# Tris

C. Charignon

Trier un tableau est un exercice classique et très formateur, qu'on retrouve dans n'importe quel cours d'informatique. De nombreuses méthodes existent ; trois sont officiellement au programme, qui ont chacune un intérêt propre selon la situation.

## Table des matières

# Première partie
# Cours

## 1 Introduction

### 1.1 Pourquoi trier ?

La première raison pour laquelle on peut vouloir trier un ensemble de données est pour faciliter les recherches ultérieures. En effet, on pourra utiliser une recherche dichotomique de complexité logarithmique, au lieu de la naïve recherche linéaire.
*Remarque :* On peut trier un ensemble de données à condition qu'elles soient d'un type muni d'une relation d'ordre. D'où l'importance de la notion de relation d'ordre en informatique. À la réflexion, on se rend compte que tous les types courants peuvent être munis d'une relation d'ordre. En fait, Caml met une relation d'ordre par défaut sur tous les types qu'on crée. Et puis pensez qu'après tout une donnée est toujours enregistrée comme une suite de bits, il y a donc en théorie toujours la possibilité de considérer cette suite de bits comme un entier en base 2 et d'utiliser la relation d'ordre de $\mathbb{N}$.

### 1.2 Critères d'appréciation d'un tri

Bien sûr le principal critère pour juger l'efficacité d'un tri est sa complexité. On étudiera la complexité au pire des cas, bien que la complexité moyenne soit sans doute plus parlante (mais plus compliquée à calculer...). En général, on étudiera la complexité en nombre de comparaisons. On essaiera néanmoins de minimiser le nombre d'affectation autant que possible !

On se posera également la question de la complexité en espace.

Un point de vocabulaire : lorsqu'on tri un tableau (type `array` pour Caml, ou `list` pour Python) une méthode de tri sera dite "en place" lorsque toutes les opérations sont effectuées au sein du tableau de départ. Ainsi la complexité en espace sera minimale. En outre, une méthode en place sera a priori plus rapide car on évite de recopier de nombreuses fois le tableau.

En général, une telle méthode fonctionnera par effet de bord, elle est donc de type `list -> None` en Python, ou `'a array-> unit` en Caml.

Une fonction qui ne renvoie rien mais qui crée des effets de bords s'appelle une "procédure" dans certains langages. Ce vocabulaire est pratique, mais il y a un problème : rien n'interdit à une fonction d'avoir des effets de bords *et* de renvoyer une valeur...

*Remarque :* Pour trier une liste chaînée (le type `list` de Caml), c'est impossible puisqu'une liste est persistante ! Et de toute façon, le problème du temps de recopier une liste ne se pose pas en pratique... On rappelle que l'opération `let l2=l1;;` si `l1` est une liste est en $O(1)$. Alors que `let v2 = copy_vect v1; ;` est en $O(n)$ si $n$ est la longueur de `v1`.

Par exemple, pour calculer la médiane d'un tableau `T` :
Avec un tri par effet de bord :

```
def mediane1(T):
    triEnPlace(T) #triEnPlace ne renvoie rien
    return(T[len(T)//2])
```

Avec un tri qui renvoie un nouveau tableau trié (solution qu'on évite en général) :

```
def mediane2(T):
    S = triPasEnPlace(T) #triPasEnPlace renvoie un nouveau tableau
    return( S( len(T)//2))
```

La première version peut être source de piège : le tableau fourni pas l'utilisateur a été trié à son insu !
On pourrait préférer ceci :

```
import copy
def mediane3(T)
    """calcule la mediane en utilisant triEnPlace,
    mais sans modifier le tableau fourni en entrée."""
    S=copy.deepcopy(T)
```

```
    triEnPlace(S)
    return(S(len(S)//2))
```

Python, comme le type ”list” est mutable, il sera souvent plus pratique de fonctionner par effet de bord, et donc de créer des tris en place.

# 2 Tri par insertion

Le tri par insertion est le tri du joueur de carte. On va insérer l'un après l'autre chaque élément à sa place parmi les éléments déjà triés.

Ce tri sera moins facile à programmer sur des tableau que sur des listes chaînées.
Si vous pensez à la manière dont un tableau ou une liste est gérée en mémoire, c'est prévisible : pour insérer un élément dans une liste, il suffit de modifier deux pointeurs.
Tandis que pour insérer dans un tableau, il faudra décaler tous les éléments suivant l'élément inséré.

## 2.1 Avec des listes

Voici l'insertion en Caml avec des vraies listes :

```
let insere l x=
  (* l est une liste triée. Ceci renvoie la liste obtenue en rajoutant x dans l « à sa place
     ↪ ». *)
  match l with
  |[]   -> [x]
  |t::q when t<x  -> t :: insere q x
  |_              -> x :: l
;;
```

D'où la fonction de tri :

```
let tri = function
  |[] -> []
  |t::q -> insere t (tri q)
;;
```

## 2.2 Avec des tableaux, en place

Soit $t$ un tableau que nous souhaitons trier. Soit $n$ sa longueur. L'invariant de boucle principal sera : $\forall i \in [\![0, n]\!]$,

En entrée d'itération $i$, `t[:i]` est trié.

De sorte qu'à chaque étape, il s'agira de prendre `t[i]` pour l'insérer dans `t[:i]` en faisant en sorte que `t[:i+1]` devienne trié.

```
def cherchePlace(t,i):
    """ Précondition : t[0:i] est déjà trié.
     Ceci cherche la place où mettre t[i]"""
    j=i
    while j>0 and t[j-1] > t[i]:
        j-=1
    return j
```

```
def permutationCirculaire(t,i,j):
    """ Permutation circulaire "vers la droite": les éléments de t[i:j] avancent d'une case,
        ↪ t[i] arrive en t[j]"""
    tmp=t[i]
    for k in range(i,j,-1):
        t[k]=t[k-1]
    t[j]=tmp
```

```python
def tri(t):
    """ tri par insertion de t"""
    for i in range(1,len(t)):
        j=cherchePlace(t,i) # On cherche où doit aller t[i]
        permutationCirculaire(t,i,j) # On fait arriver t[i] au bon endroit
```

### 2.3 Complexité

Le pire des cas est celui d'un tableau trié à l'envers. Dans ce cas, chaque élément considéré devra être ramené au début du tableau. Si $n$ est la longueur du tableau, pour tout $i \in [\![1, n-1]\!]$, il y aura $i$ comparaisons. Ce qui nous fait un total de

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O_{n\to\infty}(n^2).$$

*Remarque :* On peut aussi facilement compter le nombre d'opérations sur le tableau : pour déplacer un élément de la place $i$ à la place 0, il faut $i+1$ lectures dans le tableau et autant d'écritures ($i+2$ opérations de lectures écriture si on compte la variable temporaire utilisée), donc un total de $\sum_{i=1}^{n}(i+1) = \frac{n(2+n+1)}{2} = O(n^2)$ lectures-écritures.

L'intérêt du tri par insertion est lorsque la tableau est presque trié. Imaginons par exemple qu'il existe une constante $K$ telle que chaque élément est dans le tableau ou la liste initial(e) au plus à $K$ cases de sa place finale. Alors le nombre de comparaisons pour insérer chaque élément au maximum $K$, d'où une complexité au maximum de $nK$ comparaisons, donc $O(n)$.

## 3 Partition - fusion

Vu en option info en MPSI, revu au premier chapitre. Version en place : **cf exercice :** 6.
Version pour un tableau mais pas en place :

```python
def fusion(t1, t2):
    """ Les tableaux t1 et t2 doivent être triés. Cette fonction renvoie le tableau trié
        ↪ obtenu en les fusionnant."""
    res=[]
    i,j=0,0 # Ces variables retiendront le prochain élément à lire dans t1 et dans t2
    n1, n2= len(t1), len(t2)

    while i<n1 and j<n2:
        if t1[i] < t2[j]: # le prochain élément à prendre est dans t1
            res.Append(t1[i])
            i+=1
        else:
            res.Append(t2[j])
            j+=1

    # Maintenant, un des deux tableaux au moins a été transféré dans res.
    # Rajoutons-y le restant du dernier tableau
    res.extend(t1[i:])
    res.extend(t2[j:])
    # rema : une des deux instructions ci-dessus est inutile.

    return res



def triFusion(t):
    if len(t)<2:
        return t
    else:
        n=len(t)
        t1, t2 = t[:n//2], t[n//2:]
        return fusion ( triFusion(t1), triFusion(t2) )
```

La complexité temporelle est la même que pour la version sur listes chaînées vue en MPSI, à savoir $O(n \log n)$. Cependant, les nombreuses copies de portions de tableau ralentissent un peu, et nécessite de la mémoire supplémentaire, ce qui fait que ce tri est plutôt moins efficace sur tableaux.

### 3.1 Complexité spatiale

Prenons ici pour simplifier un tableau de longueur une puissance de 2 : soit $k \in \mathbb{N}$ tel que $|t| = 2^k$. L'occupation mémoire est maximale lorsque nous sommes dans une feuille de l'arbre des appels. Elle est alors de

$$\sum_{i=0}^{k} 2^i$$

Ce qui vaut $2^{k+1} - 1$. Donc en gros, trier un tableau de taille $n$ nécessite un espace mémoire supplémentaire de $2n$.

## 4 Tri rapide, ou de Hoare

Le principe de ce tri est proche du tri fusion : on partage le tableau en deux, on tri séparément chaque moitié, puis on rassemble les deux parties. La différence est qu'on ne partage pas le tableau en deux moitiés de même longueur : on choisit un élément appelé pivot (le plus simple est de choisir le premier élément du tableau) on sépare le tableau en deux parties : les éléments inférieurs au pivot, et les élément supérieurs au pivot.

Pour rassembler les deux morceaux une fois chacun d'eux triés une simple concaténation suffit, puisque les éléments du premier morceaux sont tous inférieurs à ceux du deuxième morceau.

**N.B.** Le pivot doit être retiré de la liste à trier, sans quoi on risque de faire planter le programme dans le cas où le pivot serait le maximum ou le minimum de la liste.

### 4.1 Sur liste chaînée (Caml)

Très simple : on part du tri fusion et on modifie la fonction de partition.

### 4.2 Complexité au pire

Comptons les comparaisons au pire dans le tri rapide.
Soit $T$ un tableau et $n$ sa longueur.

- L'opération de partition nécessite $n$ comparaisons.
- Étant donnés $T1$ et $T2$ deux sous-tableaux, l'opération de fusion de $T_1$ et $T_2$ nécessite au pire $\max(\texttt{len(T1)}, \texttt{len(T2)})$ opérations.

Le pire des cas est obtenu lorsque l'opération de partition conduit à un sous-tableau de longueur 1, et un autre de longueur $n-1$. C'est le cas par exemple lorsque le tableau est déjà trié, ou déjà trié à l'envers.

Plaçons-nous dans ce cas, et notons alors pour tout $k \in \mathbb{N}$ $C_k$ la complexité pour trier $k$ éléments. La relation de récurrence obtenue est :

$$\forall k \in \mathbb{N}, C_k \leqslant 2k + C_{k-1}$$

Donc au pire des cas, $C_n = \sum_{k=1}^{n} 2k = n(n-1) = O_{n\to\infty}(n^2)$.

*Remarque :* Pour remédier à ce problème, une stratégie est de choisir comme pivot non pas le premier élément venu mais la médiane du tableau. Mais ceci nécessite de savoir calculer une médiane rapidement... **cf exercice :** 14, 18

Le tri par segmentation sur listes n'a donc a priori guère d'intérêt comparé au tri fusion.

### 4.3 Sur tableau, en place

L'intérêt essentiel du tri rapide est le fait qu'il peut être implémenté pour des tableaux en place de manière efficace. C'est pourquoi il est fréquemment utilisé.

Le point délicat est la segmentation : une fois choisi un pivot, nous devons placer à gauche les éléments inférieurs au pivot, et à droite ceux supérieurs au pivot, le pivot lui-même arrivant entre les deux.

A ce moment le pivot sera à sa place, il n'y aura plus qu'à trier récursivement la partie gauche et la partie droite. Pas d'opération de fusion à faire, puisque tout est déjà en place.

*Remarque :* Nous ne savons pas à priori combien il y aura d'éléments à gauche et à droite du pivot, nous ne pouvons donc pas savoir la place finale du pivot.

Enfin le but est de réaliser la partition en $O(n)$ comparaisons.

## 5 En résumé, utilité de chaque tri

Pour schématiser :

- Le tri fusion est pratique lorsqu'on dispose de listes.
- Le tri rapide (de Hoare) est pratique lorsqu'on utiliser des tableaux. Défaut : la complexité au pire est en $O(n^2)$. Cependant, la complexité moyenne est bien optimale , c'est-à-dire en $O(n\log(n))$.
- Le tri par insertion a a priori une mauvaise complexité, celle de la plupart des tris naïfs. Cependant, il sera utile lorsque le tableau est presque trié (justement quand le tri rapide est mauvais). C'est une situation qui peut arriver assez vite : si on maintient un tableau trié et qu'on le modifie un peu de temps en temps (typiquement : base de donnée). **cf exercice :** 11

D'autres méthodes de tris sont présentes dans la feuille de TD :

- le tri bulle
  - ◇ « If you know what bubble sort is, wipe it from your mind ; if you don't know, make a point of never finding out ! » (Numerical Recipes in C : The Art of Scientific Computing)
  - ◇ « bubble sort seems to have nothing to recommend it » (Knuth, TAOCP, vol 3)
  - ◇ https://www.youtube.com/watch?v=k4RRi_ntQc8
- le tri par extraction **cf exercice :** 8

# Deuxième partie
# Exercices

# Exercices : tris

*Ce chapitre est au programme du tronc commun : on rédigera donc a priori les programmes en Python.*

## 1 Divers, révisions sur les tableaux

**Exercice 1. * Mélange de Knuth**

Voici une manière de mélanger un tableau en place de manière aléatoire, appelée « mélange de Knuth ». Soit $t$ un tableau et $n$ sa longueur. Pour tout $i \in [\![0, n[\![$, on tire au hasard uniforme une position $j \in [\![0, n[\![$ et on échange les cases $i$ et $j$ de $t$.

1. Programmer cet algorithme et donner sa complexité.
2. (***) Démontrer que toutes les permutations sont équiprobables.

**Exercice 2. ** Copie profonde**

1. ***(Vu en MPSI)*** Écrire une fonction `nouvelleMatrice` qui prend en entrée deux entiers $n$ et $p$ et qui renvoie une matrice de format $n \times p$.
2. Écrire une fonction `copie` qui effectue une copie d'un tableau, quel que soit son niveau de profondeur. C'est donc la fonction `deepcopy` de la bibliothèque `copy`.

**Exercice 3. ** Mélanger une liste**

On propose la méthode suivante pour mélanger une liste Caml :

- Associer à chaque élément un flottant aléatoire ;
- Trier la liste selon ces flottants ;
- Supprimer ces flottants.

## 2 Applications

**Exercice 4. **! Distance minimale entre deux éléments d'une liste**

On désire écrire une fonction qui prend en entrée une liste d'entiers et qui renvoie la distance minimale entre deux éléments de cette liste.

1. Quelle serait la complexité de la méthode naïve ?
2. Écrire une fonction qui commence par trier la liste, et donner sa complexité.

**Exercice 5. *** Élément dans un maximum d'intervalles**

Un intervalle sera représenté par un couple `(debut, fin)`. Le but est d'écrire une fonction prenant en entrée une liste d'intervalles et de déterminer un élément présent dans un maximum d'intervalles.

Une fois appliqué un tri, on pourra répondre à la question en $O(n)$, si $n$ est le nombre d'intervalles. Si vous ne trouvez pas, une deuxième indication est présente à la fin de la feuille de TD.

## 3 Tri fusion

**Exercice 6. ** Tri fusion, version procédure**

1. Écrire une procédure qui trie un tableau selon l'algorithme du tri fusion. Comme pour le tri par segmentation, on utilisera une fonction auxiliaire `tri` telle que pour tout tableau $t$ et tout couple $(i, j) \in [\![0, len(t)]\!]^2$, `tri(t,i,j)` trie la portion `t[i:j]` du tableau `t`.

   Pour la partie fusion, on pourra utiliser des permutations circulaires, comme dans le tri par insertion. Ou bien effectuer la fusion dans un autre tableau, et recopier celui-ci dans le tableau initial à la fin.
2. Une petite optimisation : on peut facilement tester si par hasard à l'issue de la partition et du tri des deux morceaux, les éléments de gauche sont déjà tous plus petits que les éléments de droite, et dans ce cas éviter la fusion.

   Apporter cette amélioration. Dans quels cas sera-ce utile ?

**Exercice 7. **! Nombre d'inversions**

Soit $t$ un tableau, $n$ sa longueur, $a_0, ..., a_{n-1}$ ses éléments dans l'ordre. On rappelle qu'une inversion de $t$ est un couple $(i, j) \in [\![0, n[\![^2$ tel que $i < j$ et $a_i > a_j$. Le but de l'exercice est de calculer le nombre d'inversions dans une liste.
*Remarque :* Le nombre d'inversions peut donner une mesure du « désordre » du tableau. On rappelle aussi que la signature de $t$ vaut $(-1)^{\text{nombre d'inversions de l}}$.

1. *Méthode naïve :* On programmera cette fonction en Caml sur des listes chaînées. Écrire une fonction `plusPetitQue` qui prend une liste $l$ et un élément $x$ et qui renvoie le nombre d'éléments dans $l$ qui sont inférieurs à $x$. En déduire une fonction calculant le nombre d'inversions dans une liste.

   Calculer la complexité de cette fonction.
2. *Méthode diviser pour régner :* On programmera cette fonction sur des tableaux Python.

   On peut calculer le nombre d'inversions de manière plus efficace en reprenant la structure du tri fusion (pas en place).

   Le principe est le suivant : soit $t1$ la première moitié de $t$ et $t2$ la seconde moitié. Lors de la fusion de $t1$ avec $t2$, à chaque fois qu'on insère un élément venant de $t2$, il était en inversion avec tous les éléments pas encore insérés de $t1$.

   Programmer cette méthode, puis donner l'ordre de grandeur du nombre de comparaisons effectuées.

# 4 Autres tris

Voici deux tris classiques, parmi les premiers auxquels on pense en général lorsqu'on n'a pas suivi de cours d'informatique. Ils n'ont à ma connaissance pas d'avantage, si ce n'est culturel, face aux tris présentés en cours.

**Exercice 8. ** Tri par extraction**

Le principe du tri par extraction est le suivant : on recherche le maximum du tableau, et on le place à la bonne place. Puis on recherche le maximum dans le restant du tableau, etc...

Programmer cette méthode de tri, en place. Quelle est sa complexité ?

**Exercice 9. ** Tri bulle**

https://www.youtube.com/watch?v=koMpGeZpu4Q

Pour trier un tableau de $n$ éléments, on effectuera $n$ parcours de ce tableau. A chaque passage, si on rencontre un indice $i$ tel que $t[i] > t[i+1]$ on permute $t[i]$ et $t[i+1]$.

On se convainc aisément qu'après le premier passage, le maximum est arrivé à sa place, après le deuxième passage le deuxième élément aussi, etc...

Enfin, cette méthode s'adapte aux listes chaînées.

1. Programmer le tri bulle sur listes ou sur tableau.
2. Quelle est sa complexité au pire ? Au mieux ?

# 5 Tri par insertion

**Exercice 10. *! Complexité du tri par insertion en nombre d'opérations sur les tableaux**

Compter le nombre maximal d'écritures dans le tableau pour le tri par insertion.

**Exercice 11. ** Complexité du tri par insertion sur un tableau presque trié**

Pour tout $k \in \mathbb{N}$ on dit qu'un tableau $t$ est $k$-presque trié lorsqu'il contient au plus $k$ éléments qui sont à une distance de plus de $k$ cases de leur place finale dans le tableau trié.

Calculer la complexité du tri par insertion d'un tableau $k$-presque trié.

**Exercice 12. **! Insertion dichotomique**

On peut optimiser le tri par insertion en recherchant l'emplacement où insérer l'élément par dichotomie.

1. Programmer cette amélioration. On demande un tri en place.
2. Calculer la complexité en nombre de comparaisons.
3. Le tri par insertion dichotomique est assez peu utilisé en pratique. Pourquoi ?

# 6 Tri rapide ou de Hoare ou par segmentation

**Exercice 13. ** Insertion lorsqu'il y a peu d'éléments**

Écrire une fonction Python pour comparer le temps d'exécution du tri par insertion et du tri rapide pour des tableaux de petite taille. Estimer à partir de quelle taille de tableau le tri rapide est meilleur que le tri par insertion.

En déduire une optimisation du tri rapide.

**Exercice 14. **! Calcul de médiane**

Pour simplifier, on conviendra que la médiane d'un tableau $T$ de longueur $n$ est l'élément $T[\lfloor \frac{n}{2} \rfloor]$. (Donc en python `T[n//2]`.) Ce qui signifie que dans le cas où $n$ est pair, on choisit l'élément « du dessus » (l'autre choix possible serait $\frac{T[n//2-1]+T[n//2]}{2}$).

Une méthode plus efficace que de trier complètement le tableau consiste à adapter le tri rapide. On garde la partie « segmentation », mais ensuite une fois qu'on dispose des deux sous-tableaux, on peut facilement déterminer dans lequel de ces sous-tableaux il faudra rechercher la médiane, ce qui permettra de ré-appeler récursivement la fonction. Cependant l'élément recherché ne sera plus la médiane du sous-tableau. C'est pourquoi il vaut mieux écrire une fonction plus générale qui prend un entier $i$ et qui renvoie le $i$-ème plus petit élément du tableau.

1. Écrire une fonction récursive `iemeElement` prenant en argument un tableau $T$, deux entiers $deb$ et $fin$, un indice $i \in [\![0, fin - deb[\![$ et qui renvoie le $i+1$-ème plus petit élément de $T$, en supposant qu'il est dans `T[deb:fin]`.
2. En déduire une fonction renvoyant la médiane d'un tableau.
3. Quelle est la complexité au pire de cette méthode ?
4. Comment utiliser `iemeElement` pour calculer le minimum ou le maximum d'un tableau ? Est-ce judicieux ?

**Exercice 15. **! Dérécursifier le tri rapide**

Écrire une version non récursive du tri rapide. On utilisera une pile pour contenir les prochaines opérations à faire.

**Exercice 16. *** Tri par segmentation et tri par ABR**

On utilise dans cet exercice des listes chaînées de Caml, qu'on supposera de plus pour simplifier sans doublon. Pour toute liste $l$ et tout élément $x$, on notera $l_{<x}$, respectivement $l_{>x}$ la liste des éléments plus petits, respectivement plus grands que $x$, dans le même ordre que dans $l$. Ainsi, $l_{<x}, l_{>x}$ est le résultat de `segmente x l` (à condition que $x \notin l$).

1. Récupérer la fonction `abr_of_list` vue en MPSI. On rappelle qu'on peut en déduire facilement une fonction de tri, que nous appellerons dans cet exercice le « tri par ABR ».
2. On définit à présent une autre fonction, qu'on appelle `f` qui transforme une liste en un ABR en suivant l'algorithme du tri par segmentation : $f : \left| \begin{array}{l} [\,] \mapsto \emptyset \\ q@[x] \mapsto Noeud(fq_{<x}, x, fq_{>x}) \end{array} \right.$

   Démontrer que `f = abr_of_list`.

   *Remarque :* Si la fonction `abr_of_list` avait été programmée à l'aide d'un accumulateur, les insertions se feraient dans l'autre sens, et il faudrait remplacer $q@[x]$ ci-dessus par $x :: q$.

3. Démontrer que pour toute liste `l`, à condition de choisir comme pivot le dernier élément de la liste, les comparaisons effectuées dans `abr_of_list` sont exactement celles effectuées dans le tri par segmentation de `l`, même si elles ne le sont pas dans le même ordre.

   ⚷ Ainsi, le tri par ABR a exactement la même complexité en nombre de comparaisons que le tri par segmentation.

**Exercice 17. *** Complexité moyenne du tri rapide**

On notera, pour tout $n \in \mathbb{N}$, $C_n$ la complexité moyenne du tri rapide sur un tableau à $n$ éléments, en nombre de comparaisons entre éléments du tableau. Le but de l'exercice est de déterminer un équivalent de $C_n$.

1. Préciser $C_0$ et $C_1$.
2. Soit $n \in [\![2, \infty[\![$. Quelle est la complexité de la segmentation pour un tableau (ou un sous-tableau en pratique) à $n$ éléments ?
3. Soit $n \in [\![2, \infty[\![$. On suppose que lors de la segmentation, toutes les places finales possibles du pivot sont équiprobables.
   (a) Justifier que :
   $$C_n = \frac{1}{n} \sum_{i=0}^{n-1} (C_i + C_{n-1-i}) + n - 1.$$

(b) Puis que :

$$C_n = \frac{2}{n}\sum_{i=0}^{n-1} C_i + n - 1.$$

4. Déduire que pour tout $n \in [\![2, \infty[\![$ :

$$C_n = C_{n-1} \times \frac{n+1}{n} + 2\frac{(n-1)}{n}.$$

5. On étudie alors la suite $\left(\frac{C_n}{n+1}\right)_{n\in\mathbb{N}}$. Démontrer que $\forall n \in \mathbb{N}^*$,

$$\frac{C_n}{n+1} = \sum_{k=2}^{n} 2\frac{k-1}{k(k+1)}$$

6. Finalement, obtenir un équivalent de $\left(C_n\right)_{n\in\mathbb{N}}$.

**Exercice 18. **** Optimisation du tri rapide par la médiane des médianes**

Voici une méthode pour déterminer un pivot afin d'optimiser le tri rapide. Il nécessite un calcul de médiane, nous allons donc écrire simultanément une fonction de tri et une fonction de calcul de médiane.

L'idée est de calculer dans un premier temps une médiane approchée.

On divise le tableau en sous-tableaux de longueur au plus 5, et on calcule la médiane de chacun de ces sous-tableaux. Puis on calcule récursivement la (vraie) médiane de ce tableau des médianes. C'est ce nombre qu'on utilisera comme pivot.

*Remarque :* On procédera autant que possible en place : quand on dit « diviser le tableau » c'est purement abstrait. En réalité on utilisera une fonction `medianeEntre(T,deb,fin)` appelé avec les paramètres `deb, fin` tels que `fin - deb`$\leqslant 5$.

1. Écrire une fonction `medianeDesMedianesEntre` prenant en arguments le tableau `T` et les indices `deb` et `fin` et renvoyant la médiane des médianes de `T[deb:fin]`. Pour la médiane de chaque sous-tableau de 5 éléments, on utilisera un algorithme de médiane non optimisé (en pratique, on choisit généralement celle basée sur le tri par insertion, plus rapide sur des petits tableaux).

   Pour la médiane des médianes, dans la version finale du programme on utilisera (récursivement) la fonction de médiane optimisée qu'on va écrire. En attendant, pour faire des tests, on pourra utiliser une version quelconque.
2. Quelle est la complexité du tri rapide classique dans le cas extrême où tous les éléments de $T$ sont identiques ?

   *Dans la suite, on suppose à l'opposé que les éléments de $T$ sont deux à deux distincts.*
3. Combien d'éléments au moins seront inférieurs au pivot renvoyé ? Et combien au moins seront supérieurs ?

   On supposera pour simplifier que la longueur de $T$ est multiple de 10.
4. Écrire une fonction médiane récursive basée sur la méthode de l'exercice précédent mais utilisant comme pivot la médiane approchée calculée précédemment.
5. Notons pour tout $n \in \mathbb{N}^*$, $C(n)$ le nombre maximal de comparaisons pour calculer la médiane d'un tableau de taille au plus $n$. Écrire la relation de récurrence vérifiée par la complexité de cette fonction. Montrer qu'il existe une constante $c$ telle que pour tout $n \in \mathbb{N}$ :

$$C(n) \leqslant C\left(\frac{n}{5}\right) + C\left(\frac{7}{10}n\right) + cn.$$

   On supposera encore pour simplifier $n$ multiple de 10.
6. En déduire que $\forall k \in \mathbb{N}$, $C(10^k) \leqslant 10c10^k$.
7. En déduire que la complexité du calcul de médiane ainsi programmé est linéaire.
8. Finalement, écrire le tri rapide ainsi optimisé. Écrire la relation de récurrence vérifiée par la complexité au pire de votre fonction. Faire quelques tests pour comparer aux tris déjà vu en cours.
9. Commentaires sur la complexité mémoire ?

*Remarque :* Ceci est la méthode de tri a priori la plus efficace parmi celles que nous avons vues. Cependant, on s'en tient souvent au tri rapide simple en choisissant un pivot au hasard car pour la plupart des tableaux c'est tout aussi efficace et évite les calculs supplémentaires de médiane approchée. Une méthode plus sophistiquée consiste à essayer un tri rapide simple, puis à passer à l'utilisation de la médiane des médianes si on se rend compte qu'il est inefficace. Enfin, il y a la possibilité de mélanger le tableau avant de le trier (en utilisant l'exercice 1 ) pour éviter avec une forte probabilité les mauvais cas.

# Quelques indications

**3** Utiliser `List.map` pour ajouter puis supprimer le flottant.

**5** On propose de créer une liste contenant toutes les bornes d'intervalles triées par ordre croissant, auxquelles on adjoindra un booléen qui indique s'il s'agit du début ou de la fin d'un intervalle.

**6** 1.

2. Cette optimisation est notamment utile lorsque la portion de tableau qu'on est en train de trier est déjà triée.

**12** 3) Il n'y a pas que les comparaisons dans la vie...

**15** Par exemple, on pourra mettre dans la pile le couple $(a, b)$ pour signifier que la prochaine opération à faire est de trier $t[a : b]$.

**16** 2) récurrence forte
3) aussi

**17** 6. Utiliser le théorème sur les série de terme général équivalent.

Quelques solutions

**1**

**2**

**4**

**5**

**6**

**7**

```python
# Méthode naïve
def nb_inversions(sigma):
    """ Entrée : sigma un tableau qui représente un permutation.
        Sortie : nb d'inversions de sigma."""
    res=0
    n=len(sigma)
    for i in range(0,n):
        #Cherchons les j tels que (i,j) est une inversion
        for j in range(i+1, n):
            if sigma[i] > sigma[j]:
                res+=1
    return res
```

**8**

**9**

**10** On trouve $\frac{n(n+1)}{2}$, soit $O_{n\to\infty}(n^2)$.

**11** La complexité de tri par insertion d'un tableau $k$-presque trié est $O(kn)$ (et donc $O_{n\to\infty}(n)$ si on fixe $k$).

**12** 1.

```python
def place_dicho(t,i):
    def aux(x,t,deb,fin):
        """
            Entrée: un élément x, un tableau t
            Précondition : la place où insérer x est dans [deb, fin]
            Sortie : la place a laquelle on doit insérer x
        """

        if deb == fin:
            return deb
        else :
            m=(deb+fin)//2
            if t[m]==x:
                return m
            elif t[m]< x :
                return  aux(x,t,m+1,fin)
            else:
                return aux(x,t,deb,m)

    return aux(t[i],t,0,i)

def insere_dicho(i,t):
    """
        Précondition : t[0:i] est trié
        Sortie : rien
        Effet de bord : déplace t[i] pour que t[0:i+1] devienne trié
    """

    x = t[i]
    place=place_dicho(t,i)
```

```
31
32      for j in range(i,place,-1):
33          t[j]=t[j-1]
34      t[place]=x
35
36
37  def tri_Insertion_Dicho(t):
38
39      for i in range(len(t)):
40
41          insere_dicho(i,t)
```

2. Soit $t$ un tableau et $n$ sa longueur. Pour tout $i \in [\![0, n[\![$, la recherche de la place finale de $t[i]$ se fait en $O(\log i)$ comparaisons, puis son déplacement ne coûte aucune comparaison. D'où une complexité en $\sum_{i=1}^{n} O\left(\log(i)\right)$, soit $O(n \log n)$.
3. Le nombre de comparaisons est en $O(n \log n)$ certes, mais le nombre d'écritures dans le tableau est en $O(n^2)$ au pire car l'insertion se fait exactement de la même manière que dans le tri par insertion classique. Donc pour un tableau quelconque, le tri par insertion dichotomique est moins efficace que le tri fusion.
   Et pour un tableau presque trié, la complexité reste de l'ordre de $n \log n$ comparaisons car la dichotomie n'est pas plus rapide si la place recherchée est à l'extrémité du tableau. Donc dans ce cas, le tri par insertion dichotomique est moins bon que le tri par insertion classique.

**13**

**14** 3) Le pire des cas n'a pas changé : celui d'un tableau trié ou trié à l'envers. Et dans ce cas, la complexité est toujours en $O(n^2)$.

**15**

**16** 1.

2. Pour tout $n \in \mathbb{N}$, soit $P(n)$ : «Pour toute liste $l$ de longueur $n$, $f(l) = \texttt{abr\_of\_list}(f)$».
   - $P(0)$ et vrai car $f([]) = \emptyset = \texttt{abr\_of\_list}([])$.
   - $P(1)$ est vrai aussi : l'image d'un singleton est la feuille correspondante pour les deux fonctions.
   - Soit $n \in \mathbb{N}^*$ tel que $\forall i \in [\![0, n]\!]$, $P(i)$. Prouvons $P(i+1)$. Soit $l$ de longueur $n+1$. Donc $|l| \geqslant 2$. Soit $t$ son premier élément, $x$ son dernier et $ll$ le reste de $l$. Donc `l = t: :ll@[x]`.
     Par $P(n)$, $\texttt{abr\_of\_list}(ll@[x]) = f(ll@[x]) = Noeud(fll_{<x}, x, fll_{>x})$. Puis par $P\left(|ll_{<x}|\right)$ et $P\left(|ll_{>x}|\right)$, $\texttt{abr\_of\_list}(ll@[x]) = Noeud(\texttt{abr\_of\_list}ll_{<x}, x, \texttt{abr\_of\_list}ll_{>x})$.
     Traitons le cas où $x < t$, l'autre étant similaire. L'expression $\texttt{abr\_of\_list}t :: ll@[x]$ est évaluée en `insere t ↪ (abr_of_list (ll@[x]))`. Nous allons donc insérer $t$ dans l'arbre $Noeud(\texttt{abr\_of\_list}ll_{<x}, x, \texttt{abr\_of\_list}ll_{>x})$. Comme $t > x$, $t$ sera inséré à droite, dans $\texttt{abr\_of\_list}ll_{>x}$. Par définition de la fonction `abr_of_list`, ceci renvoie $\texttt{abr\_of\_list}(t :: ll_{>x})$ autrement dit $\texttt{abr\_of\_list}(t :: ll)_{>x}$. Et par ailleurs $ll_{<x} = (t :: ll)_{<x}$.
     On peut terminer le calcul :

$$\begin{aligned} \texttt{abr\_of\_list}(l) &= Noeud(\texttt{abr\_of\_list}(t :: ll)_{<x}, x, \texttt{abr\_of\_list}(t :: ll)_{>x}) \\ &= Noeud(f(t :: ll)_{<x}, x, f(t :: ll)_{>x}) \\ &= f(l) \end{aligned} \quad \Big) HR$$

3. On procède encore par récurrence sur la longueur de la liste. Pour tout $n \in \mathbb{N}$, soit $P(n)$ : « Pour toute liste $l$ de longueur $n$, le tri par segmentation et le tri par ABR effectuent les mêmes comparaisons ».
   - Pour une liste vide ou un singleton, pas de comparaison dans les deux cas.
   - Soit $n$ tel que pour tout $i \in [\![0, n]\!]$, $P(i)$. Soit $l$ une liste de longueur $n+1$, $x$ son dernier élément, et $d$ la liste de ses $n$ premiers éléments.
     Les comparaisons effectuées par le tri par segmentation (en choisssant $x$ comme pivot) sont :
     (a) Les comparaison entre $x$ et tous les autres (pour la segmentation)
     (b) Les comparaisons des appels récursifs, pour trier $d_{<x}$ et $d_{>x}$.
     Par ailleurs, comme $\texttt{abr\_of\_list} = f$, nous savons que $\texttt{abr\_of\_list}l = Noeud(\texttt{abr\_of\_list}d_{<x}, x, \texttt{abr\_of\_list}d_{>x})$.
     Les comparaisons pour obtenir cet arbre sont :
     (a) Les comparaison entre $x$ et tous les éléments de $d$ pour savoir si on les envoie dans le fils gauche ou le fils droit.
     (b) Les comparaisons dans les appels récursifs $\texttt{abr\_of\_list}d_{<x}$ et $\texttt{abr\_of\_list}d_{>x}$.
     On a bien les mêmes comparaisons dans les deux algos (grâce à l'hypothèse de récurrence pour le deuxième point).

**17** 1. $C_0 = C_1 = 0$ : ce sont les cas d'arrêt.

2. Lorsqu'on appelle `segmente` sur une plage de $n$ éléments d'un tableau, le pivot est comparé à tous les autres éléments, ce qui fait $n-1$ comparaisons entre éléments du tableau.

3. (a) Soit $T$ un tableau, $l$ sa longueur, soit `deb` $\in [\![0, n-1]\!]$, voyons ce qui se passe lorsqu'on effectue `triEntre(T,deb,deb+n)`.
D'abord, on appelle `segmente(T,deb,deb+n)`, qui a une complexité de $n-1$. Notons $i$ le nombre de cases après `deb` où arrive le pivot à l'issue de la segmentation. Donc $i \in [\![0, n-1]\!]$. (Le résultat renvoyé par`segmente(T,deb,deb+n)` est donc `deb+i` ).
On appelle ensuite `triEntre(T,deb,deb+i)` qui a pour complexité moyenne $C_i$, et `triEntre(T,deb+i+1,deb+n)` qui a pour complexité moyenne $C_{n-i-1}$.
Au total, la complexité moyenne de `triEntre(T,deb,deb+n)` lorsque le pivot arrive en `deb+i` vaut $C_i + C_{n-i-1} + n - 1$.

Maintenant, nous n'avons plus qu'à faire la moyenne de ces valeurs pour toutes les valeurs de $i$ possible. Comme ces valeurs sont équiprobables par hypothèse, la probabilité de chacune est $\frac{1}{n}$. Et nous obtenons :

$$C_n = \sum_{i=0}^{n} \frac{1}{n} \left(C_i + C_{n-i-1} + n - 1\right) = \frac{1}{n} \sum_{i=0}^{n} \left(C_i + C_{n-i-1}\right) + n - 1.$$

*Remarque :* Il s'agit dans le fond d'une sorte de formule des probabilités totales pour l'espérance.

(b) Changement d'indice $i \mapsto n-1-i$ dans la deuxième partie de la somme.

4. Soit $n \in [\![2, \infty[\![$. On commence par mettre de côté $C_{n-1}$ dans la somme obtenue question précédente :

$$C_n = \frac{2}{n} \left(C_{n-1} + \sum_{i=0}^{n-2} C_i\right) + n - 1.$$

Mais en appliquant la formule de la question précédente au rang $n-1$, il vient $C_{n-1} = \frac{2}{n-1} \sum_{i=0}^{n-2} C_i + n - 2$, d'où $\sum_{i=0}^{n-2} C_i = \frac{n-1}{2} \left(C_{n-1} - (n-2)\right)$. On remplace alors :

$$\begin{aligned} C_n =& \frac{2}{n} \left(C_{n-1} + \frac{n-1}{2} \left(C_{n-1} - (n-2)\right)\right) + n - 1 \\ =& C_{n-1} \times \frac{n+1}{n} + \frac{-(n-1)(n-2) + n(n-1)}{n} \\ =& C_{n-1} \times \frac{n+1}{n} + 2\frac{(n-1)}{n}. \end{aligned}$$

5. Pour tout $n \in [\![2, \infty[\![$,

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + 2\frac{n-1}{n(n+1)}$$

d'où le résultat, sachant que $C_1 = 0$.

6. On a $2\frac{k-1}{k(k+1)} \underset{k\to\infty}{\sim} \frac{2}{k}$. Or la série de terme général $\left(\frac{2}{k}\right)_{k\in\mathbb{N}^*}$ est *à termes positifs* et *divergente* donc le théorème sur l'équivalent d'une série s'applique, et :

$$\frac{C_n}{n+1} \underset{x\to\infty}{\sim} \sum_{k=1}^{n} \frac{2}{k}.$$

On sait que la série harmonique $\sum_k \frac{1}{k}$ est équivalente à $\ln(n)$,d'où :

$$\frac{C_n}{n+1} \underset{n\to\infty}{\sim} 2\ln(n)$$

$$\text{donc } C_n \underset{n\to\infty}{\sim} 2(n+1)\ln(n)$$

$$\text{donc } C_n \underset{n\to\infty}{\sim} 2n\ln(n).$$

Ainsi, la complexité moyenne est quasi-linéaire.

**18** **N.B.** Pour faire les calculs exacts, il faudrait mettre des parties entières partout...

1.

2.

3. Dans le cas où la longueur de $T$ est multiple de 5 : notons $n$ cette longueur, et notons $M$ la médiane des médianes calculée. Il y a $n/5$ sous-tableaux. Parmi eux, $n/10$ ont une médiane inférieure à $M$. Et dans chacun de ces $n/10$ sous-tableaux, il y a trois éléments inférieurs à $M$ (la médiane elle-même, et deux éléments plus petits).

   Nous obtenons un total de $\dfrac{3n}{10}$ éléments inférieurs à $M$.

   De même, il y a aussi $\dfrac{3n}{10}$ éléments supérieurs à $M$.

   (On ne sait pas comment sont les $4n/10$ éléments restant.)

4.

5. Soit $n \in \mathbb{N}^*$. Appliquons la fonction à un tableau $T$ de $n$ éléments.

   - Pour commencer, on calcule chacune des médianes de sous-tableau de 5 éléments : ceci fait $\dfrac{n}{5}C_5$ comparaisons.
   - Déjà pour calculer la médiane de toutes ces médianes, il faut $C(n/5)$ comparaisons.
   - Ensuite, on segmente, en $n$ comparaisons (la fonction segmente a été étudiée en cours, elle n'a pas changé).
   - On obtient alors deux sous-tableaux. Vues les questions précédentes on sait que chaque sous-tableau fait au moins $3n/10$ éléments et au plus $7n/10$.

     Le pire des cas est celui on devra poursuivre la recherche de la médiane dans le plus grand des deux sous-tableau, qui est au plus de longueur $7n/10$.

     Ainsi, la suite de la recherche de la médiane aura une complexité inférieure à $C(7n/10)$ (on suppose $C$ croissante).
   - *Remarque :* Il y a encore une comparaison pour choisir le sous-tableau dans lequel poursuivre la recherche. On peut la négliger au sens où ce n'est pas la comparaison de deux éléments du tableau mais entre deux entiers directement disponibles (et petits). (ou alors augmenter $c$ de 1 pour être rigoureux).

   D'où le résultat, pour $c = C_5/5 + 1$.

6. Récurrence forte : posons pour tout $n \in \mathbb{N}^*$, $\mathcal{P}(n) : C(n) \leqslant 10n$.
   - Pour un tableau de longueur 1, il n'y aucune comparaison entre éléments du tableau (une comparaison pour se rendre compte que la longueur est 1). D'où $\mathcal{P}(1)$.
   - Soit $n \in \mathbb{N}$. Supposons $\forall k \in [\![1, n]\!]$, $\mathcal{P}(k)$. Alors :

$$\begin{aligned} C(n+1) &\leqslant C\left(\frac{n+1}{5}\right) + C\left(7\frac{n+1}{10}\right) + c(n+1) \\ &\leqslant \frac{n+1}{5}10c + 7\frac{n+1}{10}10c + (n+1)c \\ &\leqslant 2c(n+1) + 7c(n+1) + c(n+1) \\ &\leqslant 10c(n+1) \end{aligned}$$

   ↘ (par $\mathcal{P}\left(\left\lfloor \frac{n+1}{5} \right\rfloor\right)$ et $\mathcal{P}\left(\left\lfloor \frac{n+1}{5} \right\rfloor\right)$)

   D'où $\mathcal{P}(n+1)$.

   Par récurrence forte, on a bien $\forall n \in \mathbb{N}^*$, $C(n) \leqslant 10cn$.

   Ainsi, le calcul de la médiane se fait en temps linéaire dans tous les cas.

7.

8. La création du tableau des médianes à chaque étape nécessite un espace mémoire supplémentaire de $n/5$. On ne se contente donc pas uniquement du tableau initial comme dans la version de base du tri de Hoare.