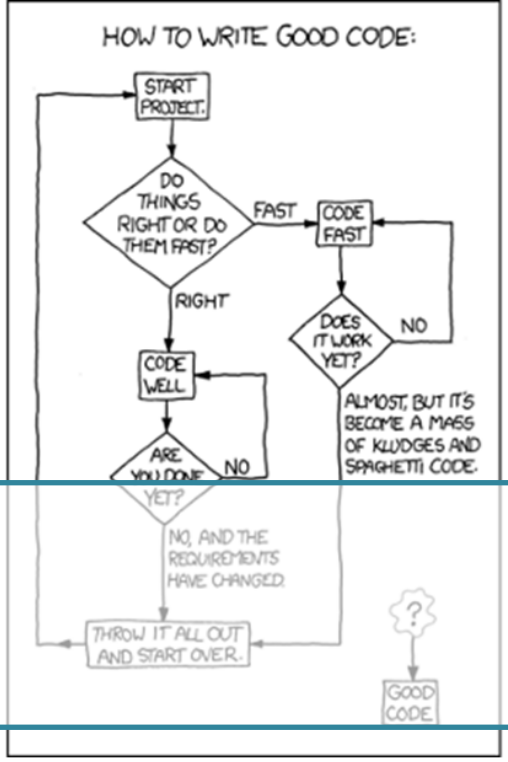
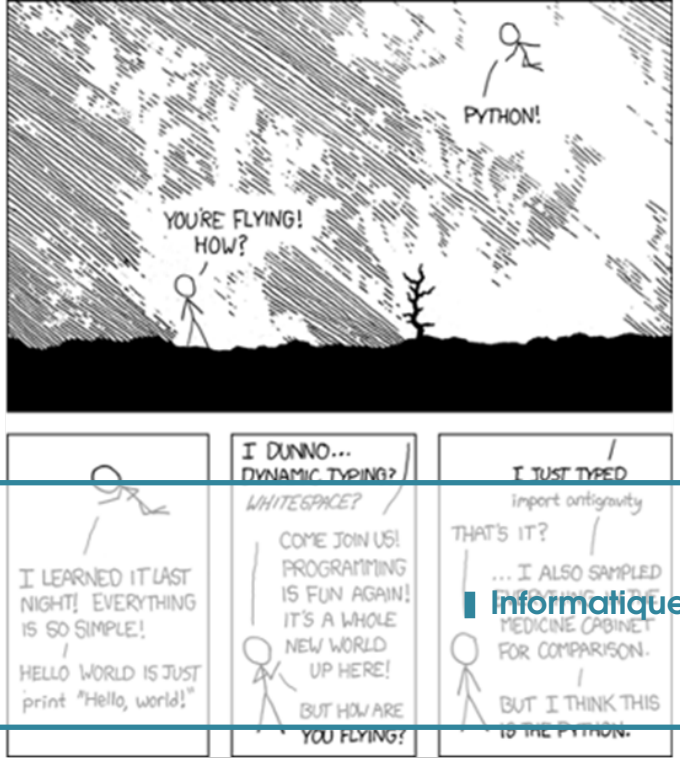




Cours

Ch. 2
Méthodes de tri

E. Durif – X. Pessoles
J.-P. Berne
MPSI

Thème : Tris.
Commentaires :
- algorithmique quadratique : tri par insertion, par sélection ;
- tri par partition-fusion ;
- tri par comptage.

On fait observer différentes caractéristiques (par exemple2 stable ou non, en place ou non, comparatif ou non ...).

1 Introduction et objectifs

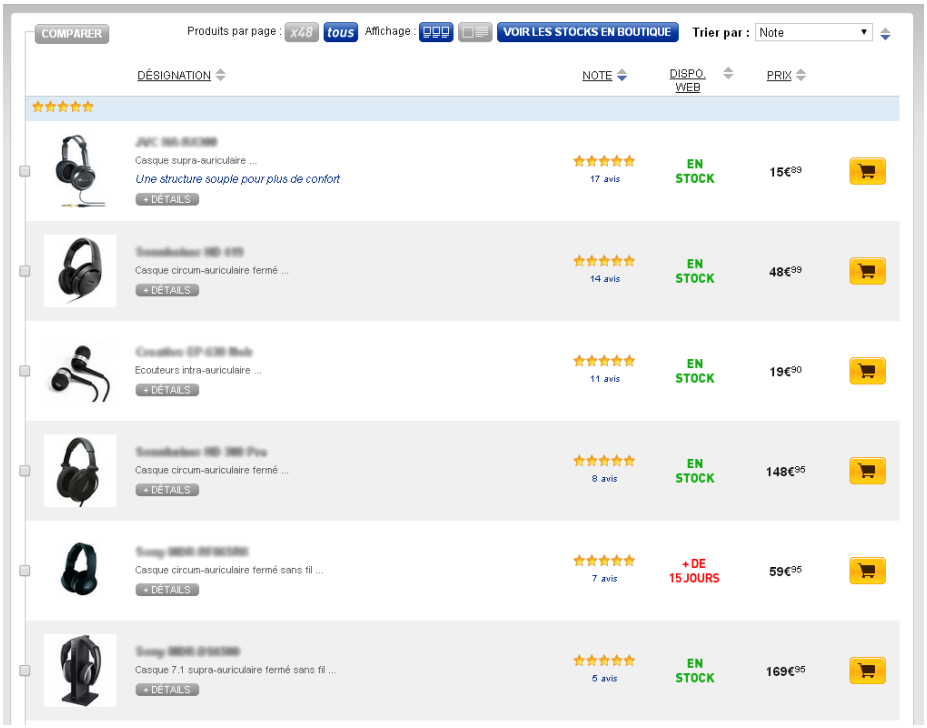
Objectif On suppose dans toute la suite que l'on trie des tableaux dans l'ordre croissant. Les algorithmes présentés permettent de trier n'importe quel type d'éléments à partir du moment où ces éléments sont munis d'un ordre total. Cependant nous allons nous focaliser ici sur le tri d'entiers.

Les principales capacités développées ici sont :
- Comprendre un algorithme de tri et expliquer ce qu'il fait.
- S'interroger sur l'efficacité algorithmique temporelle d'un algorithme.
- Distinguer par leurs complexités deux algorithmes résolvant un même problème.
- Programmer un algorithme dans un langage de programmation moderne et général.

1.1 exemple d'application

Exemple 1 : *tri d'avis sur un site marchand*

On s'intéresse ici à l'algorithme permettant à un client de trier les résultats d'un site marchand.



On ne s'intéresse ici qu'à une version simplifiée de l'exemple. On ne se focalise que sur le critère note et plus précisément sur le nombre d'avis.
Avant que l'utilisateur choisisse de classer par notes on a une liste non triée $T = [5, 14, 11, 8, 17, 7]$ et l'on souhaite obtenir $T = [5, 7, 8, 11, 14, 17]$.
Pour ce type d'application c'est le temps maximum qu'il faut limiter. En effet une valeur moyenne pourrait permettre de garantir que sur tous les achats les temps d'attente soient raisonnables pour un ensemble de clients. Cependant, ceci ne pourrait garantir qu'aucun client n'ait pas une attente trop longue.

1.2 Complément sur la complexité

Propriété Différents types d'opérations

Dans un algorithme on distinguera trois types d'opérations élémentaires différentes :

- les accès mémoire pour lire ou écrire la valeur d'une variable ;
- les opérations arithmétiques : addition, soustraction, multiplication, division ...
- les comparaisons entre deux entiers ou deux réels.

Exemple 2 :

exemple : si on considère l'instruction $c \leftarrow a + b$, on peut compter quatre opérations élémentaires :

- l'accès en mémoire pour lire la valeur de a,
- l'accès en mémoire pour lire la valeur de b,
- l'addition de a et b,
- l'accès en mémoire pour écrire la nouvelle valeur de c.

Définition 1 : ***Application de la complexité dans le cas d'algorithme de tri***

- Pour les algorithmes de tri on pourra par exemple exprimer la complexité en fonction du nombre de données à trier.
- Néanmoins, il se peut qu'avec deux jeux de données différents correspondant aux mêmes paramètres, le nombre d'opérations effectuées ne soit pas le même. Par exemple, un algorithme de tri pourra être plus rapide s'il s'agit de trier des données déjà triées que s'il s'agit de données très désordonnées. On peut alors s'intéresser à :
  - la complexité **dans le pire des cas** : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus grand nombre d'opérations élémentaires effectuées sur l'ensemble des instances correspondant à ces paramètres ; on cherche ainsi un majorant de la complexité qui puisse être atteint dans certains cas ; c'est ce qu'on fait le plus généralement ;
  - la complexité **dans le meilleur des cas** : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus petit nombre d'opérations élémentaires effectuées sur l'ensemble des instances correspondant à ces paramètres ; cette complexité peut venir compléter la précédente mais ne sera jamais suffisante pour l'utilisateur.

Exemple 3 : ***Choix du type de complexité pour le tri d'avis sur un site marchand***

Dans notre exemple, c'est la complexité dans le pire des cas qui nous intéresse. En effet, même si les attentes longues sont rares, cela peut faire perdre quelques clients, ce qui n'est ici pas acceptable.

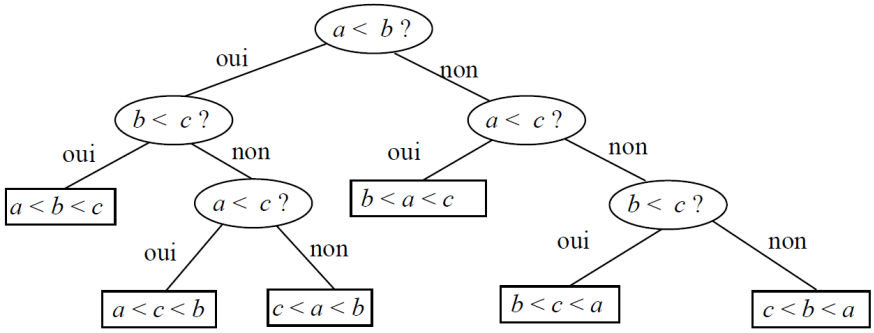
1.3 Complexité d'un algorithme de tri

Définition 2 : ***Tri comparatif***

Un **algorithme comparatif** (à opposer au non-comparatif comme le tri baquet ou par paquet) est basé sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. On va chercher ici à évaluer la complexité théorique des tris comparatifs en se basant sur le nombre de comparaisons.

Exemple 4 : *Algorithme de tri de trois données a, b, c*

Cet algorithme correspond au fonctionnement du tri insertion que nous verrons plus loin :



Cet arbre signifie :

1. la première comparaison faite est « a < b ? ».
2. Si la réponse est oui, la comparaison suivante est « b < c ? »,
3. si la réponse est non c'est « a < c ? ».
4. Lorsqu'une permutation est déterminée, on est dans ce que l'on appelle une feuille de l'arbre.

On voit qu'on peut avoir plus ou moins de chance ;

- pour deux des permutations, on fait deux comparaisons,
- pour les quatre autres, on fait trois comparaisons.
- Le plus grand nombre de comparaisons est 3,
- le meilleur est 2,
- le nombre moyen de comparaisons est : $(2\times 2+4\times 3)/6\approx 2,67$.

Propriété Principe Soit un tableau de n données, la réponse cherchée par le tri est l'une des $n!$ permutations possibles des données. exemple : Pour trier $[a,b,c]$, il y a une seule possibilité parmi ces 6 permutations qui conviennent :

$[a,b,c]$

- $[a,c,b]$
- $[b,a,c]$
- $[b,c,a]$
- $[c,a,b]$
- $[c,b,a]$

Pour tout algorithme de tri comparatif, la première comparaison faite consiste à prendre deux des données a et b de la liste et à poser la question : "a-t-on a < b?", ce qui divise les $n!$ permutations envisageables en deux parties égales ;

R les comparaisons suivantes ne garderont pas une telle symétrie : on aura de la chance quand il y aura moins de la moitié des permutations restant possibles qui correspondent à la réponse obtenue et de la malchance sinon.

Propriété Complexité Si on s'intéresse à la complexité dans le **pire des cas** d'un algorithme (la permutation considérée correspond toujours à un cas où on n'a jamais de chance), alors chaque comparaison élimine au plus la moitié des permutations encore envisageables. Dans ce cas,

après k comparaisons, nombre de permutations envisageables :

- Condition pour avoir une permutation unique ;
- Nombre minimum de comparaisons :
- Comme $log_2(n!)$ est équivalent à $n.log_2(n)$, on a le résultat suivant :

Résultat : *Tout algorithme comparatif fait dans le pire des cas au moins de l'ordre de* $n.log_2(n)$ ***comparaisons*** *soit une complexité de* $O(N\cdot log(N))$.

2 Tri par insertion

2.1 Principe

Définition 3 : *Principe du tri par insertion*

Le **tri par insertion** est le tri que l'on effectue naturellement, par exemple pour trier un jeu de cartes. On trie les premières puis à chaque nouvelle carte on l'ajoute à l'ensemble déjà trié à la bonne place.
Ce trie s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données.

Exemple 5 : *Illustration du tri par insertion*

On représente en rouge les valeurs triées. La clef est en bleu. Les données décalées sont en vert.

Tableau T initial : 5 est la clef.	5	14	11	8
La clef est comparée avec la liste des valeurs déjà triées (vide ici).	5	14	11	8
La clef est 14	5	14	11	8
On compare la clef aux valeurs déjà triées (ici 5). Elle est plus grande et reste donc à sa position.	5	14	11	8
La clef est 11.	5	14	11	8
On compare la clef aux valeurs déjà triées : la première valeur comparée est 14. 14 est plus grand donc on décale 14 vers la droite pour laisser une place à la clef.	5	14	14	8
On compare la clef aux valeurs déjà triées : la valeur suivante est 5. 5 est plus petit donc on a trouvé la place de la clef et on l'écrit.	5	11	14	8
La clef est 8.	5	11	14	8
On compare la clef 8 à 14. 14 est plus grand donc on le décale en le copiant à la place de la clef.	5	11	14	14
On compare la clef 8 à 11. 11 est plus grand donc on le décale en le copiant à la place de la valeur supérieure (on écrase donc une des occurrences de 14).	5	11	11	14
On compare la clef 8 à 5. 5 est plus petit donc on est à la bonne place pour copier la clef (on écrase donc l'occurrence de 11 en doublon).	5	8	11	14

2.2 Implémentation

```
for i ← 2 to n do
    j ← i
    v ← T[i]
    while j > 1 and v < T[j−1] do
        T[j] ← T[j−1]
        j ← j−1
    end
    T[j] ← v
end
Output: t
```

- On stocke les données dans un tableau T entre les indices 1 et n.
- On utilise deux variables notées i, j et une variable v (la clef) du même type que les données de la liste.
- On réalise une première boucle qui va permettre de parcourir tous les éléments $T[i]$ du tableau à trier. Chacun de ces éléments sera alors successivement la clef.
- La deuxième boucle (*while*) va nous permettre pour chaque valeur de la clef de tester successivement les valeurs déjà triées jusqu'à trouver une valeur inférieure. On aura ainsi déterminé la place de la clef.
- A chaque fois qu'une valeur est plus grande que la clef on la décale vers la droite pour laisser la place d'écrire la clef. Dès que l'on arrive sur une valeur inférieure on a trouvé la bonne place pour écrire la clef.

Algorithme 1 :

Écriture sous Python de l'algorithme tri par insertion

Exemple 6 : ***Application avec affichage pas a pas sur l'exemple***

On insère les lignes $print$ pour réaliser l'affichage

```
i= 1 j= 1 v= 14 T= [5, 14, 11, 8, 17, 7]
i= 1 j= 1 v= 14 T= [5, 14, 11, 8, 17, 7]
i= 2 j= 2 v= 11 T= [5, 14, 11, 8, 17, 7]
i= 2 j= 1 v= 11 T= [5, 14, 14, 8, 17, 7]
i= 2 j= 1 v= 11 T= [5, 11, 14, 8, 17, 7]
i= 3 j= 3 v= 8 T= [5, 11, 14, 8, 17, 7]
i= 3 j= 2 v= 8 T= [5, 11, 14, 14, 17, 7]
i= 3 j= 1 v= 8 T= [5, 11, 11, 14, 17, 7]
i= 3 j= 1 v= 8 T= [5, 8, 11, 14, 17, 7]
i= 4 j= 4 v= 17 T= [5, 8, 11, 14, 17, 7]
i= 4 j= 4 v= 17 T= [5, 8, 11, 14, 17, 7]
i= 5 j= 5 v= 7 T= [5, 8, 11, 14, 17, 7]
i= 5 j= 4 v= 7 T= [5, 8, 11, 14, 17, 17]
i= 5 j= 3 v= 7 T= [5, 8, 11, 14, 14, 17]
i= 5 j= 2 v= 7 T= [5, 8, 11, 11, 14, 17]
i= 5 j= 1 v= 7 T= [5, 8, 8, 11, 14, 17]
i= 5 j= 1 v= 7 T= [5, 7, 8, 11, 14, 17]
>>>
```

2.3 Complexité

Propriété Complexité

La fonction $tri\_insertion$ effectue le même nombre de comparaisons et d'affectations. Lorsque la boucle while insère l'élément $T[i]$ à la position $i-k$, elle effectue $k+1$ comparaisons.

- **Dans le meilleur des cas** le tableau est déjà trié. La valeur clef est donc forcément plus grande que toutes les précédentes. Dès la première comparaison on s'arrête : k vaut 0. Soit une comparaison par boucle while réalisée, il y a donc $n-1$ comparaisons à effectuer au total. La complexité est donc de classe linéaire : $C(n)=O(n)$.
- **Dans le pire des cas**, le tableau est trié à l'envers. La valeur clef est donc forcément plus petite que toutes les précédentes. Il faut donc comparer et décaler toutes les valeurs précédentes jusqu'à mettre la valeur clé en première position : k vaut i, avec i qui varie de 1 à $n-1$.
- On en déduit donc un nombre total de $\frac{n\times(n-1)}{2}$ comparaisons (somme d'une suite arithmétique).

La complexité est donc de classe quadratique : $C(n)=O(n^2)$.

	meilleur cas	pire cas
comparaisons	N	$N^2/2$
affectations	N	$N^2/2$

L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » ($C(n)=O(n)$). Il surpasse alors toutes les autres méthodes de tri qui sont au mieux en $O(n\times ln(n))$.

3 Tri rapide (ou "quicksort")

3.1 Principe

Définition 4 : *Tri rapide*

L'algorithme de tri rapide fait partie de la catégorie des algorithmes « diviser pour régner ».
A chaque appel de la fonction tri on choisit une valeur **"pivot"**, par exemple le premier élément. On effectue une partition des élément à trier. Un premier groupe est constitué de valeurs inférieures au pivot et un deuxième avec les valeurs supérieures. Le pivot est alors placé définitivement dans le tableau.
On traite alors chacun des groupes de façon indépendante. On peut les traiter avec le même algorithme.

Exemple 7 : *Illustration du tri rapide*

On représente en :
- bleu, le pivot ;
- vert, les valeurs triées ;
- rouge, les valeurs de gauche d'une segmentation ;
- jaune, les valeurs de droite d'une segmentation.

Tableau T initial : 5 est le pivot	5 14 11 8 17 7
On partitionne pour trouver une partie droite (en jaune) et une partie gauche (en rouge). Sauf qu'ici il n'y aura pas de partie gauche car 5 est la plus petite valeur. 5 est donc à sa place définitive.	5 14 11 8 17 7
On traite de façon indépendante les partie gauche et droite. Ici pas de partie gauche. On cherche alors les nouveaux pivots (ici 14).	5 14 11 8 17 7
On segmente alors les parties droite et gauche autour du pivot 14 dont on a ainsi déterminé la position définitive.	5 7 8 11 14 17
On traite de façon indépendante les partie gauche et droite. On recherche pour chacune le pivot (ici 7 et 17).	5 14 7 11 8 17
Pour la chaine de pivot 7 on segmente mais il n'y a pas de partie gauche. Le pivot 17 est seul dans sa chaine il est donc directement à la bonne position.	5 14 7 11 8 17
On traite de façon indépendante les partie gauche et droite. Ici pas de partie gauche. On cherche alors les nouveaux pivots (ici 8).	5 7 14 17 8 11
On classe le seul élément restant par rapport au pivot.	5 7 14 17 8 11

3.2 Implémentation

- On crée tout d'abord une fonction "**partition**".
  - Les arguments de cette fonction seront le tableau a et deux indices correspondant au début et à la fin de la partie à segmenter du tableau (par convention g inclus et **d** exclu). En effet on ne segmente pas à chaque fois l'ensemble du tableau.
  - On suppose qu'il y a au moins un élément dans ce segment ce que l'on peut vérifier avec $assert$:
    ```
    assert g<d
    ```
  - On choisit $a[g]$ comme pivot que l'on peut stocker dans une variable v.
  - Le principe consiste alors à parcourir le tableau de la gauche vers la droite, entre les indices $g+1$ et d avec une boucle for.
  - on stockera dans deux sous listes $ainf$ et $asup$ toutes les composantes respectivement inférieures ou supérieures au pivot.

- on rassemblera les deux sous listes ainsi que le pivot pour donner la liste a partitionnée (en assemblant également le reste de la liste a).
- On note m l'indice qui correspond au partitionnement de a.
- On sortira de cette fonction m et le tableau a.

Algorithme 2 : *Algorithme de partitionnement pour le tri rapide*

- La fonction tri rapide est alors implémentée de façon récursive.
  - On utilise pour cela une fonction tri_rapide qui prend les même arguments que la fonction $partition$.
  - **Si** $g \geq d-1$, il y a au plus un élément à trier donc rien à faire (initialisation de la récursivité),
  - **Sinon** il faut
    partitionner les élément entre g et d,
    faire un appel récursif pour trier $a[g \cdots m[$ (partie gauche),
    faire un appel récursif pour trier $a[m+1 \cdots d[$ (partie droite),

Algorithme 3 :

Algorithme récursif du tri rapide

- Pour trier un tableau il suffit d'appeler tri_rapide sur la totalité des éléments.

Exemple 8 : *Algorithme complet du tri rapide avec affichage des tris successifs*

Propriété Efficacité de l'algorithme

[**Beynet**] Cette méthode de tri est très efficace lorsque les données sont distinctes et non ordonnées. La complexité est alors globalement en O(n ln(n)). Lorsque le nombre de données devient petit (<15) lors des appels récursifs de la fonction de tri, on peut avantageusement le remplacer par un tri par insertion dont la complexité est linéaire lorsque les données sont triées ou presque.

- [**wack**] D'autre part si le tableau est déjà trié avec le code mis en place on tombe sur la complexité "dans le pire des cas". Une solution simple consiste à ne pas choisir systématiquement le premier élément du segment comme pivot, mais plutôt un élément au hasard. On peut par exemple choisir la valeur de façon aléatoire.

3.3 Complexité (wack)

Pour ce calcul de complexité nous ne considérerons ici que les comparaisons.

La fonction partition fait toujours exactement $d-g-2$ comparaisons. Si la fonction partition détermine un segment de longueur K et un autre de longueur $N-1-K$, la fonction tri_rapide va donc effectuer $N-1$ comparaisons par l'intermédiaire de partition, puis d'autres comparaisons par l'intermédiaire des deux appels récursifs à tri_rapide.

- **Le pire des cas** correspond à $K=0$, ce qui donne, en notant $C(N)$ la complexité du tri d'un tableau de longueur N, l'équation de récurrence suivante :

$$C(N) = N-1+C(N-1)$$

,
Avec $C(1)=0$
d'où $C(N) \sim \frac{N\times(N-1)}{2}$

- **Le meilleur des cas** correspond à un segment coupé en deux moitiés égales, c'est-à-dire $K=N/2$. L'équation de récurrence devient :

$$C(N) = N-1+2C(N/2).$$

On peut montrer que $C(N) \sim NlogN$.

	meilleur cas	pire cas
comparaisons	$NlogN$	$N^2/2$

4 Tri par fusion

4.1 Principe

Cet algorithme fait aussi partie des algorithmes "diviser pour régner".

Le principe consiste à couper le tableau de départ en deux. On trie chacun des groupes indépendamment. Puis on fusionne les deux groupes en utilisant le fait que chacun des groupe est déjà ordonné.

Il est possible pour réaliser l'ordonnancement de chacun des groupes d'utiliser à nouveau l'algorithme de tri de façon récursive.

Exemple 9 : *Illustration sur l'exemple*

On représente en rouge le groupe de gauche, en jaune le groupe de droite, en bleu la valeur testée du groupe de gauche et en gris la valeur testée du groupe de droite. Les valeurs en vert sont les valeurs définitivement triées.

On notera que pour chaque étape, on utilise deux tableaux, le tableau de départ sur la ligne supérieure et le tableau d'arrivée sur la ligne inférieure.

Tableau T initial : On découpe en une deux moitiés (rouge et jaune).	5 14 11 8 17 7
Chacune de ces deux listes est triée de façon indépendante (pas illustré ici) soit par un autre algorithme soit de façon récursive.	5 11 14 7 8 17
On procède alors à l'opération de fusion en comparant les premiers nombres de chaque bloc. On a besoin d'un second tableau que l'on	5 11 14 7 8 17
remplit après les comparaisons successives des éléments des listes L_1 (rouge) et L_2 (jaune). 5<7 on le stock de façon définitive dans T_2.	11 14 7 8 17 5
On continue la fusion en comparant les premiers nombres restants dans chaque bloc (ici 7 et 11).	11 14 7 8 17 5
7<11 on le stocke dans le deuxième tableau.	11 14 8 17 5 7
On continue la fusion en comparant les premiers nombres restants dans chaque bloc (8 et 11).	11 14 8 17 5 7
8<11 on le stocke dans le deuxième tableau.	11 14 17 5 7 8
On continu la fusion en comparant les premiers nombres restants dans chaque bloc (11 et 17).	11 14 17 5 7 8
11<17 on le stocke dans le deuxième tableau.	14 17 5 7 8 11
On continu la fusion en comparant les premiers nombres restants dans chaque bloc (14 et 17).	14 17 5 7 8 11
14<17 on le stock dans le deuxième tableau et comme c'est le dernier restant on ajoute 17.	 5 7 8 11 14 17

4.2 Implémentation

On souhaite trier un tableau a :

- Le caractère récursif du tri fusion fait que l'on s'intéresse seulement à la partie de a entre les indices g (inclus) et d (exclu).
- Afin de découper le groupe à trier en deux, on calcule l'indice médian :

$$m = \frac{g+d}{2}$$

.

- On trie alors récursivement les deux parties :
  - délimitées par g et m d'une part,
  - et m et d d'autre part.
  - **Si** $g \geq d-1$, il y a au plus un élément à trier donc rien à faire (initialisation de la récursivité).
- Il reste à effectuer la fusion. On utilise pour cela un second tableau a_0, alloué une et une seule fois au début du tri.
- On réalise la fonction fusion :

- qui prend en arguments deux tableaux, $a0$ et a, et les trois indices g, m et d.
- Les portions $a0[g \cdots m[$ et $a0[m \cdots d[$ sont supposées triées.
- On veut les fusionner dans $a[g \cdots d[$.
- On va parcourir chacune des listes déjà triées en utilisant un indice i pour L_1 et j pour L_2.
- On compare à chaque itération de la boucle **for** $a0[i]$ et $a0[j]$ et on place la plus petite des valeurs dans $a[k]$ et on incrémente l'indice correspondant i ou j selon celui que l'on a déplacé.
- On traite aussi le cas ou une des deux listes est vide.

Exemple 10 : *Algorithme complet du tri par fusion*

4.3 Complexité (wack)

Si on note $C(N)$ (resp. $f(N)$) le nombre total de comparaisons effectuées par tri_fusion (resp. fusion) pour trier un tableau de longueur N, on a l'équation de récurrence

$$C(N) = 2C(N/2) + f(N)$$

car les deux appels récursifs se font sur deux segments de même longueur $N/2$.

- **Dans le meilleur des cas**, la fonction fusion n'examine que les éléments de l'un des deux segments car ils sont tous plus petits que ceux de l'autre segment. Dans ce cas $f(N) = N/2$ et donc $C(N) \sim \frac{1}{2} N log N$.
- **Dans le pire des cas**, tous les éléments sont examinés par $fusion$ et donc $f(N) = N-1$, d'où $C(N) \sim N log N$.

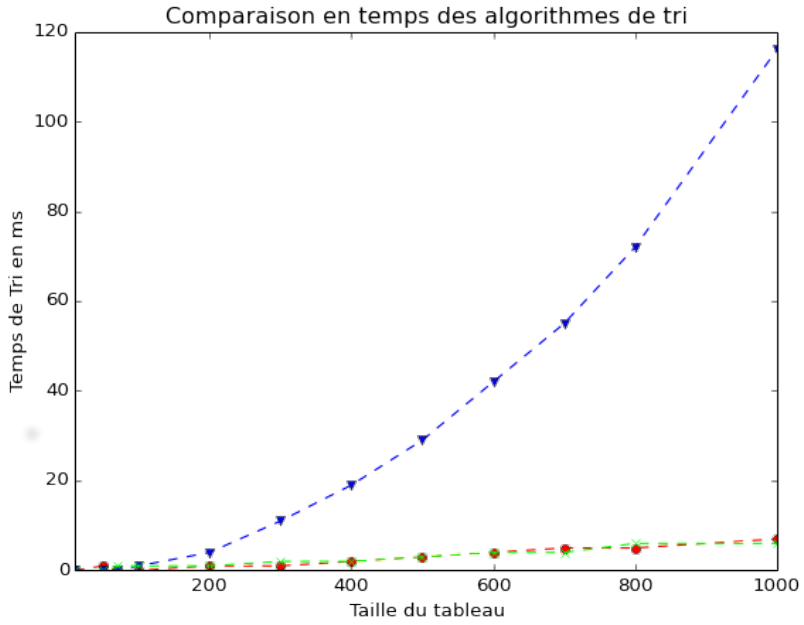
5 Conclusion et temps

Propriété Méthode DISTINGUER PAR LEURS COMPLEXITÉS DEUX ALGORITHMES RÉSOLVANT UN MÊME PROBLÈME :
La première étape consiste à vérifier que les algorithmes résolvent exactement le même problème.

- On compare la complexité en temps dans le pire des cas (préférable a l'utilisation du module Time comme ci dessous car indépendant de la machine).
- Il faut vérifier que le pire des cas peut arriver en situation réelle
- Penser à comparer la complexité en espace qui peut permettre de départager des algorithmes équivalents

Exemple 11 : *Illustration de la comparaison*

Pour comparer les temps globaux sur une même machine (Ici processeur Intel Core i7-4510U Haswell (2 GHz, TDP 15W),Mémoire vive 4 Go) on génère des tableaux de dimensions différentes avec des nombres aléatoires. On utilise le module time pour déterminer les temps. Ici c'est un temps global dépendant fortement de la machine utilisée. On fait varier la taille du tableau et on compare les temps mis par chacun des algorithmes (en rouge tri rapide, en bleu tri insertion, en vert tri fusion).



Références

[1] Irène Charon Olivier Hudry, Algorithmes de tri, cours de Télécom ParisTech, Avril 2014.

[2] B. Wack, S. Conchon, J. Courant, M. de Falco, G. Dowek, J.-C. Filliâtre, S. Gonnord, Informatique pour tous en classes préparatoires aux grandes écoles. Manuel d'algorithmique et programmation structurée avec Python. Nouveaux programmes 2013. Voies MP, PC, PSI, PT, TPC et TSI, Eyrolles, 2013.

[3] Beynet Patrick, Cours d'informatique publié sur le site de l'UPSTI.

6 Présentation

Le tri de données ou de valeurs est omniprésent en informatique. Pour cela, beaucoup d'algorithmes ont été développés afin de réaliser des tris rapidement, notamment lorsque le nombre de données est important.

Définition Stabilité

Définition Tri en place
Un tri est effectué en place lorsque la liste à trier est modifiée jusqu'à devenir triée. Dans le cas contraire, la fonction de tri pourra renvoyer une novelle liste contenent les mêmes éléments, mais triés.

Définition Tri comparatif
Un tri est dit comparatif lorsqu'il s'appuie uniqument sur la comparaison deux à deux des éléménts de la liste et pas sur la valeur ce ces éléments.

7 Tris

Définition Tri par insertion

À partir d'une sous-liste triée, le tri par insertion consiste à parcourir les éléments non triés et de les insérer

successivement dans la sous-liste déjà triée.

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

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

Définition Tri rapide

Soit une liste L non triée. Soit p un terme appelé pivot. Le tri rapide consiste à répartir les éléments strictement inférieur au pivot avant ce dernier et les termes plus grand après le pivot (segemntation). Le pivot est à ce stade trié correctement par rapport aux autres valeurs de la liste. Ce principe est alors appliqué récursivement aux deus-sous listes séparées par le pivot.

```
def segmente(t, i, j):
    p = t[j-1] # On prend comme pivot le dernier élément de la sous liste.
    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] # On positionne le pivot "à sa place".
    return a # On retourne l'index du pivot. Le tableau a été modifié en place.
```

```
def quickSort(t, i, j):
    if i + 1 < j:
        a = segmente(t, i, j)
        quickSort(t, i, a)
        quickSort(t, a + 1, j)
```

```
# Instruction pour trier une liste
quickSort(t, 0, len(t))
```

Définition Tri fusion

Il s'agit d'un tri s'appuyant sur la stratégie diviser pour régner.

L'algorithme est le suivant :

- on divise la liste en deux listes de tailles quasi-identiques ;
- on trie récursivement ces deux listes ;
- on fusionne les deux listes triées.

```
def placer(L :list, p :int, x) :
    """Place un élément x àsa place dans une liste L triée àpartir de l'indice p
    Entrée :
        L : une liste, p : un entier, x : un élément
    Sorties :
        La liste est modifiée mais n'est pas renvoyée. k la valeur de l'indice de la liste où ↙
            l'élément a été placé"""
    k = p
    while ( k < len(L) and x > L[k]) :
        k = k+1
    L.insert(k, x)
    return k

def fusion(a:list, b:list) :
    """Fusionne les deux listes
    Entrée : deux listes a et b triées.
    Sortie : La liste b modifiée"""
    p = 0
```

```
    for x in a :
        p = placer(b, p, x)+1
    return b

def tri_fusion(t : list) :
    """Trie la liste t
    Entrée : une liste.
    Sortie : la liste est modifiée."""
    if len(t) < 2 :
        return (t)
    else :
        m = len(t) // 2
        return (fusion (tri_fusion(t[:m]) , tri_fusion(t[m:]) ))
```

Proposition **Complexité des algorithmes** On note $T(n)$ le nombre de comaraisons nécessaire pour trier une liste de longueur n. On montre que dans le pire des cas, les complexités sont les suivantes :

- tri par insertion : $T_{\text{Max}}(n) = \mathcal{O}\left(n^2\right)$;
- tru rapide : $T_{\text{Max}}(n) = \mathcal{O}\left(n^2\right)$;
- tru partition-fusion : $T_{\text{Max}}(n) = \mathcal{O}\left(n \log n\right)$.

8 Acticité préparatoire

Pour réaliser l'activité associée à ce cours, suivre le lien suivant :

- Sujet : `https://bit.ly/3iAc5do`
- Corrige : `https://bit.ly/3eFuHrt`

9 QCM

Question 1 *On dispose d'une liste de triplets : t = [(1,12,250),(1,12,251), (2,12,250),(2,13,250),(2,11,250),(2,12,249)]. On trie cette liste par ordre croissant des valeurs du second élément des triplets. En cas d'égalité, on trie par ordre croissant du troisième champ. Si les champs 2 et 3 sont égaux, on trie par ordre croissant du premier champ. Après ce tri, quel est le contenu de la liste ?*

1. `[(1,12,249),(1,12,250),(1,12,251),(2,11,250),(2,12,250),(2,13,250)]`.
2. `[(2,11,250),(1,12,249),(1,12,250),(2,12,250),(1,12,251),(2,13,250)]`.
3. `[(2,11,250),(1,12,249),(1,12,250),(1,12,251),(2,12,250),(2,13,250)]`.
4. `[(1,12,249),(2,11,250),(1,12,250),(2,12,250),(2,13,250),(1,12,251)]`.

Question 2 *Quelle valeur retourne la fonction "mystere" suivante ?*

```
def mystere(liste):
    valeur_de_retour = True
    indice = 0
    while indice < len(liste) - 1 :
        if liste[indice] > liste[indice + 1]:
            valeur_de_retour = False
        indice = indice + 1
    return valeur_de_retour
```

1. Une valeur booléenne indiquant si la liste liste passée en paramètre est triée.
2. La valeur du plus grand élément de la liste passée en paramètre.
3. La valeur du plus petit élément de la liste passée en paramètre.
4. Une valeur booléenne indiquant si la liste passée en paramètre contient plusieurs fois le même élément.

Question 3 *Combien d'échanges effectue la fonction Python suivante pour trier un tableau de 10 éléments au pire des cas ?*

```
def tri(tab) :
    for i in range (1, len(tab)) :
        for j in range (len(tab) - i) :
            if tab[j] > tab[j+1] :
                tab[j], tab[j+1] = tab[j+1], tab[j]
```

1. 45.
2. 100.
3. 10.
4. 55.

Question 4 *Que vaut l'expression* `f([7, 3, 1, 8, 19, 9, 3, 5], 0)` *?*

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

1. 1.
2. 2.
3. 3.
4. 4.

Question 5 *Laquelle de ces listes de chaînes de caractères est triée en ordre croissant ?*

1. `['Chat' , 'Cheval' , 'Chien' , 'Cochon']`.
2. `['Cochon' , 'Chat' , 'Cheval' , 'Chien']`.
3. `['Cheval' , 'Chien‘ , 'Chat' , 'Cochon']`.
4. `['Chat' , 'Cochon' , 'Cheval' , 'Chien']`.

Question 6 *Laquelle de ces listes de chaînes de caractères est triée en ordre croissant ?*

1. ['12','142','21','8'].
2. ['8','12','142','21'].
3. ['8','12','21','142'].
4. ['12','21','8','142'].

Question 7 *Quelle est la valeur de la variable* `table` *après exécution du programme Python suivant ?*

```
table = [12, 43, 6, 22, 37]
for i in range(len(table) - 1):
    if table [i] > table [i+1] :
        table [i] ,table [i+1] = table [i+1] ,table [i]
```

1. [12, 6, 22, 37, 43].
2. [6, 12, 22, 37, 43].
3. [43, 12, 22, 37, 6].
4. [43, 37, 22, 12, 6].

Question 8 *Un algorithme cherche la valeur maximale d'une liste non triée de taille n. Combien de temps mettra cet algorithme sur une liste de taille $2n$?*

1. Le même temps que sur la liste de taille n si le maximum est dans la première moitié de la liste.
2. On a ajouté n valeurs, l'algorithme mettra donc n fois plus de temps que sur la liste de taille n.
3. Le temps sera simplement doublé par rapport au temps mis sur la liste de taille n.
4. On ne peut pas savoir, tout dépend de l'endroit où est le maximum.

Question 9 *Quel est le coût en temps dans le pire des cas du tri par insertion ?*

1. $\mathcal{O}(n)$.
2. $\mathcal{O}\left(n^2\right)$.
3. $\mathcal{O}\left(2^n\right)$.
4. $\mathcal{O}\left(\log n\right)$.

Question 10 *On souhaite écrire une fonction* `tri_selection(t)`*, qui trie le tableau* `t` *dans l'ordre croissant : parmi les 4 programmes suivants, lequel est correct ?*

```
def tri_selection(t) :
    for i in range (len(t)-1) :
        min = i
        for j in range(i+1,len(t)):
            if t[j] < t[min]:
                min = j
        tmp = t[i]
        t[i] = t[min]
        t[min] = tmp
def tri_selection(t) :
    for i in range (len(t)-1) :
        min = i
        for j in range(i+1,len(t)-1):
            if t[j] < t[min]:
                min = j
        tmp = t[i]
        t[i] = t[min]
        t[min] = tmp
def tri_selection(t) :
    for i in range (len(t)-1) :
        min = i
        for j in range(i+1,len(t)):
            if t[j] < min:
                min = j
        tmp = t[i]
        t[i] = t[min]
        t[min] = tmp
def tri_selection(t) :
    for i in range (len(t)-1) :
        min = i
        for j in range(i+1,len(t)):
```

```
            if t[j] < t[min]:
                min = j
        tmp = t[i]
        t[min] = t[i]
        t[i] = tmp
```

1. Fonction 1.
2. Fonction 2.
3. Fonction 3.
4. Fonction 4.

Question 11 *De quel type de tri s'agit-il ?*

```
def tri(lst):
    for i in range(1,len(lst)):
        valeur = lst[i]
        j = i
        while j>0 and lst[j-1]>valeur:
            lst[j]=lst[j-1]
            j = j-1
        lst[j]=valeur
```

1. Tri par insertion.
2. Tri fusion.
3. Tri par sélection.
4. Tri à bulles.

Question 12 *De quel type de tri s'agit-il ?*

```
def tri(lst):
    nb = len(lst)
    for i in range(0,nb):
        ind_plus_petit = i
        for j in range(i+1,nb) :
            if lst[j] < lst[ind_plus_petit] :
                ind_plus_petit = j
        if ind_plus_petit is not i :
            temp = lst[i]
            lst[i] = lst[ind_plus_petit]
            lst[ind_plus_petit] = temp
```

1. Tri par insertion.
2. Tri fusion.
3. Tri par sélection.
4. Tri à bulles.

Question 13 *Un algorithme est en complexité quadratique. Codé en python, son exécution pour des données de taille 100 prend 12 millisecondes. Si l'on fournit des données de taille 200 au programme, on peut s'attendre à un temps d'exécution d'environ :*

1. 48 millisecondes.
2. 24 millisecondes.
3. 12 millisecondes.
4. 96 millisecondes.

Question 14 *À quel type tri correspond l'invariant de boucle ci-dessous :*

- tous les éléments d'indices 0 à $i-1$ sont déjà triés,
- tous les éléments d'indices i à n sont de valeurs supérieures à ceux de la partie triée.

1. Tri par insertion.
2. Tri fusion.
3. Tri par sélection.
4. Tri à bulles.

Question 15 *Quel est l'invariant de boucle qui correspond précisément à cet algorithme ?*

On considère un algorithme de tri par sélection, dans lequel la fonction `echanger(tab[i], tab[j])` effectue l'échange des ième et jième valeurs du tableau `tab`.

```
nom: tri_sélection

paramètre: tab, tableau de n entiers, n>=2

Traitement:
pour i allant de 1 àn-1:
    pour j allant de i+1 àn:
        si tab[j] < tab[i]:
            echanger(tab[i], tab[j])
renvoyer tab
```

1. Tous les éléments d'indice supérieur ou égal à i sont triés par ordre croissante.
2. Tous les éléments d'indice compris entre 0 et i sont triés et les éléments d'indice supérieurs ou égal à i leurs sont tous supérieurs.
3. Tous les éléments d'indice supérieur ou égal à i sont non triés.
4. Tous les éléments d'indice compris entre 0 et i sont triés, on ne peut rien dire sur les éléments d'indice supérieur ou égal à i.

Question 16 *Quel est le type de tri qui correspond à cet algorithme ?*

```
nom: tri_mystere

paramètre: tab, tableau de n entiers, non trié, non vide

Traitement:
pour i allant de 1 àn-1:
    pour j allant de i+1 àn:
        si tab[j] < tab[i]:
            echanger(tab[i], tab[j])
renvoyer tab
```

1. Tri par insertion.
2. Tri fusion.
3. Tri par sélection.
4. Tri rapide.

Question 17 *Quel est l'invariant de boucle qui correspond précisément à cet algorithme ?*

```
nom: tri_insertion

paramètre: tab, tableau de n entiers, n >= 2

Traitement:
pour i allant de 2 àn:
    j = i
    tant que j > 1 et tab[j-1] > tab[j]:
        echanger(tab[j-1], tab[j])
        j = j-1
renvoyer tab
```

1. Tous les éléments d'indice compris entre 0 et i sont triés et les éléments d'indice supérieurs ou égal à i leurs sont tous supérieurs.
2. Tous les éléments d'indice supérieur ou égal à i sont triés par ordre croissant.
3. Tous les éléments d'indice compris entre 0 et i sont triés, on ne peut rien dire sur les éléments d'indice supérieur ou égal à i.
4. Tous les éléments d'indice supérieur ou égal à i sont non triés par ordre croissants.

Question 18 *Parmi les propositions suivantes, quelle est celle qui ne correspond pas à une méthode de tri ?*

1. Par sélection.
2. Par insertion.
3. Par rotation.
4. Par fusion.