1NSI	Algorithmique Tris	TP – **Les algorithmes de tri**

Objectifs :

- ⇨ Savoir qu'il existe de nombreux algorithmes de tri
- ⇨ Avoir une idée de ce qui peut différencier les différents algorithmes de tri
- ⇨ Programmer quelques algorithmes de tri simples

Comment trier ses cartes ?

I – Les algorithmes de tri

1) Qu'est-ce qu'un algorithme de tri ?

Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité. Lorsque le type de ces éléments possède un ordre (comme les nombres ou les lettres), on peut les ranger en ordre croissant ou décroissant.
Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant.
Dans ce cours on ne fera que des tris en ordre croissant.
Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur.

2) Les différentes caractéristiques d'un algorithme de tri

a. Complexité

La **complexité temporelle** d'un algorithme est une estimation de son temps d'exécution, exprimée comme fonction de la taille de l'entrée. Le temps d'exécution brut n'est pas une grandeur fiable car il dépend de la puissance de la machine sur laquelle on exécute l'algorithme. Pour mesurer la vitesse on compte le nombre d'étapes de calcul avant d'arriver à un résultat. La durée d'exécution d'une étape de calcul dépend de la machine qui l'exécute tandis que le nombre d'étapes à faire ne dépend que de l'algorithme (et éventuellement des données en entrée).
On calcule presque toujours de la complexité *dans le pire des cas*, c'est-à-dire le cas où la valeur des données à traiter entraine le temps de calcul le plus long (ex : tri d'un tableau complétement en désordre au lieu d'un tableau presque déjà trié). On note alors cette complexité dans le pire des cas avec un O(*fonction de n*). Le nombre d'action a réaliser pour parvenir au bout de l'algorithme est alors proportionnel à ce O().

Liste de complexités en temps classiques

Temps de calcul ↓

Nom	Complexité	Exemple de temps de calcul	Exemple d'algorithmes
Constant	O(1)	10	Algo simple ne dépendant pas de la taille des données (ex : `x = len(tab)`)
Logarithmique	O(log n)	log n, $\log(n^2)$	Recherche dichotomique
Linéaire	O(n)	n	Recherche séquentielle, algorithme simple de recherche du plus petit élément d'un tableau
Pseudo-linéaire	O(n log n)	n log n, log n!	Tri fusion
Quadratique	$O(n^2)$	n^2	Tri par insertion
Cubique	$O(n^3)$	n^3	Algorithme naïf de multiplication matricielle.
Exponentiel	$O(a^n)$	1.1^n, 10^n	Algorithme en force brute, par exemple pour le problème du voyageur de commerce[1] ou la résolution d'un sudoku.

[1] Le problème du voyageur de commerce est un problème d'optimisation qui, étant donné une liste de villes, et des distances entre toutes les paires de villes, détermine un plus court circuit qui visite chaque ville une et une seule fois.

Dans les cas des algorithmes polynomiaux ($O(n^2)$, $O(n^3)$, …) le temps de calcul peut augmenter très vite avec la taille des données. Pour les algorithmes exponentiels, l'augmentation est tellement rapide qu'elle rend impossible les calculs sur des tailles de données vraiment importante ($n > 10^3$).

Nous pourrons donc utiliser la complexité des algorithmes de tri pour comparer leur vitesse d'exécution qui peut être un paramètre important dans le choix d'un algorithme.

b. Complexité spatiale

La complexité spatiale étudie la quantité de mémoire dont un algorithme a besoin au cours de son exécution. Pour comparer les algorithmes de tri nous nous contenterons de déterminer si l'algorithme est ***en place*** ou pas.
Un algorithme de tri est dit « en place » s'il tri les données directement dans le tableau d'origine et ne nécessite pas de mémoire supplémentaire. S'il doit faire appel à des tableaux auxiliaires (qui recopient tout ou partie du tableau d'origine à trier), il n'est pas en place.

c. Stabilité

Un algorithme de tri est dit « stable » s'il préserve l'ordre relatif des données. Cette « stabilité » se voit si les données à trier ont plusieurs caractéristiques et que l'on ne trie que selon l'une d'elle.

Exemple :

	Algorithme stable	Algorithme non stable
Tableau avant le tri	7 2 8 6 2 3 5 8	
Tableau après le tri	2 2 3 5 6 7 8 8	2 2 3 5 6 7 8 8
	L'ordre relatif est préservé : le 2 bleu reste avant le deux vert et le 8 rouge avant le 8 bleu	L'ordre relatif n'est pas préservé : le 2 bleu reste bien avant le deux vert, mais le 8 rouge se retrouve après le 8 bleu

Le caractère stable d'un algorithme de tri n'a d'importance que dans certaines situations[2].

d. Simplicité

Une caractéristique non quantifiable, mais importante d'un algorithme est sa simplicité de compréhension et de programmation. Si les algorithmes les plus simples ne sont pas les plus rapides, ils ont au moins l'avantage d'être programmés facilement avec un risque de bug moindre. Cela peut donc être un élément à prendre en compte.

II - La procédure échanger

La plupart des algorithmes de tri utilisent une procédure qui permet d'échanger (de permuter) la valeur de deux cases d'un même tableau. L'algorithme de la procédure `echanger` est le suivant :

```
Procédure echanger(t : tableau, i : indice du premier nombre, j : indice du
second nombre)
Début
      temp←t[i]
      t[i]←t[j]
      t[j]←temp
Fin
```

[2] On a vu lorsqu'on a manipulé les données en table que pour trier selon plusieurs critères, on pouvait appeler plusieurs fois l'algorithme de tris (en changeant la clé de tri à chaque fois), mais cela ne peut fonctionner que si l'algorithme de tri est stable car sinon il mélangerait le résultat du tri précédant.

Question 1 : La procédure « echanger »
Ouvrir le fichier procedure_echanger.py et programmer la procédure `echanger`. Exécuter le programme et vérifier son bon fonctionnement.

Remarques :

- Le tableau est effectivement modifié par la procédure car on lui fournit l'adresse du tableau (passage *par référence* pour les listes en python) et non une copie des valeurs. Si on échangeait directement les valeurs, cela ne fonctionnerait pas car la procédure recevrait une copie des variables (passage *par valeur* des entiers en python) et seules les copies seraient échangées. Voir le programme « procedure_echanger_par valeur.py » à exécuter en pas à pas pour comprendre mieux les mécanismes.
- Il est nécessaire de passer par une variable temporaire pour échanger 2 variables car écrire l'une dans l'autre efface le contenu de la première. Il existe cependant une façon de faire typiquement « pythonique » qui utilise les tuples : `(a, b) = (b,a)` (ou même `a,b = b,a` car cette notation implique que a,b forment un tuple et python accepte cette notation non-explicite). On peut alors faire la permutation sans passer par une variable temporaire.

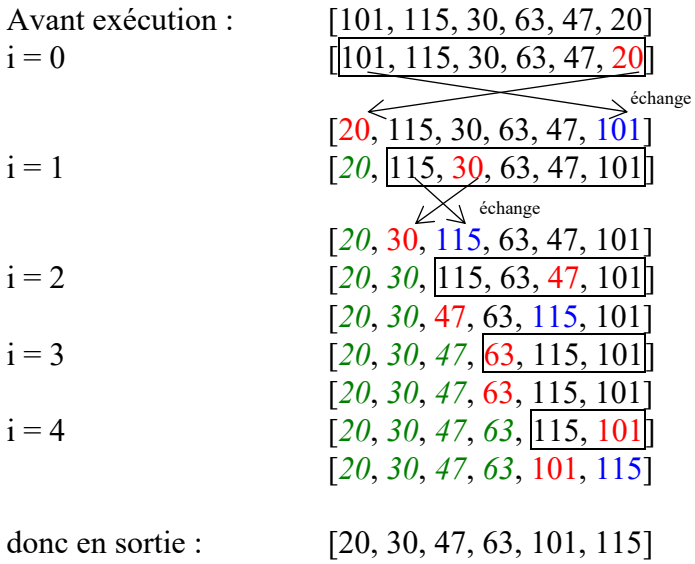
III – Tri par minimum successif (ou tri par sélection)

1) Principe

Le tri par minimum successif est un tri par sélection : pour une place donnée, on sélectionne l'élément qui doit y être positionné.
De ce fait, si on parcourt le tableau de gauche à droite, on positionne à chaque fois le plus petit élément qui se trouve dans le sous-tableau droit. Ou plus généralement : pour trier le sous-tableau t[i..nbElements] il suffit de positionner au rang i le plus petit élément de ce sous-tableau et de trier le sous-tableau t[i+1..nbElements].

2) Exemple

Voyons ce qui se passe lorsque l'on utilise cet algorithme sur le tableau [101, 115, 30, 63, 47, 20] :



En rouge : plus petit élément du sous-tableau de droite
En bleu : élément échangé
Encadré : sous-tableau dans lequel on recherche le min
En vert : éléments déjà triés (à leur place définitive)

pas d'élément échangé puisque le plus petit est déjà au bon emplacement

3) Fonction minimum_ tableau

Pour écrire l'algorithme de tri par minimum successif, il nous manque la fonction `minimum_tableau` qui détermine l'indice du plus petit élément dans le sous-tableau de droite. Elle doit prendre en entrée un tableau d'entier et l'indice à partir duquel commencer la recherche du minimum et renvoyer un entier correspondant à l'indice de l'élément le plus petit du tableau.

Sa signature est donc : `int minimum_tableau(int[] tableau, int indice_debut)`

Question 2 : Recherche du minimum d'un tableau
1) Ecrire l'algorithme de cette fonction
2) Programmer la fonction en python en utilisant éventuellement la base fournie dans le fichier minimum_tableau.py.

4) Programme complet

Maintenant que nous avons écrit les fonctions principales, nous pouvons écrire un programme implémentant l'algorithme de tri par minimums successif.

Question 3 : Tri par sélection
1) En se servant du travail précédent, écrire la procédure `tri_par_minimums_successifs(t)` où t est un tableau à trier et modifier le programme principal pour que le programme fasse le tri du tableau et affiche le tableau trié.
2) Cet algorithme est-il en place ? stable ?

5) Conclusion

Cet algorithme est simple à expliquer et à programmer, il est en place mais instable. On pourrait montrer que sa complexité dans le pire des cas est quadratique : $O(n^2)$. Il n'est donc pas très rapide.

IV - Tri à bulles

Voyons maintenant un algorithme légèrement plus compliqué. Le tri à bulles ou tri par propagation consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

1) Principe

On parcourt les n cases du tableau du début à la fin en intervertissant systématiquement les éléments lorsqu'ils ne sont pas en ordre croissant. Cela permet de faire remonter le maximum à la fin du tableau tout en ayant partiellement trié. Ensuite on recommence l'opération, mais on s'arrêtera à n-1 cases car on sait déjà que le maximum est à sa place à la fin du tableau. On reprend ensuite avec les n-2 premières cases, etc... jusqu'à n'avoir plus que 1 case à trier.

2) Exemple

Voyons ce qui se passe lorsque l'on utilise cet algorithme sur le tableau [101, 115, 30, 63, 47, 20] :
En rouge : plus grand élément de la paire examinée
Encadré : paire dont on compare les valeurs
En vert : éléments triés qui ne sont plus examinés

Avant exécution :	[101, 115, 30, 63, 47, 20]	
i = 0 ; iMax = 5	[101, 115, 30, 63, 47, 20]	115 est bien placé, donc on ne change rien
i = 1	[101, 115, 30, 63, 47, 20]	115 est plus grand que 30, donc on échange
	[101, 30, 115, 63, 47, 20]	
i = 2	[101, 30, 115, 63, 47, 20]	115 est plus grand que 63, donc on échange
	[101, 30, 63, 115, 47, 20]	
i = 3	[101, 30, 63, 115, 47, 20]	115 est plus grand que 47, donc on échange
	[101, 30, 63, 47, 115, 20]	
i = 4	[101, 30, 63, 47, 115, 20]	115 est plus grand que 20, donc on échange
i = iMax	[101, 30, 63, 47, 20, *115*]	115 est maintenant placé
i = 0 ; iMax = 4	[101, 30, 63, 47, 20, *115*]	
i = 1	[30, 101, 63, 47, 20, *115*]	
i = 2	[30, 63, 101, 47, 20, *115*]	
i = 3	[30, 63, 47, 101, 20, *115*]	
i = iMax	[30, 63, 47, 20, *101*, *115*]	101 est également placé
i = 0 ; iMax = 3	[30, 63, 47, 20, *101*, *115*]	
i = 1	[30, 63, 47, 20, *101*, *115*]	
i = 2	[30, 47, 63, 20, *101*, *115*]	
i = iMax	[30, 47, 20, *63*, *101*, *115*]	
i = 0 ; iMax = 2	[30, 47, 20, *63*, *101*, *115*]	
i = 1	[30, 47, 20, *63*, *101*, *115*]	
i = iMax	[30, 20, *47*, *63*, *101*, *115*]	
i = 0 ; iMax = 1	[30, 20, *47*, *63*, *101*, *115*]	
i = iMax	[*20*, *30*, *47*, *63*, *101*, *115*]	
donc en sortie :	[20, 30, 47, 63, 101, 115]	

3) Programme

Question 4 : Tri à bulle
1) En se servant de la procédure `echange()` écrire une procédure de tri à bulle. On pourra réutiliser le programme principal fait pour le tri par minimum successifs en changeant juste le nom de la procédure de tri appelée.
2) Cet algorithme est-il en place ? stable ?

4) Comparaison des algorithmes

Pour comparer les algorithmes, on va déterminer leur temps d'exécution. Pour calculer le temps d'exécution d'une portion de programme, on peut utiliser la fonction `perf_counter()` du module `time` qui renvoie un nombre flottant égal au temps système pour notre processus en secondes. En faisant la différence entre la date après et avant exécution, on en déduit la durée.

```
import time

tempsDebut = time.perf_counter()
…
Code dont on doit évaluer la durée d'exécution
…
tempsFin = time.perf_counter()

duree = tempsFin - tempsDebut    # la durée est la différence des deux dates
```

Question 5 : Mesure des temps d'exécution
Dans un même programme appeler successivement le tri par minimum successif et le tri par bulle sur le même tableau (*attention à bien le ré-initialiser avant la deuxième exécution : le temps de traitement d'un tableau déjà trié et d'un tableau non trié n'est pas le même*) et comparer leurs temps d'exécution. Quel algorithme est le plus rapide ?

5) Conclusion

Cet algorithme est assez simple à expliquer et à programmer, il est en place et stable. On pourrait montrer que sa complexité dans le pire des cas est quadratique : $O(n^2)$. Il n'est donc pas très rapide, mais cependant généralement plus rapide que le tri par minimum successif pour de petits tableaux (ils n'ont la même complexité que dans le pire des cas).
Son intérêt est surtout de trier assez efficacement des listes chainées (plutôt que des tableaux) car il ne fait que permuter des éléments adjacents de la liste (ce qui est simple à faire sur une liste chaînée, tandis qu'échanger deux éléments quelconques peut être plus complexe).

V - Tri rapide

Voyons maintenant un algorithme plus efficace puisqu'il opère en temps pseudo-linéaire : $O(n \cdot \log(n))$.
Il est instable, n'est pas en place (il doit allouer un espace de travail égal à la taille du tableau à trier) et est un peu plus complexe à mettre en œuvre que les autres.

Question 6 : Tri rapide (Quicksort)
Voir sur https://fr.wikipedia.org/wiki/Tri_rapide les explications sur le fonctionnement de ce tri et programmer ce tri en python.
Comparer sa vitesse d'exécution avec les deux autres algorithmes (la différence se verra mieux avec de grands tableaux) et conclure.

VI - Conclusion

Il existe de nombreux algorithmes de tri (nous n'en avons vu qu'une toute petite partie) qui se distinguent par leur complexité de compréhension, leur vitesse d'exécution, leur stabilité et leur complexité spatiale (en place ou non). Suivant le type de données à trier (alphabétiques, nombres, tableaux ou listes, peu désordonnées ou très désordonnées, …) et leur taille tel ou tel algorithme sera préférable.

Pour plus de détails, voir :
https://fr.wikipedia.org/wiki/Algorithme_de_tri
http://lwh.free.fr/pages/algo/tri/tri.htm
Vous pouvez aussi jeter un œil au fichier « Comparaison_tris.py » et l'exécuter en changeant les paramètres.