Chapitre 3

Algorithmes de tris

1. Introduction

Outre l'intérêt intrinsèque que peut représenter le tri des éléments d'un ensemble, il peut être utile, en préalable à un traitement de données, de commencer par trier celles-ci. Considérons par exemple le problème de la recherche d'un élément dans un tableau. On sait que ce problème a un coût linéaire, mais si on prévoit de faire de nombreuses recherches, il peut être intéressant de commencer par trier ces données, car le coût d'une recherche dichotomique est logarithmique.
L'objet de ce chapitre est donc de décrire les principales méthodes de tri, et de faire l'étude de leur coût.

1.1 Présentation du problème

Avant toute chose, il importe de prendre conscience que la performance d'un tri va être directement liée à la structure de données utilisée et en particulier du temps d'accès à un élément : nous savons (cf. le chapitre 1) que l'accès à un élément d'un tableau est de coût constant contrairement à l'accès à un élément d'une liste chaînée qui est de coût linéaire. Certains algorithmes peuvent se révéler plus adaptés à l'une de ces deux structures qu'à l'autre. Les méthodes de tris peuvent aussi différer suivant que la structure de donnée à trier soit mutable ou pas : dans le cas d'une structure mutable, on cherchera à trier les éléments *en place*, c'est à dire sans coût spatial supplémentaire.
En ce qui nous concerne, nous allons essentiellement nous intéresser aux algorithmes de tris qui procèdent par comparaison entre les éléments d'un tableau. Nous allons donc considérer un tableau $t = [a_0, a_1, \dots, a_{n-1}]$ d'éléments d'un ensemble E muni d'une relation d'ordre $\preccurlyeq$ et nous autoriser les seules opérations suivantes :

- comparer deux éléments a_i et a_j à l'aide de la relation $\preccurlyeq$;
- permuter deux éléments a_i et a_j du tableau.

Nous supposerons ces deux opérations élémentaires de coûts constants.

Dans la pratique, ces algorithmes seront illustrés en Python par le tri d'une liste à valeurs numériques.

Remarque. Il existe des algorithmes qui n'utilisent pas de comparaison entre éléments mais tirent profit d'une information supplémentaire dont on dispose sur les éléments à trier. Le tri par base (*radix sort* en anglais) utilise la décomposition dans une base donnée des nombres entiers pour les trier. D'autres algorithmes tirent partie de la représentation en mémoire des objets à trier. Ces algorithmes seront brièvement évoqués dans la dernière partie de ce chapitre.

• Algorithmes stables

Il peut arriver que $\preccurlyeq$ soit seulement un pré-ordre total, c'est à dire une relation vérifiant :

- $\forall (x,y) \in \mathrm{E}^2$, $x \preccurlyeq y$ ou $y \preccurlyeq x$;
- $\forall x \in \mathrm{E}$, $x \preccurlyeq x$;
- $\forall (x,y,z) \in \mathrm{E}^3$, $x \preccurlyeq y$ et $y \preccurlyeq z \Rightarrow x \preccurlyeq z$;

Mais deux éléments x et y peuvent vérifier $x \preccurlyeq y$ et $y \preccurlyeq x$ sans avoir $x = y$ (ils sont alors dits *équivalents*).

L'exemple le plus courant consiste à trier des couples de valeurs en s'intéressant uniquement à la première composante de ces couples : la relation : $(x,y) \preccurlyeq (x',y') \iff x \preccurlyeq x'$ est un pré-ordre total sur $\mathbb{N}^2$.

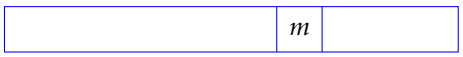
On dit d'un algorithme de tri qu'il est *stable* lorsqu'il préserve l'ordre des indices entre deux éléments équivalents. Autrement dit, si a_i et a_j sont équivalents et si $i < j$, alors dans le tableau trié a_i sera toujours placé avant a_j.

Nous allons maintenant étudier deux algorithmes de tri élémentaires : le tri par sélection et le tri par insertion, avant de nous intéresser aux algorithmes de tri les plus performants.

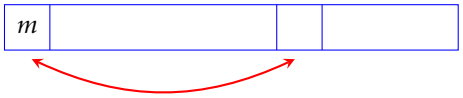
1.2 Le tri par sélection

Appelé *selection sort* en anglais, c'est l'algorithme le plus simple qui soit : on cherche d'abord le plus petit élément du tableau, que l'on échange avec le premier. On applique alors cette méthode au sous-tableau restant.

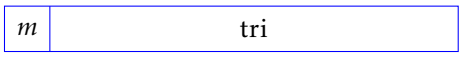
– recherche d'un élément minimal :



– permutation avec le premier :



– tri du tableau restant :



La description de cet algorithme montre que nous avons besoin d'une fonction qui calcule l'indice du minimum de la partie du tableau comprise entre les indices j et $n-1$. Pour des raisons de lisibilité nous allons la rédiger séparément de la fonction principale.

```
def minimum(t, j):
    i, m = j, t[j]
    for k in range(j+1, len(t)):
        if t[k] < m:
            i, m = k, t[k]
    return i
```

On justifie la validité de cette fonction en prouvant par récurrence l'invariant suivant :

à l'entrée de la boucle d'indice k, $t[i] = m$ et $m = \min\{t[\ell] \mid \ell \in [\![j, k-1]\!]\}$.

```
def select_sort(t):
    for j in range(len(t)-1):
        i = minimum(t, j)
        if i != j:
            t[i], t[j] = t[j], t[i]
```

On justifie la validité de cette fonction en prouvant par récurrence l'invariant suivant :

à l'entrée de la boucle d'indice j le tableau $t[0:j]$ est trié et $\forall k \geqslant j$, $t[j-1] \leqslant t[k]$.

• Étude de la complexité

Le nombre de comparaisons effectuées par la fonction `minimum(t, j)` est égal à $n-1-j$ donc le nombre total de comparaisons est égal à $\displaystyle\sum_{j=0}^{n-2}(n-1-j) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$ et le nombre d'échanges inférieur ou égal à $n-1$.

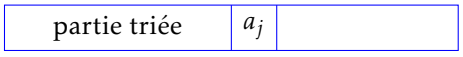
Il s'agit d'un algorithme de coût quadratique. Une de ses particularités est le nombre réduit (linéaire) d'échanges effectués. Il peut donc présenter un intérêt sur des données coûteuses à déplacer mais aisément comparables.

Remarque. Il s'agit d'un algorithme stable à condition, quand il y a plusieurs minimums équivalents, de permuter le premier rencontré. La version ci-dessus est stable (mais elle ne le serait pas si $<$ était remplacé par $\leqslant$ à la ligne 4 de la fonction `minimum`).

1.3 Le tri par insertion

Appelé *insertion sort* en anglais, il consiste à parcourir le tableau en insérant à chaque étape l'élément d'indice j dans la partie du tableau (déjà triée) à sa gauche, un peu à la manière d'un joueur de cartes insérant dans son jeu trié les cartes au fur et à mesure qu'il les reçoit.

– insertion dans un tableau trié :



– tri du tableau restant :

Nous allons commencer par rédiger une fonction qui a pour objet d'insérer l'élément $t[j]$ dans la partie du tableau $t[0:j]$ supposée triée par ordre croissant :

```
def insere(t, j):
    k = j
    while k > 0 and t[k] < t[k-1]:
        t[k-1], t[k] = t[k], t[k-1]
        k -= 1
```

On justifie la validité de cette fonction en prouvant l'invariant :

à l'entrée de la boucle d'indice $k > 0$ les tableaux $t[0:k]$ et $t[k:j+1]$ sont triés.

Remarque. Cette fonction respecte les exigences posées en début de chapitre (on s'autorise uniquement des permutations entre deux éléments du tableau) mais en contrepartie réalise un nombre inutilement élevé d'affectations ($2p$, où p est le nombre d'éléments de $t[0:j]$ qui sont supérieurs à $t[j]$). En procédant à un simple décalage on réduit le nombre d'affectations à $p+2$:

```
def insere(t, j):
    k, a = j, t[j]
    while k > 0 and t[k] < t[k-1]:
        t[k] = t[k-1]
        k -= 1
    t[k] = a
```

Quelle que soit la version choisie, la fonction principale s'écrit ensuite :

```
def insertion_sort(t):
    for j in range(1, len(t)):
        insere(t, j)
```

On justifie la validité de cette fonction en prouvant l'invariant :

à l'entrée de la boucle d'indice j le tableau $t[0:j]$ est trié.

• Étude de la complexité

Théorème. — *Dans le pire des cas, le nombre de comparaisons et d'échanges du tri par insertion est équivalent à* $\frac{n^2}{2}$.

Preuve. Insérer un élément dans un tableau trié de longueur j nécessite au plus j comparaisons et échanges donc en tout $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$ comparaisons et échanges. Cette situation a effectivement lieu dans le cas où la liste initiale est triée par ordre décroissant. □

Théorème. — *Le tri par insertion effectue en moyenne un nombre de comparaisons et d'échanges équivalent à* $\frac{n^2}{4}$.

Preuve. On suppose dans ce modèle que la suite des nombres à trier est la suite des entiers $1, 2, \ldots, n$ et l'on admet que toutes les permutations $(a_1, \ldots, a_n)$ de ces entiers sont équiprobables.
Rappelons qu'une *inversion* est un couple (i, j) vérifiant : $1 \leqslant i < j \leqslant n$ et $a_j < a_i$. Chaque permutation supprime une et une seule inversion et, une fois le tri terminé, il n'y a plus aucune inversion. Ainsi le nombre total d'échanges effectués est égal au nombre d'inversions dans la permutation initiale, et le nombre de comparaisons majoré d'au plus n. Calculer le nombre moyen d'échanges dans la procédure de tri par insertion revient donc à compter le nombre moyen d'inversions de l'ensemble des permutations sur n éléments.
Remarquons maintenant que l'image miroir d'une permutation σ est une permutation σ'. Il est clair que (i, j) est une inversion de σ si et seulement si ce n'est pas une inversion de σ'. La somme du nombre d'inversions de σ et de celles de σ' est donc égal à $\binom{n}{2} = \frac{n(n-1)}{2}$. On peut donc regrouper deux par deux les termes de la somme des nombres d'inversions des permutations sur n éléments et on obtient ainsi que le nombre moyen d'inversions sur l'ensemble des permutations est égal à : $\frac{n(n-1)}{4}$. □

Remarque. Le tri par insertion (tel qu'il est rédigé ci-dessus) est stable.

2. Algorithmes de tris efficaces

Nous venons d'étudier deux algorithmes de tri, tous deux de coût quadratique, aussi bien dans le pire des cas qu'en moyenne. Peut-on faire mieux ? Avant de répondre à cette question par l'affirmative, nous allons chercher à minorer le coût d'un algorithme de tri par comparaison, en introduisant la notion d'*arbre de décision*.

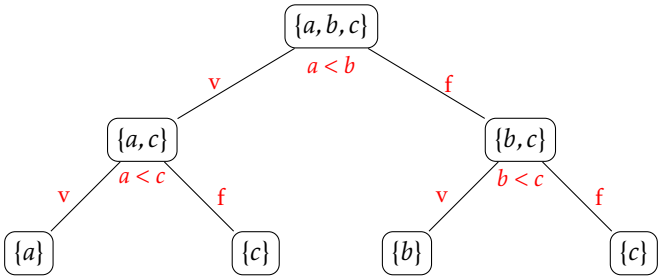
2.1 Coût minimal d'un algorithme de tri

• Arbre de décision

Soit $n \in \mathbb{N}^*$ un entier ; on désire faire un choix parmi n possibilités données dans un ensemble $\mathscr{C}$ de cardinal n.
Un *arbre de décision* est un arbre binaire tel que :

- la racine est étiquetée par $\mathscr{C}$;
- chacune des feuilles est étiquetée par un singleton inclus dans $\mathscr{C}$;
- chaque nœud qui n'est pas une feuille est étiqueté par un sous-ensemble $\mathscr{C}_1$ de $\mathscr{C}$, et ses deux fils par des sous-ensembles stricts et non vides $\mathscr{C}_2$ et $\mathscr{C}_3$ de $\mathscr{C}_1$ vérifiant : $\mathscr{C}_1 = \mathscr{C}_2 \cup \mathscr{C}_3$.

De plus, on adjoint à chaque nœud un test booléen ; les deux arêtes issues de ce nœud portent respectivement les labels v (pour *vrai*) en direction de $\mathscr{C}_2$, et f (pour *faux*) en direction de $\mathscr{C}_3$.
Voici par exemple un arbre de décision permettant de choisir le plus petit parmi trois nombres a, b et c :

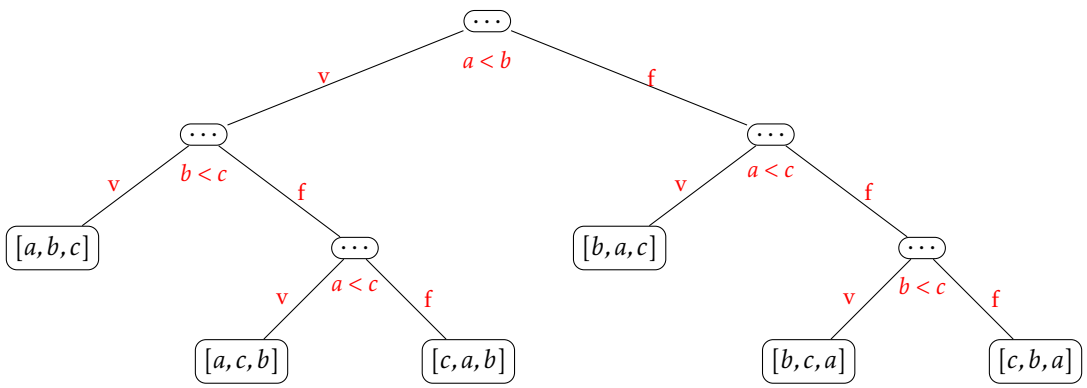


L'instruction définie par un nœud d'un arbre de décision est la suivante : on sait que le choix à réaliser se trouve dans l'ensemble $\mathscr{C}_1$. On effectue le test correspondant ; si le test est vrai le choix est à réaliser dans $\mathscr{C}_2$; sinon il est à réaliser dans $\mathscr{C}_3$.

Un arbre de décision pour un choix dans un ensemble de cardinal n doit posséder au moins n feuilles ; un arbre binaire de hauteur h possédant au plus 2^h feuilles, la hauteur h d'un arbre de décision doit vérifier l'inégalité : $2^h \geqslant n$, soit $h \geqslant \lceil \log n \rceil$.

• Tri par comparaison

Un algorithme de tri procédant par comparaison peut être associé à un arbre de décision sur l'ensemble $\mathscr{C}$ des $n!$ permutations possibles des éléments de ce tableau, les tests associés à chacun des nœuds étant une comparaison entre deux éléments du tableau.
Voici par exemple l'arbre de décision associé au tri par insertion d'un tableau à trois éléments $[a, b, c]$ (pour des raisons de lisibilité, les étiquettes des nœuds non terminaux n'ont pas été représentés) :



Sur cet arbre de décision chaque descente dans l'arbre correspond à une comparaison, accompagnée d'une permutation lorsqu'on se dirige vers le fils droit. De la sorte, il est aisé d'observer que le cas le plus favorable est celui du tableau déjà trié qui ne nécessite que deux comparaisons et aucune permutation, et que le cas le plus défavorable est celui du tableau trié à l'envers qui nécessite trois comparaisons et autant de permutations.

Sachant qu'un arbre de décision associé à un algorithme de tri par comparaison possède au moins $n!$ feuilles, on en déduit le résultat suivant :

Théorème. — *Tout algorithme de tri par comparaison utilise dans le pire des cas au moins* $\lceil \log n! \rceil$ *comparaisons.*

Preuve. La hauteur h de l'arbre de décision vérifie : $h \geqslant \lceil \log n! \rceil$, ce qui signifie qu'il existe au moins une configuration nécessitant $\lceil \log n! \rceil$ comparaisons, voire plus. □

Au prix d'un raisonnement un peu plus délicat, on peut même prouver le :

Théorème. — *Tout algorithme de tri par comparaison utilise en moyenne au moins* $\lceil \log n! \rceil$ *comparaisons.*

Preuve. Nous n'allons pas prouver complètement ce théorème mais nous contenter d'un résultat plus faible en prouvant (par récurrence sur f) que dans tout arbre binaire strict à f feuilles, ces dernières ont une profondeur moyenne supérieure ou égale à $\log f$. Si atteindre chacune de ces feuilles était équiprobable, nous pourrions conclure, mais rien ne dit que ce soit le cas.

- C'est bien évident lorsque $f = 1$.
- Si $f > 1$, supposons le résultat acquis jusqu'au rang $f-1$, et considérons un arbre $\mathscr{F}$ à f feuilles.
  Puisque $f \geqslant 2$, la racine de cet arbre possède un fils gauche $\mathscr{F}_1$ contenant f_1 feuilles et un fils droit $\mathscr{F}_2$ contenant f_2 feuilles, avec $1 \leqslant f_1$, $1 \leqslant f_2$ et $f_1 + f_2 = f$.
  La profondeur moyenne des feuilles de cet arbre est donc égale à :

$$h_m = \frac{1}{f}\sum_{x\in\mathscr{F}} h(x) = \frac{1}{f}\sum_{x\in\mathscr{F}_1}\Big(1+h_1(x)\Big) + \frac{1}{f}\sum_{x\in\mathscr{F}_2}\Big(1+h_2(x)\Big) = 1 + \frac{1}{f}\sum_{x\in\mathscr{F}_1} h_1(x) + \frac{1}{f}\sum_{x\in\mathscr{F}_2} h_2(x)$$

  où $h(x)$ désigne la profondeur de la feuille x dans $\mathscr{F}$, et $h_1(x)$ et $h_2(x)$ la profondeur respectivement dans $\mathscr{F}_1$ et dans $\mathscr{F}_2$.
  Par hypothèse de récurrence on a : $\dfrac{1}{f_k}\displaystyle\sum_{x\in\mathscr{F}_k} h_k(x) \geqslant \log f_k$, donc $h_m \geqslant 1 + \dfrac{1}{f}\Big(f_1 \log f_1 + f_2 \log f_2\Big)$.
  La fonction $x \mapsto x\log x$ étant convexe, on en déduit : $f_1 \log f_1 + f_2 \log f_2 \geqslant (f_1+f_2)\log\Big(\dfrac{f_1+f_2}{2}\Big)$, puis :

$$h_m \geqslant 1 + \log\Big(\frac{f}{2}\Big) = \log f.$$

□

Corollaire. — *Le coût d'un algorithme de tri par comparaison est en moyenne comme dans le pire des cas un* $\Omega(n\log n)$.

Preuve. La croissance de la fonction ln prouve l'encadrement : $\displaystyle\int_{k-1}^{k} \ln t\,\mathrm{d}t \leqslant \ln k \leqslant \int_{k}^{k+1} \ln t\,\mathrm{d}t$.

Sachant que $\log(n!) = \dfrac{1}{\ln 2}\displaystyle\sum_{k=2}^{n} \ln k$ on en déduit : $\dfrac{1}{\ln 2}\displaystyle\int_1^n \ln t\,\mathrm{d}t \leqslant \log(n!) \leqslant \dfrac{1}{\ln 2}\int_1^{n+1} \ln t\,\mathrm{d}t$.

Le calcul de ces intégrales conduit alors à : $n\log n - \dfrac{n-1}{\ln 2} \leqslant \log(n!) \leqslant (n+1)\log(n+1) - \dfrac{n}{\ln 2}$, ce qui prouve l'équivalent : $\log(n!) \sim n\log n$. □

Ce dernier résultat permet envisager l'existence d'algorithmes semi-linéaires pour trier un tableau. Nous allons en étudier deux dans les sections suivantes.

2.2 Le tri fusion

Appelée *merge sort* en anglais, nous l'avons déjà rencontré comme exemple d'algorithme récursif. Cette méthode adopte une approche « diviser pour régner » : on partage le tableau en deux parties de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$ que l'on trie par un appel récursif, puis on fusionne les deux parties triées.

– tri des deux moitiés du tableau : partie triée | partie triée

– fusion des parties triées : fusion

Malheureusement, une difficulté se présente : comment fusionner deux demi-tableaux en place, c'est à dire en s'autorisant uniquement la permutation de deux éléments dans le tableau ? Ce n'est pas impossible à réaliser mais la démarche est trop complexe pour être intéressante en pratique. C'est pourquoi nous allons déroger à nos exigences initiales en s'autorisant l'utilisation d'un tableau provisoire pour y stocker le résultat de la fusion. Ainsi, le schema de l'algorithme de tri fusion devient :

– tri des deux moitiés du tableau : partie triée | partie triée

– fusion dans un tableau auxiliaire : fusion

– recopie dans le tableau initial : recopie

La première fonction que nous allons rédiger est la fonction de fusion ; ses paramètres sont :

- le tableau initial t et trois indices $i \leqslant j \leqslant k$;
- le tableau auxiliaire *aux*.

On suppose les sous-tableaux $t[i : j]$ et $t[j : k]$ triés par ordre croissant et on les fusionne dans $aux[i : k]$.

```
def fusion(t, i, j, k, aux):
    a, b = i, j
    for s in range(i, k):
        if a == j or (b < k and t[b] < t[a]):
            aux[s] = t[b]
            b += 1
        else:
            aux[s] = t[a]
            a += 1
```

On notera que la position relative de deux éléments équivalents n'est pas modifiée, ce qui permettra d'avoir une fonction de tri stable.

```
def merge_sort(t):
    aux = [None] * len(t)
    def merge_rec(i, k):
        if k > i + 1:
            j = (i + k) // 2
            merge_rec(i, j)
            merge_rec(j, k)
            fusion(t, i, j, k, aux)
            t[i:k] = aux[i:k]
    merge_rec(0, len(t))
```

• Étude de la complexité

Coût spatial

Cette fonction a un coût spatial dû à la création du tableau *aux* (de coût linéaire) et à la pile d'exécution de la fonction récursive `merge_rec`. Notons $C(n)$ ce dernier coût. Il est raisonnable de penser qu'il est proportionnel

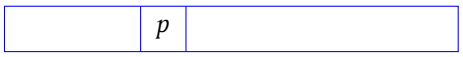
au nombre d'appels récursifs effectués (le contexte ne grandit pas) donc $C(n)$ vérifie une relation de récurrence de la forme : $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1)$.
Lorsque $n = 2^p$ la suite $u_p = C(2^p)$ vérifie la relation $u_p = 2u_{p-1} + \Theta(1)$ donc $u_p = \Theta(2^p)$ et $C(n) = \Theta(n)$. Nous admettrons que ce résultat reste vrai pour un entier n quelconque. Ainsi, *le coût spatial du tri fusion est linéaire*.

Coût temporel
Notons maintenant $C(n)$ le coût temporel du tri fusion. Il est clair que la fusion des sous-tableaux $t[i:j]$ et $t[j:k]$ dans le tableau *aux* puis la copie de $aux[i:k]$ vers $t[i:k]$ sont des opérations qui s'effectuent en coût linéaire vis-à-vis de $k-i$ donc $C(n)$ vérifie une relation de récurrence de la forme : $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n)$.
Lorsque $n = 2^p$ la suite $u_p = C(2^p)$ vérifie la relation $u_p = 2u_{p-1} + \Theta(2^p)$ que l'on peut encore écrire : $\frac{u_p}{2^p} = \frac{u_{p-1}}{2^{p-1}} + \Theta(1)$. Par télescopage on en déduit $\frac{u_p}{2^p} = \Theta(p)$ puis $u_p = \Theta(p2^p)$, soit $C(n) = \Theta(n \log n)$. Nous admettrons que cette formule reste vraie pour un entier n quelconque. Ainsi, *le coût temporel du tri fusion est semi-logarithmique*.

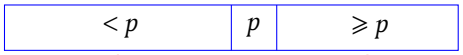
2.3 Le tri rapide

Appelé *quick sort* en anglais, ce tri adopte lui aussi une démarche de type « diviser pour régner » : on commence par segmenter le tableau autour d'un pivot choisi parmi les éléments du tableau en plaçant les éléments qui lui sont inférieurs à sa gauche, et les éléments qui lui sont supérieurs, à sa droite. À l'issue de cette étape, le pivot se trouve à sa place définitive, et les parties gauche et droite sont triées par l'intermédiaire d'un appel récursif.

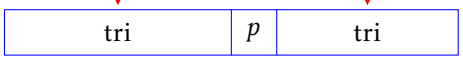
– choix d'un pivot :



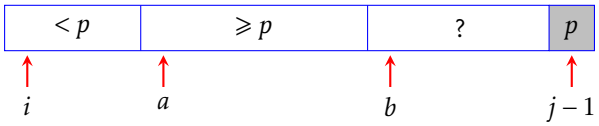
– segmentation :



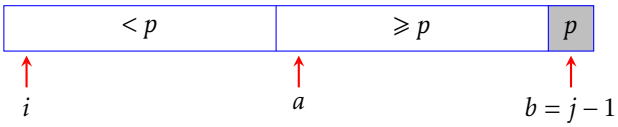
– appels récursifs :



Il existe plusieurs façons de segmenter une portion de tableau $t[i:j]$. la méthode que nous allons suivre consiste à choisir pour pivot l'élément $p = t[j-1]$ et à maintenir l'invariant suivant :



Les valeurs initiales sont $a = b = i$ et la condition d'arrêt $b = j-1$. Lorsque le processus se termine, la situation est la suivante :



et il suffit alors de permuter les cases d'indices a et $j-1$ pour achever la segmentation du tableau $t[i:j]$.

```
def segmente(t, i, j):
    p = t[j-1]
    a = i
    for b in range(i, j-1):
        if t[b] < p:
            t[a], t[b] = t[b], t[a]
            a += 1
    t[a], t[j-1] = t[j-1], t[a]
    return a
```

On notera que cette fonction renvoie l'indice a où se trouve désormais le pivot p.

```
def quick_sort(t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if i + 1 < j:
        a = segmente(t, i, j)
        quick_sort(t, i, a)
        quick_sort(t, a + 1, j)
```

Remarque. L'algorithme de tri rapide présenté ici n'est pas stable.

• Étude de la complexité

Il apparaît clairement que le coût de la segmentation est linéaire. Cette dernière ayant pour objet de partager le tableau en deux sous-ensembles de tailles n_1 et n_2 avec $n_1+n_2=n-1$ (le pivot n'appartient à aucun de ces deux sous-ensembles), le coût de l'algorithme de tri rapide vérifie la relation : $C(n)=C(n_1)+C(n_2)+\Theta(n)$.

L'intuition nous pousse à penser que cette méthode est d'autant meilleure que n_1 et n_2 sont proches et qu'à l'inverse un partitionnement très déséquilibré risque d'avoir une influence négative sur le coût total. Avant de justifier rigoureusement ces assertions, intéressons-nous à ces deux situations extrêmes.

Partitionnement dans le cas le plus défavorable
Supposons qu'à chaque partitionnement l'une des deux parties du tableau segmenté soit vide : nous avons alors $n_1=n-1$ et $n_2=0$ et la relation de récurrence devient : $C(n)=C(n-1)+\Theta(n)$, ce qui conduit immédiatement à $C(n)=\Theta(n^2)$. Cette situation a lieu par exemple lorsque le tableau est déjà trié (dans un sens comme dans l'autre) puisque le pivot choisi est à chaque étape un élément extrémal de la partie à segmenter.

Partitionnement dans le cas le plus favorable
À l'inverse, supposons qu'à chaque étape du partitionnement les deux parties du tableau segmenté soient exactement de même taille ; nous avons alors $n=2^p-1$ et la relation de récurrence devient $C(n)=2C(\lfloor n/2\rfloor)+\Theta(n)$. Posons $u_p=C(2^p-1)$; alors $u_p=2u_{p-1}+\Theta(2^p)$. Nous avons déjà effectué ce calcul pour le tri fusion et obtenu $u_p=\Theta(p2^p)$ soit $C(n)=\Theta(n\log n)$.

Théorème. — *Lorsque le choix du pivot est arbitraire, le coût de l'algorithme de tri rapide dans le pire des cas est quadratique.*

Preuve. Nous avons déjà montré qu'il existe des configurations (tableaux déjà triés) pour lesquelles le coût est quadratique. Montrons maintenant que dans tous les cas on a $C(n)=O(n^2)$ en raisonnant par induction :
Supposons L'existence d'une constante M telle que $C(k)\leqslant Mk^2$ pour tout $k<n$. Alors

$$C(n)\leqslant M(n_1^2+n_2^2)+\Theta(n)=M(n_1^2+(n-1-n_1)^2)+\Theta(n).$$

Il est facile de montrer que la fonction $x\mapsto x^2+(n-1-x)^2$ est maximale aux extrémités de l'intervalle $[0,n-1]$ donc $C(n)\leqslant M(n-1)^2+\Theta(n)=Mn^2-M(2n-1)+\Theta(n)$. Il suffit donc d'avoir choisi une constante M suffisamment grande pour que le terme $M(2n-1)$ domine le terme en $\Theta(n)$ pour obtenir $C(n)\leqslant Mn^2$. □

Partitionnement équilibré
Pourquoi le qualifier de tri rapide alors que le cas défavorable est quadratique ? Parce qu'en moyenne le coût est semi-linéaire. Avant de le justifier rigoureusement, nous allons imaginer le cas de figure où le partitionnement produise systématiquement un découpage dans une proportion de 9 pour 1, ce qui paraît à première vue assez déséquilibré. On obtient dans ce cas la relation $C(n)=C(n/10)+C(9n/10)+\Theta(n)$. Nous allons montrer par induction que $C(n)=\Theta(n\log n)$.

Supposons l'existence d'une constante M telle que $C(k)\leqslant Mk\log k$ pour tout $k<n$. Alors

$$C(n)\leqslant M\frac{n}{10}(\log n-\log 10)+M\frac{9n}{10}(\log n+\log 9-\log 10)+\Theta(n)=Mn\log n-\frac{Mn}{10}(10\log 10-9\log 9)+\Theta(n).$$

Il suffit donc de choisir une constante M suffisamment grande pour en déduire $C(n)\leqslant Mn\log n$.
Bien entendu, tout découpage ayant un facteur de proportionnalité constant donnera un résultat identique.

Théorème. — *En moyenne, le coût du tri rapide est un $\Theta(n\log n)$.*

Preuve. Nous allons déterminer le nombre moyen de comparaisons effectuées en faisant l'hypothèse qu'à l'issue du processus de segmentation le pivot a la même probabilité de se trouver dans chacune des n cases du tableau. Le nombre moyen c_n de comparaisons vérifie alors la relation :

$$c_n = n-1+\frac{1}{n}\sum_{k=0}^{n-1}(c_k+c_{n-k-1}) = n-1+\frac{2}{n}\sum_{k=0}^{n-1}c_k.$$

Écrivons alors : $nc_n = n(n-1)+2\sum_{k=0}^{n-1}c_k$ et $(n-1)c_{n-1} = (n-1)(n-2)+2\sum_{k=0}^{n-2}c_k$ et soustrayons ; on obtient $nc_n-(n-1)c_{n-1} = 2(n-1)+2c_{n-1}$ qui peut encore s'écrire : $nc_n-(n+1)c_{n-1} = 2(n-1)$.

Transformons cette égalité en divisant par $n(n+1)$; on obtient $\frac{c_n}{n+1}-\frac{c_{n-1}}{n} = \frac{4}{n+1}-\frac{2}{n}$, une égalité télescopique qui conduit en sommant à : $\frac{c_n}{n+1}-c_0 = 4\sum_{k=1}^{n}\frac{1}{k+1}-2\sum_{k=1}^{n}\frac{1}{k} = 2\sum_{k=1}^{n}\frac{1}{k}+\frac{4}{n+1}-4.$

Sachant que $c_0 = 0$ on obtient : $c_n = 2(n+1)h_n-4n$, où $h_n = \sum_{k=1}^{n}\frac{1}{k}$ est la série harmonique.

On sait que $h_n \sim \ln n$, donc $c_n \sim 2n\ln n \approx 1,4n\log n$. □

- **Choix du pivot**

Nous l'avons constaté, le choix du dernier élément comme pivot pour segmenter $t[i:j]$ donne en moyenne un coût semi-linéaire, mais pour des tableaux triés ou quasiment triés le coût devient quadratique. Dans le cas de figure où l'on prévoit d'appliquer *quicksort* à des données peu mélangées, on peut avoir intérêt à choisir au hasard un pivot dans $t[i:j]$. La modification de la fonction de segmentation est mineure : il suffit de tirer au hasard un entier α dans l'intervalle $[\![i, j-1]\!]$ puis à échanger les éléments $t[\alpha]$ et $t[j-1]$ avant de procéder à la segmentation. Le pivot étant maintenant choisi au hasard on peut s'attendre à ce que le partitionnement du tableau d'entrée soit en moyenne relativement équilibré.

3. Tris en temps linéaire

La borne minimale $\Omega(n\log n)$ que nous avons établie à la section 2.1 ne s'applique qu'à des algorithmes opérant par comparaisons *entre éléments du tableau* exclusivement. Elle suppose en outre que le coût d'une comparaison puisse s'effectuer à coût constant, ce qui n'est pas forcément réaliste si on envisage de trier des chaînes de caractères de grandes tailles. Il existe des algorithmes qui s'exécutent en temps linéaire en exploitant certaines informations relatives à la nature des objets à trier.

3.1 Tri par dénombrement

L'exemple le plus évident est sans doute l'algorithme de tri par dénombrement, applicable lorsque les éléments à trier ne peuvent prendre qu'un nombre fini de valeurs. Considérons le cas d'un tableau t de longueur n dont les valeurs sont prises dans l'intervalle entier $[\![0, k-1]\!]$. En un parcours du tableau t il est possible de dénombrer le nombre d'apparitions de chacune des k valeurs possibles, au prix d'un espace mémoire proportionnel à k. Chaque valeur de l'intervalle $[\![0, k-1]\!]$ est ensuite rangée dans le tableau t autant de fois que nécessaire.

```
def counting_sort(t, k):
    occ = [0] * k
    for i in t:
        occ[i] += 1
    s = 0
    for i in range(k):
        for j in range(occ[i]):
            t[s] = i
            s += 1
```

À l'issue de la première boucle des lignes 3-4, le tableau *occ* contient dans sa case d'indice i le nombre d'occurrences de $i \in [\![0, k-1]\!]$ dans le tableau t. Le coût temporel de cette première phase est un $\Theta(n)$. La boucle

des lignes 6-9 écrit pour chacune des valeurs de $i \in [\![0, k-1]\!]$ autant de fois i dans le tableau t que nécessaire. Cette opération est de coût $\Theta(k+n)$.
Le coût temporel de l'algorithme de tri par comptage est donc un $\Theta(n+k)$, auquel s'ajoute un coût spatial en $\Theta(k)$ lié à la création du tableau *occ*. Il est raisonnable de l'utiliser lorsque $k = O(n)$ et donne dans ce cas un algorithme de coût linéaire.
Notons encore une fois que la borne minimale $\Omega(n \log n)$ ne s'applique pas puisqu'on ne procède pas à des comparaisons entres éléments du tableau t.

• Cas d'un pré-ordre

Lorsqu'on utilise un pré-ordre le tri par dénombrement est un peu plus couteux en espace car il devient nécessaire d'utiliser un second tableau de taille n pour y ranger les éléments triés. Nous allons supposer connue une fonction $c : E \to [\![0, k-1]\!]$ qui servira de clé du tri : $\forall (x, y) \in E^2, x \preccurlyeq y \iff c(x) \leqslant c(y)$. Ainsi défini, $\preccurlyeq$ est un pré-ordre.

Un premier parcours du tableau t permet de connaître le nombre d'occurrence de chacune des clés et donc l'emplacement de chaque zone du tableau de sortie qui sera attribuée à une clé donnée. Un second parcours de t permet le rangement des données ordonnées par la fonction c dans le tableau de sortie.

```
def counting_sort(t, c, k):
    occ = [0] * k
    s = [None] * len(t)
    for x in t:
        occ[c(x)] += 1
    for i in range(1, k):
        occ[i] += occ[i-1]
    for x in reversed(t):
        occ[c(x)] -= 1
        s[occ[c(x)]] = x
    t[:] = s
```

Après la boucle des lignes 6-7, $occ[i]$ contient le nombre d'éléments dont la clé est inférieure ou égale à i. En parcourant le tableau t par ordre décroissant, on place chaque élément à sa place définitive dans le tableau de sortie s.
Illustrons ceci en triant un tableau d'entiers suivant leur dernier chiffre en base 10 :

```
In [1]: t = [123, 451, 678, 942, 102, 134, 156, 741, 684, 235, 910, 236]

In [2]: counting_sort(t, lambda x: x % 10, 10)

In [3]: t
Out[3]: [910, 451, 741, 942, 102, 123, 134, 684, 235, 156, 236, 678]
```

En procédant ainsi on obtient un algorithme de tri stable [1] dont la complexité tant temporelle que spatiale est un $\Theta(n+k)$.

3.2 Tri par base

Appelé *radix sort* en anglais, cette famille d'algorithmes de tri utilise la décomposition des objets à trier dans une base donnée. Ces objets peuvent être bien entendu des nombres (par exemple décomposés en base 10) mais aussi des chaînes de caractères (vues comme des quantités décomposées en base 26).

• LSD radix sort

Une première méthode consiste à trier les éléments en commençant par leur chiffre le moins significatif (*least signifiant digit first radix sort*), autrement dit en les considérant de la droite vers la gauche : les éléments sont tout d'abord triés suivant leur dernier chiffre, puis suivant l'avant-dernier, etc. (illustration figure 1). Cette démarche suppose que tous ces éléments sont de même taille (si les données à trier sont des chaînes de caractères il est nécessaire qu'elles soient toutes de même longueur pour pouvoir être triées par ordre lexicographique).

Notez bien qu'il est impératif que le tri utilisé pour ranger les éléments suivant leur k-ème chiffre soit stable.

1. C'est la raison pour laquelle le tableau `t` est parcouru dans le sens rétrograde lors de la troisième boucle.

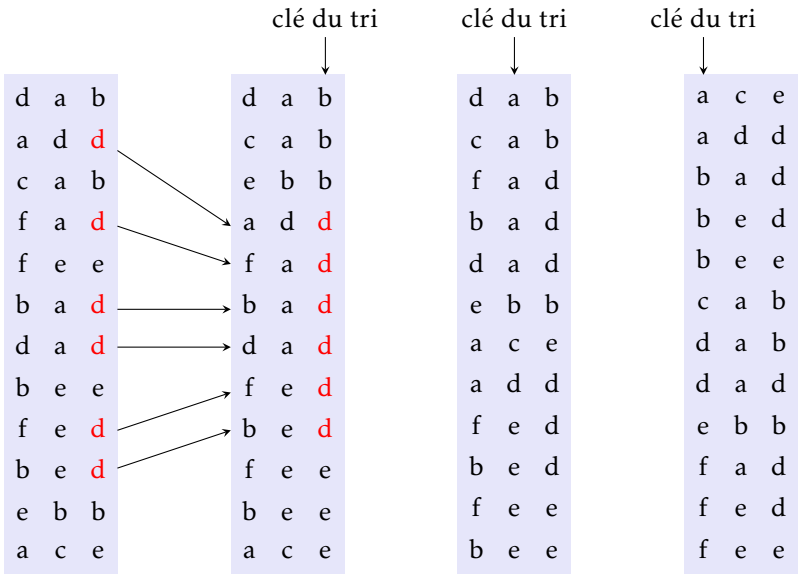


FIGURE 1 – Un exemple de tri par base à partir du chiffre le moins significatif.

Exemple. Poursuivons le tri du tableau entamé dans la section précédente en triant le tableau obtenu suivant le second chiffre et enfin suivant le troisième :

```
In [4]: counting_sort(t, lambda x: (x // 10) % 10, 10)
In [5]: t
Out[5]: [102, 910, 123, 134, 235, 236, 741, 942, 451, 156, 678, 684]

In [6]: counting_sort(t, lambda x: (x // 100) % 10, 10)
In [7]: t
Out[7]: [102, 123, 134, 156, 235, 236, 451, 678, 684, 741, 910, 942]
```

Ce tri peut se faire en temps linéaire, par exemple par comptage pour un coût en $\Theta(n+k)$ où k désigne la base choisie. Si les nombres à trier s'écrivent avec d chiffres le coût total de LSD radix sort est un $\Theta(d(n+k))$.

On observera que pour que cet algorithme soit plus intéressant qu'un algorithme de tri par comparaison en $\Theta(n\log n)$ il est nécessaire qu'on ait au pire $d = O(\log n)$. Il peut être intéressant pour trier de grandes listes de données de taille fixe (code postal, date, numéro de sécurité sociale, ...)

• MSD radix sort

Une seconde méthode consiste à trier les éléments en commençant par le chiffre le plus significatif (*most signifiant digit first radix sort*) puis à procéder récursivement pour chaque partie du tableau débutant par le même chiffre (illustration figure 2).

Bien que cette démarche paraisse de prime abord plus naturelle que LSD radix sort, la multiplicité des appels récursifs qui peuvent potentiellement intervenir dans ce tri rend cette seconde approche inadaptée à des langages pour lesquels la récursivité est peu ou mal optimisée (PYTHON pour ne pas le citer). En revanche, elle possède quelques avantages :

- elle est adaptée au tri lexicographique de chaînes de caractères de longueurs variables ;
- *elle ne nécessite pas d'examiner nécessairement chacune des clés.*

En effet, les appels récursifs ne sont pas nécessaires dès lors qu'il ne reste plus qu'un seul objet à trier : dans l'exemple qui nous sert à illustrer ces méthodes, seuls 78% des caractères ont étés examinés (voir figure 3).

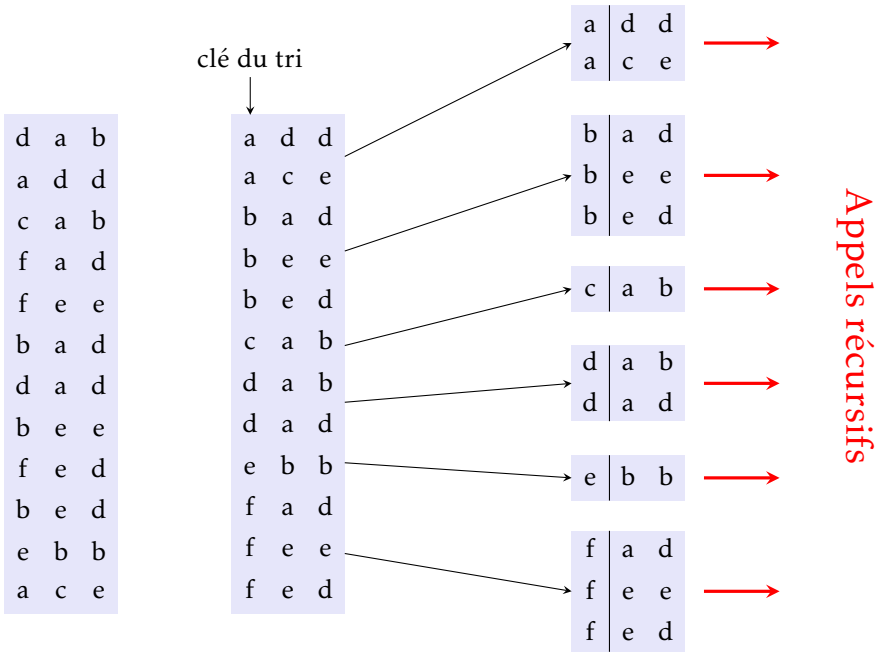


FIGURE 2 – Un exemple de tri par base à partir du chiffre le plus significatif.

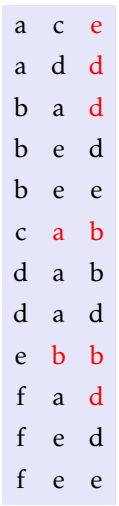


FIGURE 3 – Les caractères qui n'ont pas été examinés par MSD radix sort sont indiqués en rouge.

Cette proportion peut se révéler beaucoup plus basse dans le cas de chaînes de caractères de plus grandes longueurs.

• MSD radix sort et segmentation

Enfin on notera l'existence d'un algorithme récent (1997) qui combine les avantages du tri rapide et du tri par base pour trier efficacement des chaînes de caractères. L'idée est de procéder à une segmentation du tableau sur le premier caractère à la manière de l'algorithme du drapeau tricolore (voir exercice 6) puis à procéder récursivement. Les caractères égaux au caractère ayant servi de pivot n'auront pas à être examinés de nouveau (illustration figure 4).
Cet algorithme se révèle être l'un des plus efficace pour trier des chaînes de caractères.

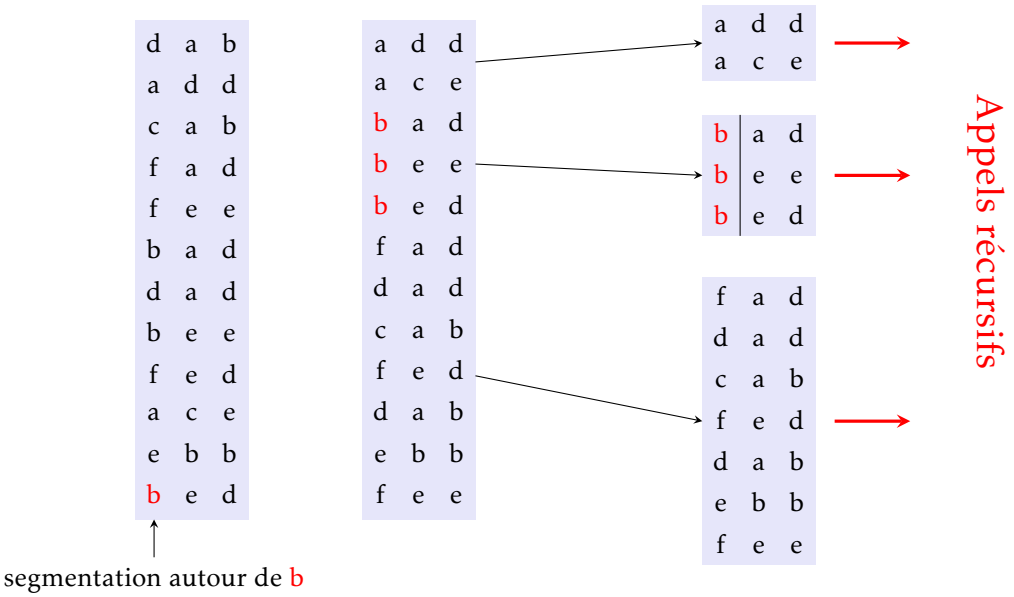


FIGURE 4 – MSD radix sort par segmentation.

4. Exercices

4.1 Tris par comparaison

Exercice 1 Rédiger des versions récursives des algorithmes de tri par sélection et de tri par insertion.

Exercice 2 On considère un tableau à n cases contenant les entiers de $[\![1,n]\!]$ et on suppose toutes les permutations de $\mathfrak{S}_n$ équiprobables. Calculer le nombre moyen d'échanges effectués par l'algorithme de tri par sélection, et en donner un développement asymptotique.

Exercice 3 Montrer que $n-1$ comparaisons et échanges *entre éléments consécutifs* permettent de placer l'élément maximal en queue d'un tableau à trier. En déduire une nouvel algorithme de tri (qui s'appelle le tri bulle, *bubble sort* en anglais), que vous rédigerez en PYTHON.
Faire une analyse de son coût.

Exercice 4 Rédiger une version non récursive du tri fusion.

Exercice 5 Imaginé par Donald SHELL en 1959, le tri de SHELL est une optimisation du tri par insertion.
Dans le tri par insertion, un élément se rapproche de sa place finale en progressant lentement, case par case. L'accélération consiste à le déplacer en commençant par faire des grands pas, puis des pas de plus en plus petits, jusqu'à, évidemment, des pas de 1 pour que le tableau soit trié.
On considère une suite d'entiers $(h_p)_{p\in\mathbb{N}^*}$ strictement croissante, débutant par $h_1 = 1$. Pour tout $n \in \mathbb{N}^*$, il existe donc un unique entier $p \in \mathbb{N}^*$ tel que $h_p \leqslant n < h_{p+1}$.
L'étape de base de l'algorithme consiste à trier à l'aide du tri par insertion chacun des sous-tableaux débutant respectivement par $a_0,\dots,a_{h_p-1}$, et dont les éléments sont séparés de h_p cases. Une fois cette étape achevée, on dit que le tableau initial est h_p-trié. On répète alors cette opération de base avec la valeur h_{p-1}, et ce jusqu'à la valeur finale 1.

Dans un premier temps on suppose $h_p = 2^p$, et on note $C(n)$ le nombre maximal d'échanges effectué par l'algorithme de SHELL pour trier un tableau de longueur n.

a) Intéressons nous d'abord à la dernière étape de cet algorithme : après le 2-tri, les éléments du tableau vérifient : $a_0 \leqslant a_2 \leqslant a_4 \leqslant \cdots$ et $a_1 \leqslant a_3 \leqslant a_5 \leqslant \cdots$. Quel est le nombre maximal d'échanges effectués par le 1-tri (c'est à dire le tri par insertion du tableau ainsi préparé) ?

b) En déduire un exemple de tableau à 4 cases pour lequel le nombre d'échanges par l'algorithme de SHELL est maximal, puis un exemple avec un tableau à 8 cases.

c) On note $u_p = C(2^p)$. Montrer que $u_p = 2u_{p-1} + 2^{p-2}(2^{p-1}+1)$, et en déduire une forme close de u_p.
Pensez-vous que ce choix pour la suite $(h_p)_{p\in\mathbb{N}^*}$ soit pertinent ?

d) Hibbard a démontré en 1963 que la suite $h_p = 2^p - 1$ conduit à un coût en $\Theta(n^{3/2})$. Programmer en Python le tri de Shell pour cette suite de valeurs.

Remarque. Connaître la suite (h_p) conduisant à la meilleure complexité de ce tri est à l'heure actuelle un problème ouvert. Des considérations empiriques conduisent à penser que ses premiers termes sont 1, 4, 10, 23, 57, 132, 301, 701 et conduisent à suggérer une croissance géométrique de l'ordre de 2,2.

Exercice 6 On considère l'algorithme suivant :

```
def tri(t, i, j):
    if t[i] > t[j-1]:
        t[i], t[j-1] = t[j-1], t[i]
    if j - i > 2:
        k = (j - i) // 3
        tri(t, i, j-k)
        tri(t, i+k, j)
        tri(t, i, j-k)
```

a) Montrer que si t est un tableau de taille n, l'exécution de `tri(t, 0, n)` trie correctement le tableau t.

b) Donner une relation de récurrence vérifiée par le coût temporel $C(n)$ de cet algorithme, puis la résoudre lorsque $n = 3^p$. Cet algorithme de tri mérite-t-il de passer à la postérité ?

4.2 Segmentation et tri rapide

Exercice 7 (Algorithme du *drapeau tricolore*)
On considère un tableau contenant des objets possédant une couleur qui ne peut être que rouge, blanc ou bleu. On souhaite trier ce tableau de sorte que les éléments rouges soient placés en tête, ensuite les éléments blancs, puis enfin les éléments bleus [2].
Pour modéliser cette situation en Python, on suppose que les objets à trier possèdent un attribut `couleur` ne pouvant prendre que les valeurs `'rouge'`, `'blanc'` ou `'bleu'`.
Rédiger en Python une fonction effectuant ce tri en procédant par segmentation ; l'algorithme ne devra effectuer qu'un seul parcours du tableau, et ne déterminer la couleur qu'une fois par élément.

Exercice 8 Déduire du principe de segmentation un algorithme, qu'on rédigera en Python, pour déterminer le k^e plus petit élément d'un vecteur, puis pour déterminer l'élément médian d'un tableau. Que penser du coût de ce dernier algorithme ?

Exercice 9 Lorsque $\alpha \in]0,1[$ et $t = [a_0,\dots,a_{n-1}]$, un α-pseudomédian du tableau t est un entier $k \in [\![0,n-1]\!]$ tel que au moins n^α éléments du tableau sont plus petits que a_k, et au moins n^α plus grands.
On suppose que n est une puissance de 3, et on applique l'algorithme suivant :

- si $n = 3$, on trie les trois valeurs, et on renvoie l'élément médian ;
- sinon, on sépare la liste en $n/3$ sous-listes à trois éléments, on trie chacune de ses sous-listes, et on applique récursivement l'algorithme à la liste des $n/3$ éléments médians.

a) Calculer le nombre c_n de comparaisons effectuées par cet algorithme.

b) Montrer que cet algorithme calcule un élément α-pseudomédian de t pour une valeur de α à déterminer.

c) Expliquer en quoi cet algorithme peut être utile pour améliorer le coût dans le pire des cas de l'algorithme de tri rapide.

4.3 Tris de coûts linéaires

Exercice 10 Montrer que n éléments de $[\![1,p]\!]$ peuvent être triés à l'aide d'un nombre de comparaisons n'excédant pas $n\log p$ (procéder par segmentation).

2. ce problème est attribué à Edsger Dijkstra, il suit donc l'ordre des couleurs du drapeau néerlandais.

Exercice 11 Le tri crêpe (*pancake sort* si on préfère) est destiné aux cuisiniers maladroits incapables de faire deux crêpes de même taille. Ces derniers se retrouvent donc en face d'une pile de crêpes de tailles différentes, et doivent les ordonner par taille décroissante, la plus petite devant se trouver au sommet de la pile.
Pour réaliser cette opération, ils ne disposent que d'une spatule qu'ils peuvent insérer entre deux crêpes pour retourner d'un coup toutes les crêpes situées au dessus de celle-ci.

a) Montrer qu'un tas de n crêpes peut être ordonné à l'aide d'au plus $2n - 3$ manipulations.

b) Un cuisinier particulièrement maladroit a brulé une face de chacune de ses crêpes. Peut-il trier son tas de crêpes en imposant en plus que les faces brulées soient invisibles ?