2. Quelques algorithmes de tri

Pour trier un tableau de n valeurs, nous allons étudier un algorithme naïf (parmi d'autres) dont la complexité est un $O\left(n^2\right)$. Puis la stratégie "diviser pour régner" permettra d'élaborer des algorithmes de tri en $O\left(n \log_2 n\right)$. On peut montrer que cette complexité est **optimale**, parmi tous les algorithmes de tri basés sur des comparaisons entre valeurs du tableau (eh oui, on peut faire autrement, voir par exemple le *tri par dénombrement* !).

I - Tri par insertion

1) Version itérative

L'idée est de trier progressivement le tableau : supposant que $t\,[0:k]$ est déjà trié, j'insère $t\,[k]$ à sa place parmi les valeurs de $t\,[0:k]$ (en décalant les plus grandes valeurs d'un cran vers la droite si nécessaire) de sorte que $t\,[0:k+1]$ se retrouve trié.
Le principe de modularité pourrait nous conduire à écrire une fonction qui détermine la place de $t\,[k]$, puis une deuxième fonction qui se charge de l'insertion. On obtient un code plus compact en décalant les valeurs au fur et à mesure de la recherche de la place de $t\,[k]$ (sans oublier de mémoriser la valeur de $t\,[k]$, qui risque d'être écrasée très vite !).

Plus précisément, pour k allant de 1 à $n-1$:

- je stocke la valeur de $t\,[k]$ dans une variable *temp* et j'initialise j à la valeur k
- tant que $j > 0$ et $temp < t\,[j-1]$ (j'ai une valeur supérieure à *temp* à gauche de $t\,[j]$), je décale $t\,[j-1]$ d'un cran vers la droite et je décrémente j
- à la sortie de la boucle j'installe *temp* à sa place.

La fonction de tri peut ainsi s'écrire en Python :

```
def tri_ins(t):
    for k in range(1,len(t)):
        temp=t[k]
        j=k
        while j>0 and temp<t[j-1]:
            t[j]=t[j-1]
            j-=1
        t[j]=temp
    return t
```

Noter que cette fonction **modifie le tableau** t. Il faut penser si besoin à en effectuer une copie...

Le `return t` est facultatif, si l'on considère ce programme comme une **procédure** modifiant t et non comme une **fonction** renvoyant un tableau.

Le test de la boucle `while` n'est correct que grâce à l'évaluation paresseuse des expressions booléennes, sans laquelle on risquerait d'accéder à $t[-1]$ (ce qui existe en Python, mais pourrait poser problème dans d'autres contextes).

Justification :

- pour la boucle `while`, la valeur de j diminue strictement à chaque passage, il y aura donc au maximum k itérations ; terminaison naturelle pour la boucle `for` ;
- preuve par récurrence de l'invariant de boucle $\mathcal{P}_k$: "après l'exécution de la boucle `for` pour la valeur k les valeurs $t[0], \ldots, t[k]$ sont les valeurs initiales rangées par ordre croissant".

Complexité au pire : $\displaystyle\sum_{k=1}^{n-1} k = \frac{n\,(n-1)}{2}$ (obtenue lorsque le tableau est initialement rangé en ordre décroissant !).

Complexité au mieux : $\displaystyle\sum_{k=1}^{n-1} 1 = n-1$ (obtenue lorsque le tableau est initialement rangé en ordre croissant).

Complexité en moyenne : en admettant que l'insertion se fait "en moyenne" au milieu du segment balayé, on obtient $\sum_{k=1}^{n-1} \frac{k}{2} = \frac{n(n-1)}{4}$ (l'étude précise peut se faire à l'aide de la notion d'*inversions d'une permutation* et conduit au même ordre de grandeur).

2) Version récursive

L'idée de l'algorithme se prête bien à une vision récursive, en écrivant une procédure `tri_ins(t,j)` qui trie (récursivement) la tranche $t[:j]$:

- si $j = len(t)$, il n'y a rien à faire
- sinon j'insère $t[j]$ dans la tranche $t[:j-1]$ (qui à ce stade est triée, cf. l'hypothèse de récurrence dans la preuve) puis je lance l'appel récursif `tri_ins(t,j+1)`.

Pour la modularité, j'écris une procédure récursive qui insère $t[j]$ dans la tranche $t[:j-1]$ supposée triée.

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

```
def tri_ins(t,j=1):
    if j<len(t):
        insere(t,j)
        tri_ins(t,j+1)
```

Noter la fonctionnalité de Python, qui permet d'omettre lors d'un appel de fonction un paramètre, pourvu que celui-ci se voie attribuer une valeur "par défaut" lors de la définition de la fonction (ici le `j=1` dans la définition de `tri_ins`). Ainsi l'appel initial naturel `tri_ins(t)` sera interprété comme `tri_ins(t,1)`. Avec un langage qui n'offre pas cette possibilité, il suffit de définir une fonction auxiliaire, que la fonction principale appelle avec les paramètres adéquats.

3) Insertion dichotomique

On peut améliorer l'algorithme précédent en effectuant une recherche dichotomique de la place de l'élément à insérer dans la tranche qui le précède, puisqu'elle est triée. Cela permet de ramener le nombre de comparaisons à un $O(n \log_2 n)$, **mais** cela n'évite pas les décalages d'éléments du tableau, qui conduiront à une complexité temporelle quadratique...

II - Tri par fusion

1) Principe et implémentation

L'idée récursive est naturelle :

- s'il y a au plus une valeur, le tableau est trié ;
- s'il y a au moins deux valeurs, couper le tableau en deux, trier (récursivement) les deux sous-tableaux obtenus, puis fusionner les résultats : la fusion consiste à rassembler dans un seul tableau trié les valeurs contenues dans les deux tableaux triés fournis en paramètre (en respectant bien sûr les répétitions éventuelles : la longueur du résultat de la fusion est la somme des longueurs des deux tableaux initiaux).

D'où le programme Python, en supposant programmée ladite fusion :

```
def tri(t):
    if len(t)<2:
        return t
    else:
        m=len(t)//2
        return fusion(tri(t[:m]),tri(t[m:]))
```

La terminaison est justifiée par la décroissance stricte de n à chaque appel récursif, la correction par récurrence **forte** sur n.
Noter que la profondeur maximale des appels récursifs est de l'ordre de $\log_2(n)$, ce qui permettra de trier de grands tableaux, même avec Python...

La fusion se prête très bien également à une programmation récursive, avec les avantages et inconvénients habituels : élégante et facile à justifier, mais gourmande en mémoire.
Voici une vision récursive de l'algorithme de fusion de deux tableaux triés $t1$ et $t2$:

- si l'un des deux tableaux est vide, renvoyer l'autre ;
- sinon renvoyer le tableau formé par la plus petite des deux valeurs $t1\,[0]$ et $t2\,[0]$, suivie de la fusion de $t1$ et $t2$, l'un des deux ayant été privé de sa première valeur déjà placée !

D'où ce programme Python séduisant :

```
def fusion(t1,t2):
    if t1==[]:
        return t2
    elif t2==[]:
        return t1
    elif t1[0]<t2[0]:
        return [t1[0]]+fusion(t1[1:],t2)
    else:
        return [t2[0]]+fusion(t1,t2[1:])
```

Notons $n_1 = len\,(t1)$ et $n_2 = len\,(t2)$La terminaison est justifiée par la décroissance stricte de $n_1 + n_2$ à chaque appel récursif, la correction par récurrence sur la valeur de cette somme. La complexité (en nombre de comparaisons d'éléments de tableau) au mieux est $\min(n_1, n_2)$, au piren$_1 + n_2$.

Mais le programme ci-dessus souffre de deux gros défauts :

- le nombre de comparaisons est bien un $O\,(n_1 + n_2)$, mais la complexité temporelle est quadratique, du fait des recopies de tableaux ;
- la profondeur des appels récursifs est au pire également de $n_1 + n_2$, ce qui peut poser problème pour de grands tableaux dans certains langages (notamment Python...).

D'où l'intérêt d'une version itérative de la fusion, utilisant un tableau local t pour stocker le résultat de la fusion de $t1$ et $t2$; je fais évoluer deux indices $i1$ et $i2$ au fur et à mesure que j'avance dans $t1$ et $t2$:

- initialiser $i1$ et $i2$ à 0, t à un tableau vide
- tant que $i1 < len\,(t1)$ et $i2 < len\,(t2)$, comparer $t1\,[i1]$ et $t2\,[i2]$, ajouter la plus petite des deux à la fin de t et incrémenter l'indice correspondant ($i1$ ou $i2$)
- à la sortie de la boucle précédente, soit $i1 = len\,(t1)$, soit $i2 = len\,(t2)$; dans le premier cas, toutes les valeurs de $t1$ ont été recopiées dans t, il n'y a plus qu'à y recopier la fin de $t2$; idem dans le second cas.

D'où ce nouveau programme Python :

```
def fusion(t1,t2):
    i1,i2,n1,n2=0,0,len(t1),len(t2)
    t=[]
    while i1<n1 and i2<n2:
        if t1[i1]<t2[i2]:
            t.append(t1[i1])
            i1+=1
        else:
            t.append(t2[i2])
            i2+=1
    if i1==n1:
        t.extend(t2[i2:])
    else:
        t.extend(t1[i1:])
    return t
```

Exercice : justifier ce programme... Sa complexité en nombre de comparaisons (mais également temporelle, globalement) est bien un $O\,(n_1 + n_2)$.

On notera les "bonnes pratiques" :

- utiliser les variables $n1$, $n2$ plutôt que de recalculer sans cesse $len\,(t1)$ et $len\,(t2)$
- utiliser `i+=1` plutôt que `i=i+1` (la première version incrémente "en place", tandis que la seconde crée une nouvelle instance de `i` avant de laisser le ramasse-miettes nettoyer l'ancienne...)
- utiliser `t.append(x)`plutôt que `t=t+[x]` (la première version "ajoute" `x` à la fin de `t`, tandis que la seconde recopie toutes les valeurs de `t` dans une nouvelle instance de `t` avant d'y adjoindre `x`...)
- de même, utiliser `t.extend(u)` plutôt que `t=t+u`.

2) Complexité du tri par fusion (en nombre de comparaisons d'éléments de tableau)

Cet algorithme est typique du paradigme "diviser pour régner" ; ici, la partition ne nécessite aucune comparaison et nous avons vu que l'on peut implémenter la fusion de deux tableaux contenant en tout n cases en $\Theta\,(n)$. Ainsi, le coût $C\,(n)$ du tri d'un tableau de taille n vérifie la relation de récurrence :

$$C\,(n) = C\,(\lfloor n/2 \rfloor) + C\,(\lceil n/2 \rceil) + \Theta\,(n)$$

d'où l'on peut déduire :

$$C\,(n) = \Theta\,(n \log_2 n)\,.$$

NB : plus précisément, $C\,(n) = C\,(\lfloor n/2 \rfloor) + C\,(\lceil n/2 \rceil) + \lambda.n$ donne $C\,(n) \sim \lambda n \log_2 n$.

3) Version procédurale

Le lecteur attentif aura remarqué que la fonction présentée au § **1)** effectue des copies des deux "moitiés" du tableau au moment des appels récursifs. Cela est moins gênant que pour la recherche dichotomique, car la complexité temporelle globale vérifie une relation de récurrence du même type que ci-dessus, puisque lesdites copies se font en temps linéaire.

Toutefois, on peut éviter ces recopiages en optant pour des **procédures** qui modifient le tableau à trier. Il reste recommandé d'utiliser un tableau auxiliaire pour la fusion, mais un seul tableau doit suffire, donc pas d'inflation de la complexité spatiale. Cf. le TD 2.

III - Tri rapide (ou "quicksort", par C.A.R. Hoare – 1960)

1) Principe et implémentation

L'idée consiste à partitionner d'abord le tableau t à trier autour d'un *pivot* : on choisit l'une des valeurs du tableau (ledit pivot), par exemple $t\,[0]$ et l'on construit deux tableaux avec les $t\,[i]$ pour $i > 0$ (où l'on peut retrouver le pivot si sa valeur figure pour plusieurs indices...) :

- le premier $t1$ avec les valeurs correspondant aux indices i tels que $t\,[i] < pivot$
- le second $t2$ avec les valeurs correspondant aux indices i tels que $t\,[i] \geq pivot$

Il n'y a plus qu'à trier (récursivement !) $t1$ et $t2$ et à renvoyer les valeurs triées de $t1$, suivies de la valeur du pivot et des valeurs triées de $t2$! On obtient ainsi les valeurs de t triées (preuve immédiate par récurrence forte).

L'algorithme se termine bien puisque les longueurs des tableaux $t1$ et $t2$ transmis lors des appels récursifs sont strictement inférieures à la longueur du paramètre initial t, par construction. Notons toutefois que — contrairement à la situation du tri par fusion — $t1$ et $t2$ peuvent être de tailles déséquilibrées, ce qui fait que la complexité de ce tri peut-être quadratique dans le pire des cas (voir ci-dessous).

Mais sa complexité en moyenne est en $n \ln n$ et sa mise en œuvre s'avère très rapide, d'où son nom !

L'implémentation en Python est naturelle :

```
def quicksort(t):
    if t == []:
        return []
    else:
        pivot = t[0]
        t1 = []
        t2 = []
        for x in t[1:]:
            if x<pivot:
                t1.append(x)
            else:
                t2.append(x)
        return quicksort(t1)+[pivot]+quicksort(t2)
```

Où l'on voit que la partition est facile à faire avec les "tableaux-listes" de Python.

La programmation dans des langages moins sophistiqués (et parfois plus efficaces) se fait classiquement "en place", c'est-à-dire en permutant les éléments du tableau t tout en déterminant la place définitive du pivot. Voici une implémentation possible, où la fonction `partition` reçoit comme paramètres deux indices g et d, renvoie un indice p et réorganise la tranche $t[g:d]$ de sorte que les post-conditions suivantes soit satisfaites :

- pour i tel que $g \leq i < p$, $t[i] < pivot$
- $t[p] = pivot$
- pour i tel que $p < i < d$, $t[i] \geq pivot$

Pour cela, je procède de la façon suivante :

- p est initialisé à g et $pivot$ reçoit la valeur $t[g]$
- pour k allant de $g+1$ à $d-1$ (le reste de la tranche), si $t[k] < pivot$ j'incrémente p et j'échange $t[p]$ et $t[k]$
- j'échange $t[g]$ et $t[p]$ pour mettre le pivot à sa place et je renvoie la valeur de p qui indique où se fait la partition

```
def partition(t,g,d):
    p=g
    pivot=t[g]
    for k in range(g+1,d):
        if t[k]<pivot:
            p+=1
            t[p],t[k]=t[k],t[p]
    t[p],t[g]=t[g],t[p]
    return p
```

```
def quicksort(t):
    def tri(g,d):
        if g<d:
            p=partition(t,g,d)
            tri(g,p)
            tri(p+1,d)

    tri(0,len(t))
```

La fonction `tri` se justifie facilement par récurrence forte, en admettant la correction de la fonction `partition`...

Exercice : justifier la fonction `partition` !

2) Complexité du tri rapide (en nombre de comparaisons d'éléments de tableau)

Ici, la "fusion" ne coûte rien, la partition d'un tableau de n valeurs se fait au prix de $n-1$ comparaisons. Mais nous n'avons pas de relation de récurrence du type "diviser pour régner", car le partage ne se fait *a priori* pas toujours en deux sous-tableaux de tailles égales.

Ainsi, dans le pire des cas (par exemple lorsque le tableau est initialement trié ! Alors $t1$ est toujours vide et $t2$ de taille $n-1$), le coût $C(n)$ vérifie

$$C(0) = C(1) = 0 \quad \text{et} \quad \forall n \geq 2 \quad C(n) = n - 1 + C(n-1) \quad \text{d'où} \quad C(n) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}.$$

Pour ce qui est de la complexité en moyenne, encore notée $C(n)$, en supposant — à chaque étape — les différentes tailles de $t1$ et $t2$ comme équiprobables, outre $C(0) = C(1) = 0$, j'obtiens successivement pour $n \geq 2$:

$$C(n) = n - 1 + \frac{1}{n}\sum_{p=0}^{n-1}\big(C(p) + C(n-1-p)\big) = n - 1 + \frac{2}{n}\sum_{k=1}^{n-1} C(k) \ .$$

$$nC(n) = n(n-1) + 2\sum_{k=1}^{n-1} C(k) \ ; \quad (n+1)C(n+1) = n(n+1) + 2\sum_{k=1}^{n} C(k) \ ;$$

$$(n+1)C(n+1) = (n+2)C(n) + 2n \ ; \quad nC(n) = (n+1)C(n-1) + 2(n-1) \ ;$$

et en divisant par $n(n+1)$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + 2\frac{n-1}{n(n+1)} \ ;$$

d'où par récurrence immédiate

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

On peut en déduire que $C(n) \sim 2n\ln n = (2\ln 2)\, n\log_2 n$.

Remarque : le lecteur avisé aura compris que le choix du pivot est déterminant ; le choix de la première valeur (utilisé ci-dessus) s'avérant catastrophique pour un tableau initialement (presque) trié. D'autres stratégies sont possibles mais leur étude dépasse le cadre du programme officiel... Cf. le TD 2.

3) Application : calcul de la médiane d'une série statistique

Supposons des valeurs numériques stockées dans un tableau t de taille n (indexé de 0 à $n-1$).
Nous cherchons un indice i tel que :

- au moins la moitié des valeurs sont inférieures ou égales à $t[i]$
- au moins la moitié des valeurs sont supérieures ou égales à $t[i]$

On dit dans ce cas que $m = t[i]$ est *une médiane* de la série statistique (il existe d'autres définitions de la médiane permettant d'assurer son unicité dans le cas d'un nombre pair de valeurs...).

Un algorithme évident pour déterminer une telle médiane est de trier le tableau t et de renvoyer $t[n//2]$! Mais cet algorithme a le même coût que le tri du tableau, donc en moyenne au mieux de l'ordre de $n\ln n$...

L'idée de partition du tri rapide permet d'élaborer un algorithme linéaire en moyenne (sans trier le tableau !).

Tant qu'à faire, donnons un algorithme recevant un tableau t et un entier k (avec la pré-condition $0 \leq k < len(t)$) et renvoyant la valeur d'indice k dans la liste triée des valeurs initialement stockées dans t.

Cet algorithme récursif utilise la fonction `partition` du § **1)** ; pour déterminer $k\_ieme(g,d)$, valeur d'indice k dans la liste triée des valeurs initialement stockées dans $t[g:d]$ (pré-condition $g \leq k < d$!) :

- p reçoit l'indice renvoyé par `partition`(g,d), dont l'exécution réorganise les valeurs de la tranche $t[g:d]$
- si $p = k$, renvoyer $t[p]$; si $p > k$, renvoyer $k\_ieme(g,p)$ et si $p < k$, renvoyer $k\_ieme(p+1,d)$ (appels récursifs)

Suivant ce principe, on ne réorganise que certaines tranches de t, sans les trier complètement, mais en préservant le fait que la valeur cherchée se trouve dans la tranche $t[g:d]$ (d'où la justification par récurrence forte sur la valeur de $d-g$...).

Voici une implémentation en Python :

```
def quick_ieme(t,k):
    def k_ieme(g,d):
        p=partition(t,g,d)
        if p==k:
            return t[p]
        elif p>k:
            return k_ieme(g,p)
        else:
            return k_ieme(p+1,d)

    return k_ieme(0,len(t))
```

Comme pour le tri rapide, la complexité au pire est quadratique (par exemple si l'on cherche la plus grande valeur d'un tableau initialement trié !).
Toutefois on peut montrer que la complexité en moyenne est linéaire.

Pour cela, je note $T(n)$ le maximum, pour $k = g, \dots, d-1$, des nombres moyens de comparaisons effectuées lors de l'appel de $k\_ieme(g,d)$, lorsque $d-g=n$ (nombre d'éléments traités).

Un appel à $partition(t,g,d)$ effectue $d-g-1$ comparaisons, soit $n-1$ comparaisons ; le nombre de comparaisons entraînées par l'appel de $k\_ieme(g,d)$ est par conséquent au plus égal à $n-1\dots$

- $\dots$ plus 0 si l'on trouve le k-ième élément dès le premier appel ;
- $\dots$ plus $T(d-p-1)$ si la fonction de partition renvoie $p<k$ (appel de $k\_ieme(p+1,d)$) ;
- $\dots$ plus $T(p-g)$ si elle renvoie $p>k$ (appel de $k\_ieme(g,p)$).

En supposant ces différents cas équiprobables, j'obtiens pour majorant du nombre moyen de comparaisons nécessitées par l'appel de $k\_ieme(g,d)$, en réindexant convenablement :

$$n-1+\frac{1}{n}\left(\sum_{p=g}^{k-1}T(d-p-1)+\sum_{p=k+1}^{d-1}T(p-g)\right)\leq n-1+\max_{g\leq k<d}\left[\frac{1}{n}\left(\sum_{i=d-k}^{d-g-1}T(i)+\sum_{i=k-g+1}^{d-g-1}T(i)\right)\right],$$

or ce dernier majorant est indépendant de k, c'est donc un majorant de $T(n)$.

J'en déduis que $T(n)\leq 4n$ par récurrence forte sur n : en effet, $T(1)=0\leq 4$ et, si je suppose $n\geq 2$ tel que, pour tout $j<n$, $T(j)\leq 4j$, alors l'expression entre crochets ci-dessus est majorée par (en posant $r=k-g\in[0,n[$) :

$$\begin{aligned}\frac{4}{n}\left(\sum_{i=n-r}^{n-1}i+\sum_{i=r+1}^{n-1}i\right) &= \frac{4}{n}\left((n-1)\,n-\frac{(n-r-1)\,(n-r)}{2}-\frac{r\,(r+1)}{2}\right)\\ &= 4n-4-\frac{2}{n}\left(n^2-n+2r\,(r+1-n)\right)\\ &= 2n-2+\frac{4r\,(n-1-r)}{n},\end{aligned}$$

Or la fonction $r\mapsto r\,(S-r)$ atteint son maximum en $\dfrac{S}{2}$, ce maximum valant $\left(\dfrac{S}{2}\right)^2$ (le produit de deux facteurs de somme constante est maximum lorsque les deux facteurs sont égaux).

Il en résulte :

$$T(n)\leq 2n-2+\frac{4}{n}\frac{(n-1)^2}{4}=2n-2+n-2+\frac{1}{n}\leq 4n,$$

ce qui achève la démonstration par récurrence.

En conclusion :

$$\boxed{\forall n\geq 1\quad T(n)\leq 4n.}$$