INFORMATIQUE

ALGORITHMES DE TRI

Les objectifs de cette séquence sont :

1- D'étudier quelques algorithmes de tri qui s'appliquent à des tableaux ou à des listes, tris quadratiques dans un premier temps puis quasi-linéaire dans un second temps.
2- D'estimer leur coût calculatoire afin de mener une approche comparative.

I Notion de coût des algorithmes

Le concept de coût associé à un algorithme formalise la notion intuitive d'efficacité d'un programme, c'est à dire sa capacité à fournir le résultat escompté dans un temps minimal, on parle également de performances du programme. Du point de vue pratique, l'analyse du coût permet de déterminer quelles seront les ressources en temps de calcul, en mémoire et éventuellement en entrées/sorties nécessaires pour exécuter le calcul associé à l'algorithme.
Dans le cadre d'une approche simplifiée et, dans une certaine mesure réductrice, la comparaison des algorithmes de tri se limitera principalement à l'étude de leur **complexité temporelle**. Elle est toutefois difficile à estimer car elle dépend de très nombreux paramètres, non seulement de la dimension de l'objet à traiter mais aussi de la structure des données.
Différents types de complexité sont pertinents.
La complexité dans le « pire cas ». On estime le nombre d'opérations effectué dans un contexte du traitement tel que le nombre d'opérations est maximal.
La complexité en moyenne. Cette notion n'est applicable que si l'on dispose d'une bonne connaissance sur la statistique de la distribution des données.
La complexité « meilleur cas ». On mesure le nombre d'opérations avec les données qui mènent à un nombre minimal d'opérations.

La notion de complexité peut aussi se définir par rapport à la consommation d'espace de l'algorithme (quantité de mémoire occupée). On parle alors de complexité spatiale.

Soit un algorithme de tri $\mathcal{A}$, permettant de trier un tableau T par valeurs croissantes, par exemple. La complexité, dans le pire des cas, permet de fixer une borne supérieure du nombre d'opérations qui seront nécessaires pour trier un tableau de n éléments. Elle est définie par :

$$\mathcal{C}_{\mathcal{A},pire}(n) = \max_{T \text{ tableau de taille } n} \mathcal{C}(\mathcal{A}, T)$$

Dans cette expression, $\mathcal{C}(\mathcal{A}, T)$ représente un temps d'exécution du traitement de l'algorithme $\mathcal{A}$ pour trier le tableau T et pour une configuration précise (conditions logiques, tests, …). Ainsi, $\mathcal{C}_{\mathcal{A},pire}(n)$ est la complexité temporelle dans une situation de pire cas.
Il existe d'autres approches qui permettent d'aborder la complexité calculatoire.
La complexité en moyenne est le nombre d'opérations élémentaires effectuées en moyenne pour trier un tableau de n éléments. L'étude de la complexité en moyenne est en générale difficile et requiert une approche probabiliste. En pratique, on suppose que les éléments du tableau sont distincts et que les $n!$ permutations décrivant les positions relatives des éléments dans le tableau d'entrée sont équiprobables. En résumé, tout se passe comme si l'on considérait uniquement des tableaux de la forme $[\sigma(1), \dots, \sigma(n)]$ où σ est une permutation de $\mathfrak{S}_n$. La complexité moyenne de l'algorithme de tri $\mathcal{A}$ sur les tableaux de taille n est alors donnée par l'expression de la moyenne pour une loi uniforme :

$$\mathcal{C}_{\mathcal{A},moy}(n) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \mathcal{C}(\mathcal{A}, \sigma)$$

Dans cette expression $\mathcal{C}(\mathcal{A}, \sigma)$ est le nombre d'opérations nécessaires pour trier le tableau $[\sigma(1), \dots, \sigma(n)]$ avec l'algorithme $\mathcal{A}$. La complexité en moyenne est une estimation bien plus satisfaisante que la complexité dans le pire cas.
La complexité dans le meilleur des cas n'est pas la plus pertinente, mais permet de distinguer deux algorithmes égaux par ailleurs. A l'opposé de la complexité dans le pire cas, elle consiste à regarder les situations favorables à l'algorithme $\mathcal{A}$.

$$\mathcal{C}_{\mathcal{A},meilleur}(n) = \min_{T \text{ tableau de taille } n} \mathcal{C}(\mathcal{A}, T)$$

Outre la complexité temporelle de l'algorithme, certaines propriétés sont appréciables parmi lesquelles le caractère « en place » du tri. Dans ce cas, l'algorithme modifie directement la structure qu'il est en train de trier. Il ne requiert donc qu'un espace en mémoire constant en plus du tableau d'entrée pour le trier.

II Algorithmes de tri

Un algorithme de tri est un algorithme qui permet d'organiser un tableau homogène T d'éléments selon un ordre fixé. A priori, l'algorithme de tri ne retourne rien : en sortie de fonction, le tableau passé en entrée est trié.
En pratique, les ordres les plus utilisés sont l'ordre numérique (sur les réels ou les entiers naturels et relatifs) et l'ordre lexicographique (pour traiter des chaînes de caractères ou des n-uplets). Par la suite, on ne se préoccupera pas de la relation d'ordre utilisée, celle-ci sera vue comme une « boite noire », comparant deux éléments quelconques d'un ensemble E.
Ainsi, un algorithme générique de tri possède la propriété fondamentale suivante :

Entrée : Un tableau T dont les éléments sont à valeurs dans un ensemble ordonné.

Sortie : Le même tableau T trié (dans l'ordre croissant par exemple)

2-1 Tris quadratiques en $\mathcal{O}(n^2)$

On détaille ici les algorithmes de tris « naïfs » les plus classiques. Ceux-ci sont quadratiques et sont donc inefficaces pour de grands tableaux. On leur préférera l'un des algorithmes de la section suivante dès que le nombre d'éléments à trier dépasse quelques centaines.

2-1-1 Tri par sélection

Ce tri particulièrement simple est probablement la méthode à laquelle on pense lorsqu'on écrit un algorithme de tri.
Principe : supposons que le tableau de taille n est déjà en partie trié avec ses i premiers éléments à leur place définitive. On sélectionne le plus petit des $n - i$ éléments restants, qu'on amène en position $i + 1$. Le tableau a alors ses $i + 1$ premiers éléments à leur position définitive. Itérer ce procédé $n - 1$ fois suffit pour trier le tableau.

Algorithme : Tri par sélection
{Cet algorithme permet de trier un tableau T par une méthode de tri par sélection}
Argument passé en paramètre
Un tableau T de n éléments à trier : $T = [e_1, e_2, \dots, e_n]$

	Pseudo-code	$\times\ fois$	commentaires
2	**Début**		
	pour $i = 1$ à $n - 1$ **faire** :	$n-1$	Parcours des éléments de T triès jusqu'à i
	$Mem \leftarrow i$	$n-1$	Index i mémorisé de la fin de liste triée
3	**pour** $j = i + 1$ à n **faire** :	$\sum_{i=0}^{n-2}(n-i-1) = \sum_{i=1}^{n-1} j = \frac{n(n-1)}{2}$	Parcours des $n - i - 1$ éléments non triés
4	**si** $T[j] < T[Mem]$	$\frac{n(n-1)}{2}$	Test du plus petit élément ?
5	$Mem \leftarrow j$		Sauve de l'index j si test précédent ok
	fin		
6	$T[i], T[Mem] \leftarrow T[Mem], T[i]$	$n-1$	Echange des éléments
	fin		
7	**fin**		

La charge de calcul est donc quadratique, elle croît asymptotiquement comme n^2, le carré du nombre d'éléments du tableau T soit un coût en $\mathcal{O}(n^2)$.

2-1-2 Tri par insertion

Principe : supposons que la partition $T[0: i-1]$ du tableau T est triée. On s'intéresse à l'élément $T[i]$, que l'on va insérer dans les éléments déjà triés de la partition $T[0: i-1]$ de sorte que le tableau $T[0: i]$ soit trié. Pour se faire, à l'itérée i la valeur $T[i]$ est affectée à une variable Mem afin de permettre un décalage à droite des éléments de $T[0: i-1]$ jusque ce que la condition $T[j-1] \geq T[i]$, $j = 1, \dots, i$ alors $T[j-1] \leftarrow T[i]$.

Algorithme : Tri par insertion
{Cet algorithme permet de trier un tableau T par une méthode de tri par insertion}
Argument passé en paramètre
Un tableau T de n éléments à trier : $T = [e_1, e_2, \dots, e_n]$

	Pseudo-code	$\times fois$	**commentaires**
2	**Début**		
	pour $i = 2$ à n **faire**	$n-1$	Parcours des éléments de T
	$Mem \leftarrow T[i]$		Affectation
	$j \leftarrow i-1$		
3	**Tant que** $j > 0$ **et** $T[j] > Mem$ **faire :**	$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$	Parcours des $n-i-1$ éléments triés
4	$T[j+1] \leftarrow T[j]$	$\frac{n(n-1)}{2}$	Décalage d'un pas à droite
5	$j \leftarrow j-1$		j est décrémenté d'une unité
	fin		
6	$T[j+1] \leftarrow Mem$	$n-1$	Insertion de l'élément Mem à l'index $j+1$
	fin		
7	**fin**		

La charge de calcul est donc quadratique, elle croît asymptotiquement comme n^2, le carré du nombre d'éléments du tableau T soit un coût en $\mathcal{O}(n^2)$.

Complexité : Il suffit de dénombrer séparément les comparaisons et les affectations. Deux situations très différentes peuvent être envisagées.

Dans le meilleur des cas, la condition de la boucle *tant que* n'est jamais vérifiée car $T[j] < Mem$: le tableau est trié. Dans ce cas, $n-1$ comparaison et $2 \times (n-1)$ affectations sont effectuées en tout. Ainsi la complexité dans le meilleur cas est linéaire. Dans le pire cas, la condition logique de la boucle *tant que* est toujours vérifiée jusqu'à $j = 0$. Chaque élément traité est placé en début de liste après un décalage à droite complet de la liste déjà triée. Cette situation correspond à une liste ordonnée par ordre décroissant des valeurs :

La complexité en pire cas est développée dans le tableau ci-dessus.

2-1-3 Tri à bulles

L'algorithme parcourt le tableau et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Ainsi, à la fin du premier passage, la plus grande valeur remonte en fin de liste c'est-à-dire à l'index n. Si au moins un échange a eu lieu pendant le parcours, l'algorithme procède à un nouveau traitement mais sur les $n-1$ premiers éléments. Le tableau est trié et l'algorithme s'arrête lorsqu'il n'y a plus d'échange pendant un parcours.

Algorithme : Tri à bulles
{Cet algorithme permet de trier un tableau T par une méthode du tri à bulles}
Argument passé en paramètre
Un tableau T de n éléments à trier : $T = [e_1, e_2, \dots, e_n]$

	Pseudo-code	$\times\, fois$	**commentaires**
2	**Début**		
	TheEnd ← True		
	Tant que TheEnd **faire :**	$n-1$	Parcours de T tant que la condition est vraie
	TheEnd ← False		Sortie de la boucle par défaut
	Pour $i = 1$ **à** $n-1$ **faire :**	$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$	
3	**si** $T[i] > T[i+1]$		
4	$T[i], T[i+1] \leftarrow T[i+1], T[i]$	$\frac{n(n-1)}{2}$	Tri des deux éléments et, …
5	TheEnd ← True		le traitement doit donc continuer
	fin		
6	$n \leftarrow n-1$	$n-1$	Un élément en moins à trier
	fin		
7	**fin**		

La charge de calcul est donc quadratique, elle croît asymptotiquement comme n^2, le carré du nombre d'éléments du tableau T soit un coût en $\mathcal{O}(n^2)$.

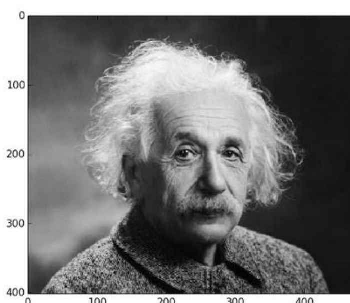
APPLICATION

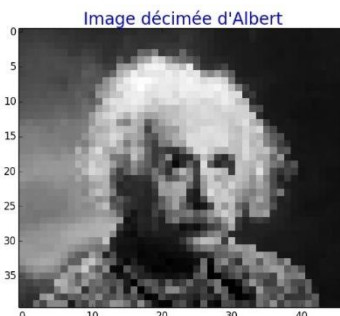
Q1. Pour chacun des pseudocodes présentés, écrivez les fonctions associées $triselection(T)$, $triinsertion(T)$ et $tribulles(T)$. Ces fonctions ont pour argument d'entrée un tableau T constitué de n éléments de type entier et retournent ce tableau passé en entrée trié.

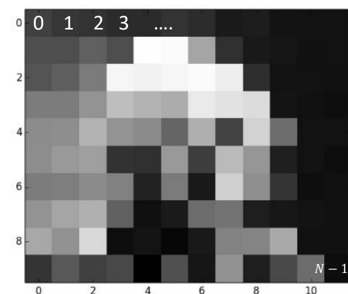
Comment observer le traitement en temps réel de ces algorithmes ?

L'algorithme peut être vu comme une boîte noire, il possède une ou plusieurs entrées et retourne une ou plusieurs sorties qui sont le résultat du traitement. Quand est-il alors de l'évolution de son état interne ?

Concernant le tri et afin de visualiser cette évolution, le principe proposé est d'indexer les N pixels de l'image par la suite des entiers naturels $0, 1, \dots, N-1$ comme illustré ci-dessous sur l'image décimée d'Albert.







Cette liste de nombres $L = [0, 1, \dots, N-1]$ est initialement mélangée et la reconstruction de l'image n'est pas fidèle à l'image initiale. Ce n'est donc pas l'image qui est « mélangée » mais les index associés à chacun de ses pixels que votre algorithme de tri va réordonnée par ordre croissant.

Pour rendre compte du déroulé de votre algorithme, il doit est intégré à un environnement graphique proposé qui a pour nom animationV1.py dont le traitement principal est représenté ci-dessous.

L, k et N sont des variables globales : L est la liste des index de l'image qui est à trier, N sa dimension et k, l'index qui correspond à la reprise du traitement après la reconstruction de l'image et de son plot.

Il faut donc ajuster votre algorithme à ce contexte, la variable k est un point un peu délicat…

```
#-----------------------------------------------
ImTraite=plt.imshow(Im,cmap='gray',interpolation='none')# définition des propriétés du plot pour la fonction animation
k=0                                                      # permet de reprendre le tri après le plot
def updatefig(*args):
    global ImTraite, L, k, N
#---algorithme de Tri à intégrer-----------------

        Vous incrustez votre algorithme ICI

#-----------------------------------------------
        if (i%50==0)or(i==(N-1)):                        # réduction du coût : affichage 1:50
            Im=ImOUT[L]                                  # réaffectation des index aux pixels
            Im=Im.reshape(dimim[0],dimim[1])             # reconstruction d'une image de dimension dimim[0],dimim[1]
            ImTraite.set_array(Im)                       # plot de l'image
            k=i+1                                        # mise à jour de k pour la suite du tri de L
            return ImTraite,

ani = animation.FuncAnimation(fig, updatefig)
plt.show()
#-----------------------------------------------
```

APPLICATION

Q2. Intégrer vos algorithmes dans le template « animationV1.py », contempler !

2-2 Tests et validation des algorithmes de tri

Pour évaluer les performances d'un algorithme, il faut générer un jeu de tests profilé au type d'évaluation souhaité, soit :

1- L'estimation comparée du coût calculatoire des algorithmes par insertion, sélection et à bulles.

2- L'estimation de la densité de probabilité associé à la variable aléatoire temps d'exécution.

Le résultat attendu est représenté ci-dessous.

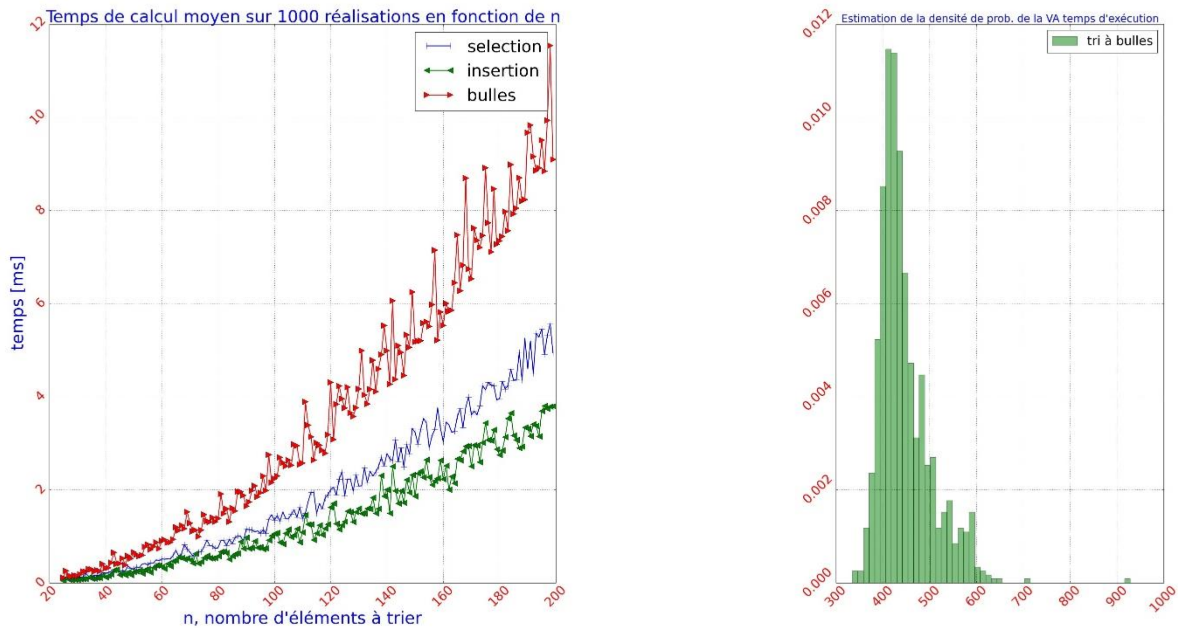


Figure 1 : A gauche, évolution du temps d'exécution comparé pour les 3 algorithmes et à droite, estimation de la loi associée du temps d'exécution pour le tri à bulle.

La figure 1 de gauche représente l'évolution du coût calculatoire en fonction du nombre d'éléments à trier. Afin de réduire la variance de l'estimation du temps d'exécution, une même liste est mélangée puis triée 100 fois, … Le résultat retenu est la moyenne de ces 100 réalisations. L'histogramme normalisé représenté sur le plot de droite est une estimation de la densité de probabilité obtenue à partir de 2000 réalisations du tri à bulles d'une liste d'entiers de 1000 éléments générée par le script suivant avec $p = 100\ 000$:

$$T = [randint(-p, p)\ for\ k\ in\ range(n)]$$

Ce résultat permet d'estimer le coût calculatoire en moyenne statistique alors qu'il est étudié en valeur sup.

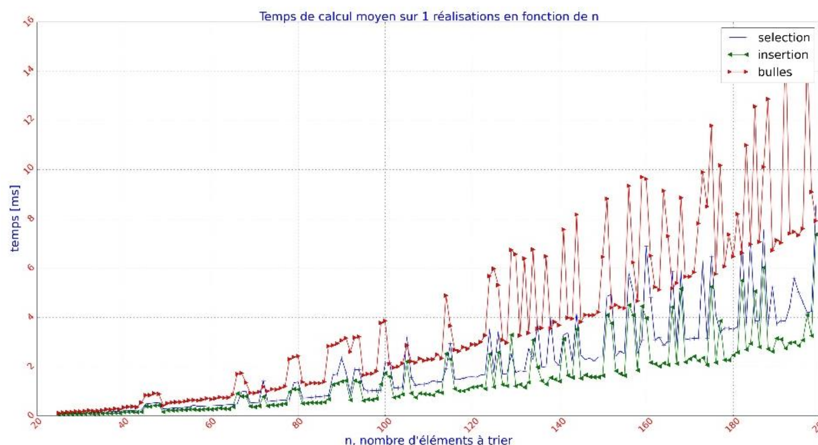


Figure 2 : Temps d'exécution comparé pour obtenu pour une réalisation

Rappelons que le temps d'exécution est une variable aléatoire qui prend ses valeurs sur un support borné. La figure 2 ci-dessus est le résultat obtenu pour une seule réalisation par liste à trier.

APPLICATION

Q3. Ecrivez une fonction qui permet d'estimer le coût calculatoire de chacun des algorithmes et de les comparer sur un même plot comme représenté sur la figure 2.
Attention aux effets de bord !

Afin de réduire la variance de l'estimation, il convient de moyenner plusieurs réalisations obtenues pour une même liste mélangée.

APPLICATION

Q4. Modifiez l'algorithme précédent afin de répondre à cette exigence.

Bonus : On s'intéresse désormais à l'estimation de la loi associée à la variable aléatoire (VA) « temps d'exécution » de l'algorithme de tri à bulles.

APPLICATION

Q5. Proposez un algorithme qui permet d'estimer la loi de cette VA à partir de m réalisations.

Le script ci-dessous permet d'obtenir une représentation des résultats dans un subplot identique à la **figure 1**.

APPLICATION

Q6. Décoder ce script afin de l'adapter, représenter vos résultats puis conclure.

```
subplot(1,2,1)
# plot des résultats
plt.plot(Lref, 1000*tmoy[0, :], label='selection',marker='+',markersize=15,linewidth=2)
plt.plot(Lref, 1000*tmoy[1, :], label='insertion',marker='<',markersize=15,linewidth=2)
plt.plot(Lref, 1000*tmoy[2, :], label='bulles',marker='>',markersize=15,linewidth=2)

# renseignement des axes
titre="Temps de calcul moyen sur "+str(m)+" réalisations en fonction de n"
plt.title(titre,fontsize=40,color = 'blue')
plt.xlabel("n, nombre d'éléments à trier",fontsize=40,color = 'blue')
plt.ylabel("temps [ms]",fontsize=40,color = 'blue')

# modification de la police des axes
plt.xticks(fontsize=30,rotation = 45,color = 'red')
plt.yticks(fontsize=30,rotation = 45,color = 'red')
plt.grid(True)
plt.legend(fontsize=40)                    # affichage de la légende

subplot(1,3,3)
# the histogram of the data
num_bins = 50
n, bins, patches = plt.hist(1000*texe.T, num_bins, normed=1, facecolor='green', alpha=0.5,label='tri à bulles')

# modification de la police des axes
plt.xticks(fontsize=30,rotation = 45,color = 'red')
plt.yticks(fontsize=30,rotation = 45,color = 'red')
plt.title("Estimation de la densité de prob. de la VA temps d'exécution",fontsize=25,color = 'blue')
plt.grid(True)
plt.legend(fontsize=30)
```