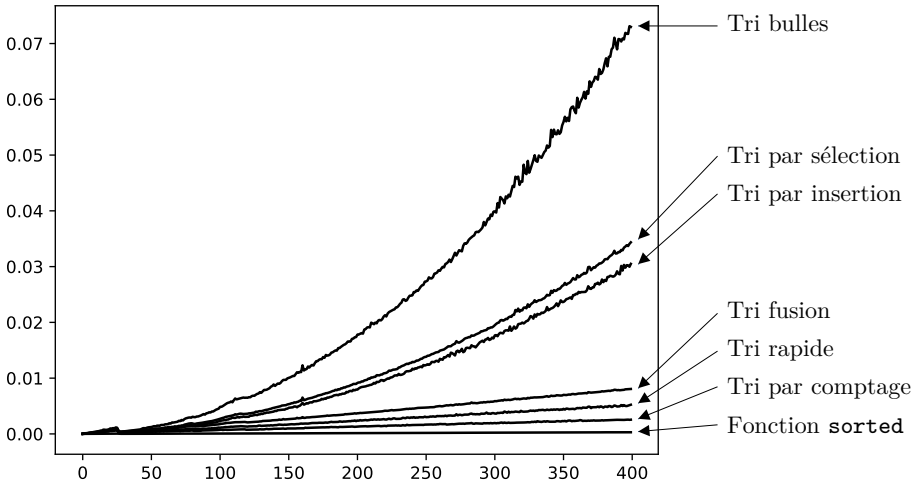


Figure : Comparaison des temps d'exécutions des différents algorithmes de tri étudiés dans ce TP. Ce graphique présente en fonction de n le temps nécessaire pour trier 10 listes de taille n composées d'entiers choisis aléatoirement dans $[\![-2n; 2n]\!]$.

Pensez à organiser vos programmes avec des fonctions intermédiaires et à tester vos fonctions sur la liste vide.

Exercice 1. Tri par sélection

Le principe du tri par sélection est de créer une liste `T` initialement vide et d'ajouter à la fin de `T` le plus petit élément de `L`, puis le deuxième plus petit élément de `L`, puis le troisième plus petit, et ainsi de suite jusqu'à ce que `T` contienne tous les éléments de `L`. Par exemple, voici l'évolution de `T` lors du tri de la liste `[5, 8, 1, 6, 5, 7]` :

`[]` → `[1]` → `[1,5]` → `[1,5,5]` → `[1,5,5,6]` → `[1,5,5,6,7]` → `[1,5,5,6,7,8]`

1. Écrire une fonction `tri_selection` qui trie une liste d'entiers à l'aide du tri par sélection. Si besoin, on pourra utiliser les indications ci-dessous. Tester avec une liste contenant plusieurs fois le même élément.
2. Quelle est la complexité de votre fonction ?
3. Est-ce un tri en place ? Un tri stable ? Expliquer comment modifier votre programme pour passer d'un tri stable à un tri non stable (ou inversement).

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On pourra écrire les fonctions intermédiaires suivantes :

(a) Une fonction `mini` qui prend en entrée une liste `L` et renvoie l'indice du plus petit élément de `L`. Si le plus petit élément apparaît plusieurs fois, on renverra le plus petit indice possible. Si la liste `L` est vide votre fonction déclenchera une erreur.

(b) Une fonction `supprimer` qui prend en entrée une liste `L` ainsi qu'un indice `i` et renvoie la liste `L` privée de `L[i]`. Si la condition `i` $\in [\![$`0; len(L) - 1`$]\!]$ n'est pas vérifiée, votre fonction déclenchera une erreur.

Exercice 2. Tri par insertion

Le principe du tri par insertion est de créer une liste `T` initialement vide et d'y ajouter `L[0]`, puis `L[1]`, puis `L[2]`, ... À chaque ajout, on fait en sorte que `T` reste triée. Par exemple, voici l'évolution de `T` lors du tri par insertion de la liste `[5, 8, 1, 6, 5, 7]` :

`[]` → `[5]` → `[5,8]` → `[1,5,8]` → `[1,5,6,8]` → `[1,5,5,6,8]` → `[1,5,5,6,7,8]`

1. Écrire une fonction `tri_insertion` qui trie une liste d'entiers à l'aide du tri par insertion. À chaque étape, on utilisera une boucle `while` pour repérer l'indice auquel il faut insérer l'élément. Si besoin, on pourra utiliser les indications ci-dessous.
2. Quelle est la complexité de votre fonction ?

3. Est-ce un tri en place ? Un tri stable ? Expliquer comment modifier votre programme pour passer d'un tri stable à un tri non stable (ou inversement).
4. Supposons qu'à chaque étape, l'indice d'insertion soit calculé à l'aide d'une recherche dichotomique. Quelle est la complexité du tri par insertion dans ce cas ?

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On pourra écrire une fonction intermédiaire « `insertion(T: list[int], x: int) -> list[int]` » qui prend en entrée une liste triée `T` et renvoie une liste triée contenant les mêmes éléments que `T + [x]`. Votre fonction devra être de complexité linéaire en `len(T)`.

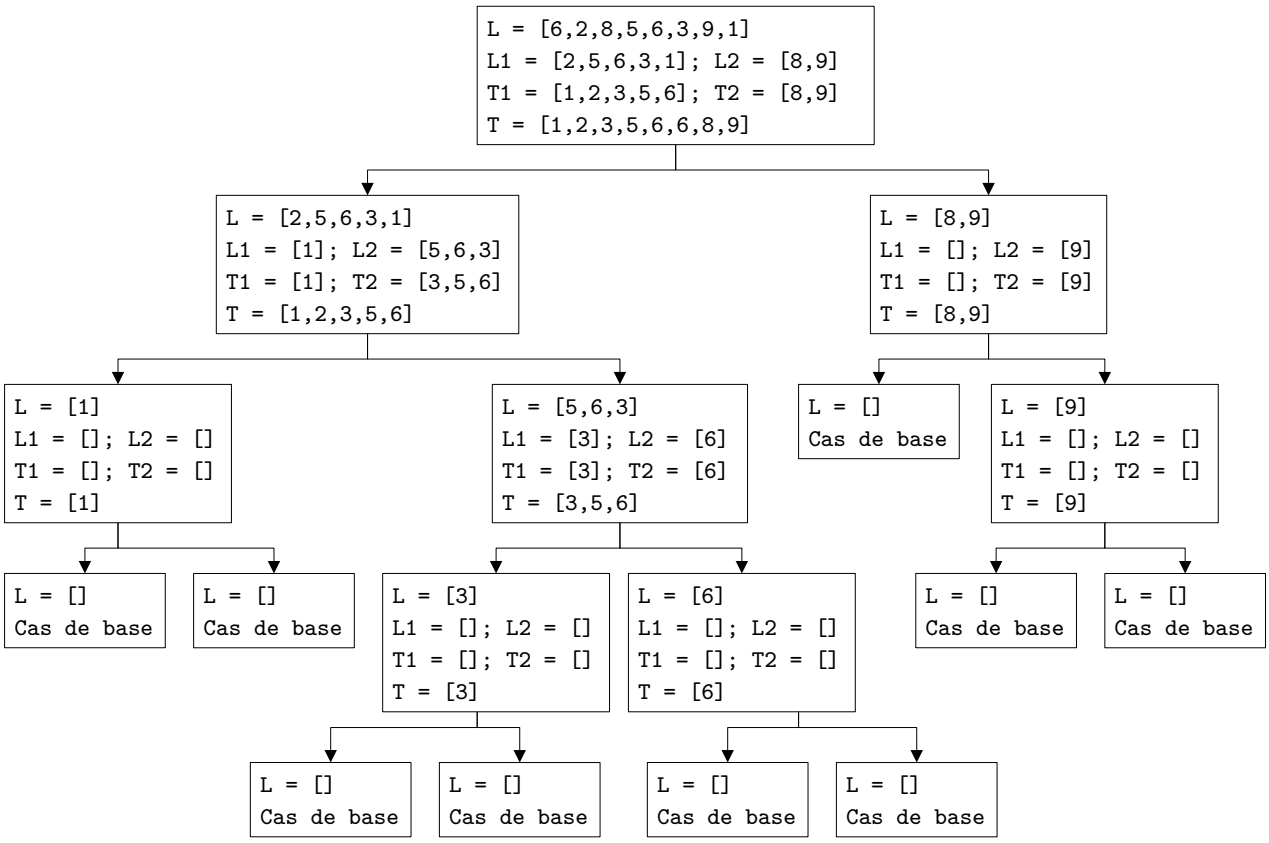
Exercice 3. Tri rapide

Le tri rapide est un algorithme récursif. Si la liste `L` à trier est vide, il suffit de renvoyer la liste vide. Sinon, on choisit l'un des éléments de `L` noté `e` qu'on appelle le pivot (on peut par exemple choisir `L[0]`). Soit `L0` la liste `L` privée de `e`, soit `L1` la liste des éléments de `L0` inférieurs ou égaux à `e` et `L2` la liste des éléments de `L0` strictement supérieurs à `e`. Soient `T1` et `T2` les listes obtenues lorsqu'on trie récursivement `L1` et `L2`. En utilisant `T1` et `T2`, on peut alors calculer une liste triée `T` contenant les mêmes éléments que `L`. L'arbre des appels récursifs lors du tri de la liste `[6,2,8,5,6,3,9,1]` est donné à la fin de l'énoncé.

1. Écrire une fonction `tri_rapide` qui trie une liste d'entiers à l'aide du tri rapide. Si besoin, on pourra utiliser les indications ci-dessous.
2. Quelle est la complexité de votre fonction dans le pire cas ?
3. Est-ce un tri en place ? Un tri stable ? Expliquer comment modifier votre programme pour passer d'un tri stable à un tri non stable (ou inversement).

Remarque. Pour tenter d'améliorer le temps d'exécution, on peut choisir le pivot d'une autre manière. Par exemple, choisir l'un des éléments de `L` au hasard, ou bien choisir la médiane des éléments (voir l'exercice 10).

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On pourra écrire une fonction intermédiaire « `separer(L: list[int]) -> (list[int], list[int])` » qui prend en entrée une liste non vide `L` et renvoie les listes `L1` et `L2` dans le cas où `L[0]` est le pivot.



Exercice 4. Tri par comptage

Pour effectuer un tri par comptage d'une liste `L`, on suppose dans un premier temps que tous les éléments de `L` sont positifs ou nuls. Soit `m` l'élément maximum de `L`. On créé une liste `M` de taille `m+1` telle que pour tout `i` $\in [\![0; m]\!]$, l'entier `M[i]` est égal au nombre d'occurrences de `i` dans `L`. Finalement, en parcourant les éléments de `M`, on peut en déduire une liste triée `T` contenant les mêmes éléments que `L`. Par exemple :

```
L = [1,0,2,1,4,5,1,2]        M = [1,3,2,0,1,1]        T = [0,1,1,1,2,2,4,5]
```

1. Écrire une fonction `tri_comptage` qui trie une liste d'entiers positifs à l'aide d'un tri par comptage. Si besoin, on pourra utiliser les indications ci-dessous.
2. Adapter le tri par comptage dans le cas d'une liste d'entiers relatifs.

Indications (essayez de résoudre l'exercice sans lire ce qui suit).

(a) Écrire une fonction « `maxi(L: list[int]) -> int` » qui renvoie l'élément maximal de `L`. On pourra supposer que `L` est non vide.

(b) Écrire une fonction « `comptage(L: list[int]) -> list[int]` » qui renvoie la liste `M`.

Vous pouvez faire les exercices 5, 6, 7 dans l'ordre que vous souhaitez.

Exercice 5. Tri de crêpes

Un forain présent lors de la foire de Nancy a préparé une pile de crêpes et veut les trier en fonction de leurs diamètres (la plus grande en dessous et la plus petite au dessus). Par exemple en partant de la figure 1 ou de la figure 2, il souhaite obtenir la figure 3 :

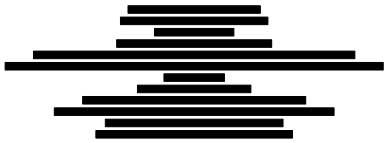


FIGURE 1

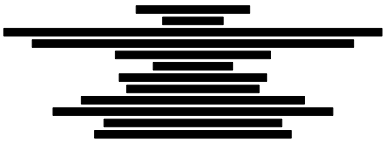


FIGURE 2

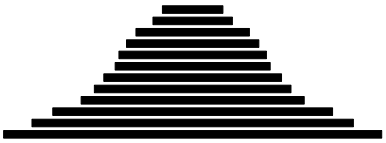


FIGURE 3

Soit $n \in \mathbb{N}$ le nombre de crêpes. On indice les crêpes de 0 à $n-1$, celle d'indice 0 étant celle se trouvant en dessous. Pour effectuer son tri, le forain dispose de deux opérations :

⋆ Opération 1 – À partir d'un indice $i \in [\![0, n-1]\!]$, il peut trouver l'indice $j \geqslant i$ correspondant à une crêpe de diamètre maximal. Avec l'exemple de la figure 1 :

i	0	1	2	3	4	5	6	7	8	9	10	11
j	6	6	6	6	6	6	6	7	8	10	10	11

Pour cela, il suffit de repérer la crêpe qui dépasse par rapport aux autres.

⋆ Opération 2 – Étant donné un indice $i \in [\![0, n-1]\!]$, le forain peut placer sa spatule sous la crêpe d'indice i et retourner la pile se trouvant au dessus de la spatule. Par exemple avec $i = 4$ et la pile de la figure 1, on obtient la figure 2.

En Python, une pile de crêpes sera représentée par la liste des diamètres en commençant par la crêpe d'indice 0. Ainsi, la figure 1 correspond à :

```
[5.2, 4.7, 7.4, 5.9, 3, 1.6, 10, 8.5, 4.1, 2.1, 3.9, 3.5]
```

1. Dans cette question, la liste « `L: list[float]` » représente la pile de crêpes.
   (a) Écrire une fonction « `maxi(L: list[float], i: int) -> int` » correspondant à l'opération 1.
   (b) Écrire une fonction « `spatule(L: list[float], i: int) -> NoneType` » correspondant à l'opération 2. La fonction doit modifier `L` directement, sans créer de liste intermédiaire et ne doit rien renvoyer. Pour tester cette fonction on écrira :

```
L = [...]; spatule(L, ...); print(L)
```

2. (a) Trouver une procédure pour trier la pile de crêpes en utilisant uniquement les opérations 1 et 2. On attend une explication en français, pas un programme Python.
   (b) Donner en le démontrant un majorant sur le nombre d'utilisations des opérations 1 et 2 dans la procédure de la question 2a.
3. (a) Écrire une fonction « `tri_crepes(L: list[float]) -> NoneType` » correspondant à la procédure décrite dans la question 2a. Cette fonction doit modifier `L` et ne rien renvoyer.
   (b) Quelle est la complexité de la fonction `tri_crepes` ?
4. Écrire une fonction « `tri_crepes_bis(L: list[float]) -> list[float]` » qui effectue le tri crêpes. La différence avec la fonction de la question 3a est qu'elle doit renvoyer une nouvelle liste et ne pas modifier `L`.

Le forain n'est pas complètement satisfait de la fonction `tri_crepes` car il pense avoir utilisé l'opération 2 plus de fois que nécessaire. Pour toute liste de crêpes P, on note $op_2(P)$ le nombre minimal d'utilisations de l'opération 2 pour trier P. Pour tout entier n, on note M_n le maximum des $op_2(P)$ lorsque P parcourt toutes les piles de crêpes de taille n. Par exemple :

n	0	1	2	3	4	5	6	7	8	9	10
M_n	0	0	1	3	4	5	7	8	9	10	11

Ainsi, n'importe quelle pile composée de 6 crêpes peut être triée en utilisant au plus 7 fois l'opération 2. De plus, il existe une pile composée de 6 crêpes qui ne peut pas être triée en utilisant moins de 7 fois l'opération 2.

5. Écrire une fonction `nb_max_op2` qui prend en entrée $n \in \mathbb{N}$ et renvoie M_n.

Exercice 6. Tri à bulles

Le tri à bulles se décompose en `n-1` étapes. Soit `e` $\in [\![1, \texttt{n-1}]\!]$ un entier qui va représenter un numéro d'étape. Lors de l'étape numéro `e`, le principe est de ne s'intéresser qu'aux éléments `L[0]`, `L[1]`, ..., `L[n-e]` et de faire en sorte qu'à la fin de l'étape, l'élément `L[n-e]` soit le maximum de `L[0]`, `L[1]`, ..., `L[n-e]`. Pour cela, on échange deux éléments consécutifs de la liste s'ils ne sont pas rangés dans l'ordre croissant, en commençant par les éléments d'indices `0` et `1`, puis les éléments d'indices `1` et `2` et ainsi de suite jusqu'aux éléments d'indices `n-e-1` et `n-e`.

Par exemple, dans le cas particulier où la liste initiale vaut `[6,2,1,7,9,0,7]`, voici l'évolution de cette liste lors de la première étape (les deux éléments comparés sont soulignés) :

$$\texttt{[}\underline{\texttt{6,2}}\texttt{,1,5,9,0,7]} \to \texttt{[2,}\underline{\texttt{6,1}}\texttt{,5,9,0,7]} \to \texttt{[2,1,}\underline{\texttt{6,5}}\texttt{,9,0,7]}$$
$$\to \texttt{[2,1,5,}\underline{\texttt{6,9}}\texttt{,0,7]} \to \texttt{[2,1,5,6,}\underline{\texttt{9,0}}\texttt{,7]} \to \texttt{[2,1,5,6,0,}\underline{\texttt{9,7}}\texttt{]} \to \texttt{[2,1,5,6,0,7,9]}$$

Voici l'évolution de la liste lors de la deuxième étape :

$$\texttt{[}\underline{\texttt{2,1}}\texttt{,5,6,0,7,9]} \to \texttt{[1,}\underline{\texttt{2,5}}\texttt{,6,0,7,9]} \to \texttt{[1,2,}\underline{\texttt{5,6}}\texttt{,0,7,9]}$$
$$\to \texttt{[1,2,5,}\underline{\texttt{6,0}}\texttt{,7,9]} \to \texttt{[1,2,5,0,}\underline{\texttt{6,7}}\texttt{,9]} \to \texttt{[1,2,5,0,6,7,9]}$$

Notez que les deux derniers éléments ne sont pas comparés lors de la deuxième étape.

1. Écrire une fonction « `tri_bulles(L: list[int]) -> NoneType` » qui trie une liste d'entiers à l'aide du tri à bulles. Votre fonction doit modifier la liste donnée en entrée et ne rien renvoyer. Si besoin, on pourra utiliser les indications ci-dessous.
2. Quelle est la complexité de votre fonction ?
3. Est-ce un tri en place ? Un tri stable ? Expliquer comment modifier votre programme pour passer d'un tri stable à un tri non stable (ou inversement).

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On pourra écrire une fonction intermédiaire `etape_tri_bulles` qui prend en entrée une liste `L` ainsi qu'un entier `i_max` et modifie la liste `L` en comparant les éléments d'indices `0` et `1`, puis les éléments d'indices `1` et `2` ... puis les éléments d'indices `i_max` et `i_max + 1`.

Exercice 7. Le tri du singe

Le tri du singe, aussi appelé le tri stupide, est un algorithme de tri probabiliste (c'est à dire que son implémentation utilise le hasard) qui n'a aucun intérêt pratique puisque son exécution est très lente.

On rappelle qu'une permutation de $[\![0, n-1]\!]$ est une fonction bijective de l'ensemble $[\![0, n-1]\!]$ dans lui-même. Lors d'un tri du singe, on applique une permutation aléatoire sur les éléments de la liste et on vérifie si la liste ainsi obtenue est effectivement triée. Si c'est le cas, l'algorithme s'arrête et sinon on recommence avec une nouvelle permutation aléatoire jusqu'à ce que la liste soit triée.

On choisit de représenter une permutation σ de $[\![0, n-1]\!]$ par une liste `S` de taille `n` où pour tout `i` $\in [\![0, n-1]\!]$, l'élément `S[i]` vaut σ`(i)`. Ainsi, une liste `S` est une permutation lorsque `S` contient chaque élément de $[\![0, n-1]\!]$ une et une seule fois. Appliquer une permutation `S` à une liste `L` consiste à créer une nouvelle liste `M` telle que pour tout `i` $\in [\![0, n-1]\!]$, l'élément d'indice `i` de `M` est l'élément d'indice `S[i]` de `L`. Par exemple :

Si `L = [8, 1, 0, 10, 5, 6]` et `S = [2, 5, 0, 3, 1, 4]` alors `M = [0, 6, 8, 10, 1, 5]`.

Afin de créer une permutation aléatoire, on procède de la manière suivante :

- ⋆ On initialise tous les éléments de `S` avec des `None`.
- ⋆ Pour `i` variant de `0` à `n-1` :
  - On note $A \in \mathbb{N}^*$ le nombre de `None` dans `S` et on choisit un entier `a` aléatoirement dans $[\![1, A]\!]$.
  - On remplace le `a`$^{\text{ème}}$ `None` de `S` par `i`.

1. Écrire une fonction `tri_singe` qui trie une liste d'entiers à l'aide du tri du singe. Si besoin, on pourra utiliser les indications ci-dessous.
2. Est-ce un tri en place ? Un tri stable ?

Indications (essayez de résoudre l'exercice sans lire ce qui suit).

(a) Écrire une fonction « `est_triee(L: list[int]) -> bool` » qui renvoie `True` si `L` est triée et `False` sinon.

(b) Écrire une fonction « `a_eme_None(a: int, S: list[int ou NoneType]) -> int` » qui renvoie l'indice du `a`$^{\text{ème}}$ `None` dans `S`. On pourra supposer que `S` contient au moins `a` fois `None`.

(c) Écrire une fonction `perm_alea` qui prend en entrée un entier `n` et renvoie `S` une permutation aléatoire de $[\![0, n-1]\!]$. Bien sûr, vous devez utiliser la méthode décrite ci-dessus. Vous pouvez utiliser la fonction `randint` du module `random` qui prend en entrée deux entiers `b` et `c`, et qui renvoie un entier aléatoire choisi dans $[\![b, c]\!]$.

(d) Écrire une fonction `appliquer_perm` qui prend en entrée les deux listes `L` et `S`, et renvoie la liste `M` obtenue lorsqu'on applique la permutation `S` à `L`. Votre fonction ne doit pas modifier `L`.

Exercice 8. Tri par sélection en place

Le principe du tri par sélection a été expliqué dans l'exercice 1.

1. Écrire une fonction « `tri_selection_EP(L: list[int]) -> NoneType` » qui effectue un tri par sélection en place (en particulier, votre fonction doit modifier `L` et ne rien renvoyer). Si besoin, on pourra utiliser les indications ci-dessous. Pour tester la fonction on écrira :

```
L = [ ... ]; tri_selection_EP(L); print(L)
```

2. Est-ce un tri stable ? Justifier.

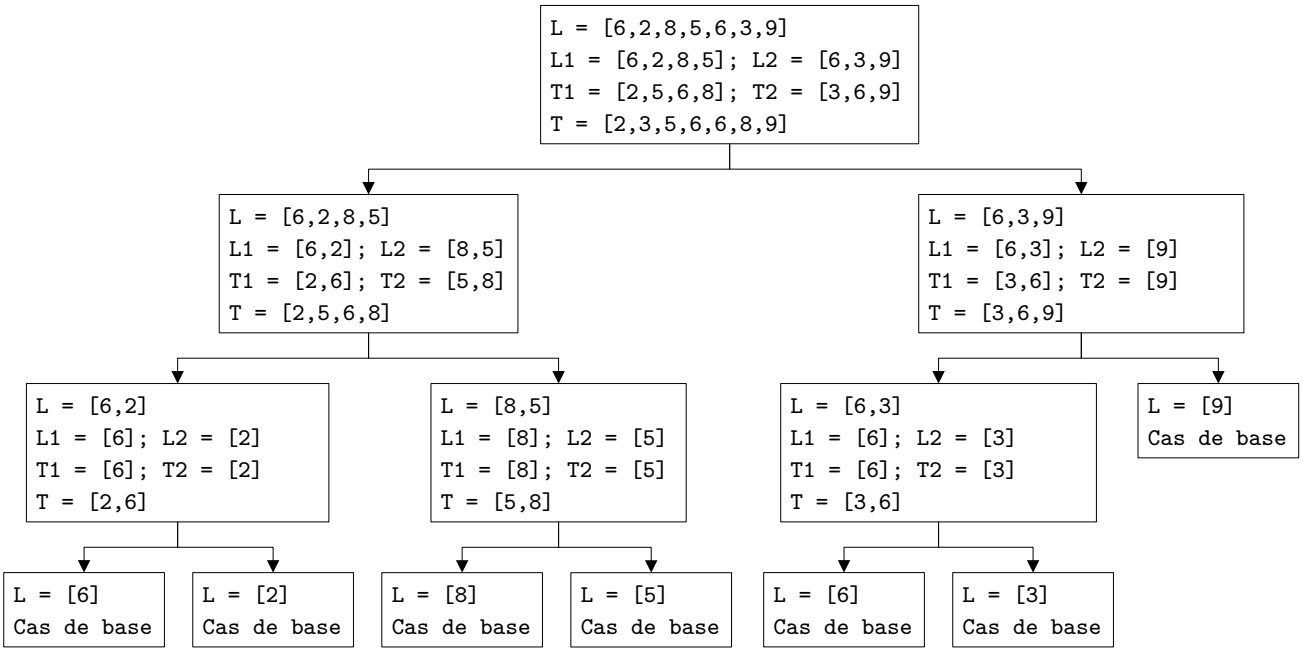
Indications (essayez de résoudre l'exercice sans lire ce qui suit). Pour `i` variant de `0` à `len(L)-1`, on repère le plus petit élément de `L[i:]`, puis on échange cet élément avec `L[i]`. Par exemple, voici l'évolution de la liste `L = [5, 8, 1, 6, 5, 7]` lors d'un tri par sélection en place :

$$\texttt{[5,8,1,6,5,7]} \to \texttt{[1,8,5,6,5,7]} \to \texttt{[1,5,8,6,5,7]}$$
$$\to \texttt{[1,5,5,6,8,7]} \to \texttt{[1,5,5,6,8,7]} \to \texttt{[1,5,5,6,7,8]}.$$

On pourra donc commencer par écrire une fonction « `mini_ter(L: list[int], i_min: int) -> int` » qui renvoie l'indice `i` $\in [\![$`i_min; len(L) - 1`$]\!]$ tel que `L[i]` est minimum. Si le minimum apparaît plusieurs fois, on renverra le plus petit indice possible.

Exercice 9. Tri fusion

Le tri fusion est un algorithme récursif. Pour l'appliquer à une liste `L`, on commence par la séparer en deux sous-listes `L1` et `L2` telles que `L1 + L2 = L` (si `L` est de taille paire alors `L1` et `L2` ont la même taille, sinon `L1` a un élément de plus que `L2`). On trie récursivement `L1` et `L2` pour obtenir deux nouvelles listes `T1` et `T2`. Finalement, à partir de `T1` et `T2`, on construit une nouvelle liste triée `T` contenant les mêmes éléments que `L` (cette opération s'appelle une ***fusion***).



1. Écrire une fonction `tri_fusion` qui trie une liste d'entiers à l'aide du tri fusion. Si besoin, on pourra utiliser les indications ci-dessous.
2. Est-ce un tri en place ? Un tri stable ? Expliquer comment modifier votre programme pour passer d'un tri stable à un tri non stable (ou inversement).

Remarque. Le tri fusion est en complexité $\mathcal{O}(n\log(n))$ où n est la taille de la liste à trier. En pratique, le tri rapide est souvent plus rapide que le tri fusion.

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On pourra écrire une fonction intermédiaire

```
fusion(T1: list[int], T2: list[int]) -> list[int]
```

qui prend en entrée deux listes triées et effectue la fusion de ces deux listes (c'est à dire qu'elle renvoie une liste triée contenant les mêmes éléments que `T1 + T2`). Votre fonction devra être de complexité linéaire en `len(T1) + len(T2)`.

Exercice 10. Sélection en temps linéaire (facultatif)

Le temps d'exécution d'un tri rapide dépend du pivot choisi à chaque étape. Dans l'idéal, il faudrait que le pivot soit la médiane des éléments de la liste à trier. On aimerait donc pouvoir trouver efficacement la médiane des éléments d'une liste. Le ***problème de la sélection*** (qui n'a rien à voir avec le tri par sélection) est une généralisation de ce problème.

Étant donnée une liste non vide `L` d'éléments deux à deux distincts ainsi qu'un entier `i` $\in [\![$`0; len(L) - 1`$]\!]$, le problème de la sélection consiste à déterminer l'élément `T[i]` où `T` est la liste triée qui contient les mêmes éléments que `L`. Par exemple, la médiane correspond à `i = (len(L)-1)//2`. Évidemment, on pourrait commencer par trier la liste `L` à l'aide de la fonction `sorted` de Python, mais le temps d'exécution serait alors en $\Omega(n\log(n))$ avec n la taille de `L`. Dans cet exercice, on va étudier un algorithme publié en 1973 qui résout le problème de la sélection en temps linéaire.

En voici le principe : si la liste `L` est de taille au plus 5, on trie les éléments de `L` et on renvoie l'élément d'indice `i` dans la liste triée, sinon :

⋆ On répartit les éléments de `L` en groupes de 5 éléments et on construit une liste `L_med` contenant les médianes de chacun de ces groupes. `L_med` est donc de taille `(len(L)-1)//5 + 1`. Par exemple, à partir de la liste

```
[15, -32, -19, -9, 3, -12, 34, -23, 30, -20, 19, 25, 5, -24, -18, -16, -8]
```

on créé quatre groupes :

```
[15, -32, -19, -9, 3]  [-12, 34, -23, 30, -20]  [19, 25, 5, -24, -18]  [-16, -8]
```

La liste `L_med` est donc égale à :

```
L_med = [-9, -12, 5, -16]
```

⋆ À l'aide d'un appel récursif, on calcule `med_med` la médiane des éléments de `L_med`. Dans l'exemple ci-dessus, on obtient `med_med = -12`.

⋆ On construit `L1` (resp. `L2`) la liste de tous les éléments de `L` strictement inférieurs (resp. strictement supérieurs) à `med_med`. Notez qu'on a supposé que les éléments de `L` sont distincts.

⋆ Il y a alors trois cas en fonction de la taille de `L1` :

- L'entier `med_med` est l'élément à sélectionner. Dans ce cas, on renvoie `med_med`.
- L'élément à sélectionner est dans la liste `L1`. Dans ce cas, on fait un appel récursif sur `L1`.
- L'élément à sélectionner est dans la liste `L2`. Dans ce cas, on fait un appel récursif sur `L2`.

1. Écrire une fonction « `selection(L: list[int], i: int) -> int` » qui résout le problème de la sélection.

Remarque. Le fait que cet algorithme soit en complexité $\mathcal{O}$`(len(L))` n'est pas du tout évident, mais c'est bel et bien le cas.