Objectifs

- comprendre les relations d'ordre sur divers types et structures de données
- connaître le fonctionnement de la méthode `sort` définie sur les listes
- découvrir les algorithmes de tri élémentaires : par *sélection* et par *insertion*
- s'initier à la complexité des algorithmes et comprendre l'intérêt d'une telle étude.

1 Motivation

Réaliser un tri ou un classement est une opération relativement courante dans la vie quotidienne :

- classer les cartes d'un jeu,
- classer par ordre alphabétique les livres d'une bibliothèque,
- classer des concurrents selon leurs performances, etc…

1.1 Remarque

Dans la vie courante, les deux verbes *trier* et *classer* ne sont pas synonymes.

- *trier* ou effectuer un *tri* c'est répartir les éléments en paquets correspondant à un certain critère : par exemple séparer les personnes d'une assemblée selon leur sexe ou selon leur langue maternelle.
- *classer* ou effectuer un *classement* c'est mettre des éléments selon un certain ordre : par exemple ranger les personnes d'une assemblée de la plus petite à la plus grande, ou de la plus jeune à la plus vieille.

En informatique les mots *tri* et *trier* sont à prendre avec le sens de *classement* et *classer*.

Gérer un agenda téléphonique est aussi une forme de tri, puisqu'un ordre (alphabétique selon le nom des individus) est appliqué, même si ce tri est fait petit à petit, au fur et à mesure de l'insertion de nouveaux numéros.

Un tri porte généralement sur un nombre assez important de données. En effet, lorsque l'on a peu de numéros de téléphone à gérer, on se contente souvent de les inscrire dans l'ordre où ils sont connus sur une feuille libre, mais, dès que leur nombre devient trop important[1] , on ressent le besoin de les classer et donc de les trier dans un annuaire, afin d'accéder plus rapidement à une information.

Effectuer un tri est souvent assez long et fastidieux (du moins quand les données initiales ne sont pas ou peu triées). Cependant, l'intérêt d'une telle opération, une fois réalisée, est de pouvoir facilement accéder aux différentes données en s'appuyant sur le critère du tri. C'est le cas en particulier de l'algorithme efficace de recherche dichotomique (cf `sec-recherche-dicho`).

Un tri est toujours réalisé selon un critère, particulier et arbitraire, portant sur les données.

Ainsi, on peut arbitrairement choisir de classer les livres d'une bibliothèque selon un ordre

- croissant ou décroissant par auteurs ou par titres ;
- ou numérique selon les dates de parutions, le nombre de pages[2] …

Du fait de leur fréquente utilisation, les tris ont été très étudiés en informatique et font partie du savoir de base de tous les informaticiens. Ils constituent une excellente introduction à l'analyse des algorithmes et sont un très bon support pour l'étude de problèmes plus généraux. De nombreux algorithmes de tri existent, plus ou moins efficaces et plus ou moins faciles à mettre en œuvre. Nous nous limiterons à l'étude de deux d'entre eux[3] :

- le tri par sélection,
- le tri par insertion.

2 Trier en Python

2.1 Méthode `sort`

Commençons par quelques mots sur une méthode prédéfinie en Python pour les listes : la méthode `sort`.

[1] Il est d'ailleurs difficile d'expliquer à partir de quand ce nombre est trop important, on retrouve ici le problème de savoir combien il faut de grains de sable pour en faire un tas…

[2] On ne voit pas très bien l'intérêt d'un tri de livres selon leur nombre de pages …

[3] D'autres algorithmes sont étudiés en seconde année de licence, en particulier des tris récursifs comme le tri rapide ou le tri par fusion.

Observons une utilisation de cette méthode :

```
>>> l1 = [3, 1, 4, 1, 5, 9, 2]
>>> l1.sort()
>>> l1
[1, 1, 2, 3, 4, 5, 9]
```

On voit sur cet exemple que

- l'application de la méthode `sort` sur la liste `l` ne renvoie aucune valeur (ou plus exactement renvoie `None`)
- et qu'à l'issue de cette application, la liste `l` a été modifiée : ses éléments sont maintenant rangés dans l'ordre croissant : la liste est triée.

Il n'y a pas que les listes de nombres qu'il est possible de trier. On peut trier des listes de chaînes de caractères :

```
>>> l2 = list('TIMOLEON')
>>> l2.sort()
>>> l2
['E', 'I', 'L', 'M', 'N', 'O', 'O', 'T']
>>> l4 = list('Timoleon')
>>> l4.sort()
>>> l4
['T', 'e', 'i', 'l', 'm', 'n', 'o', 'o']
>>> l5 = ['La', 'cigale', 'et', 'la', 'fourmi']
>>> l5.sort()
>>> l5
['La', 'cigale', 'et', 'fourmi', 'la']
```

On peut aussi trier des listes de tuples :

```
>>> l6  = [(3, 'D'), (1, 'B'), (3,'A'), (4, 'C')]
>>> l6.sort()
>>> l6
[(1, 'B'), (3, 'A'), (3, 'D'), (4, 'C')]
```

On peut aussi trier dans l'ordre inverse en ajoutant `reverse=True` en paramètre de la méthode :

```
>>> l1.sort(reverse=True)
>>> l1
[9, 5, 4, 3, 2, 1, 1]
>>> l2.sort(reverse=True)
>>> l2
['T', 'O', 'O', 'N', 'M', 'L', 'I', 'E']
```

2.2 Fonction `sorted`

À côté de la méthode `sort` réservée aux listes, existe une fonction applicable à n'importe quel itérable qui renvoie une nouvelle liste dont les éléments sont ceux de l'itérable dans l'ordre croissant. Cette fonction se nomme `sorted`.

```
>>> sorted('Timoleon')
['T', 'e', 'i', 'l', 'm', 'n', 'o', 'o']
```

Cette fonction accepte aussi le paramètre optionnel `reverse` :

```
>>> sorted('Timoleon', reverse=True)
['o', 'o', 'n', 'm', 'l', 'i', 'e', 'T']
```

La section suivante vise à essayer de comprendre ce que signifient ces classements obtenus avec la méthode `sort` ou la fonction `sorted`.

3 Comparer deux valeurs

Pour trier une structure de données, il faut en comparer les éléments. En mathématiques, les opérations de comparaisons sont désignées par les opérateurs usuels $=$, $\neq$, $<$, etc …

Nous retrouvons ces opérateurs de comparaison dans tous les langages de programmations. En Python, nous avons les opérateurs prédéfinis `==`, `<=`, `<`, etc …

La fonction `sorted` et la méthode `sort` utilisent l'ordre défini par l'opérateur `<=`.

Nous allons examiner dans cette partie la signification de cet opérateur qui est évidente pour certains types de données, mais peut-être moins pour d'autres. Et d'ailleurs que pouvons nous comparer avec ces opérateurs prédéfinis. Répondent-ils à tous nos besoins ?

3.1 Comparer une carotte à un navet ?

L'opérateur `<=` permet de comparer toutes sortes de types de données. Mais il n'est capable de comparer que des valeurs de même type.

Si on tente de comparer deux valeurs de types non comparables, une exception `TypeError` est déclenchée :

```
>>> [] <= (1, 2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: '<=' not supported between instances of 'list' and 'tuple'
```

la raison est que la méthode `__le__` des listes n'accepte pas les arguments du type `tuple`.

3.2 Les nombres

L'opérateur `<=` appliqué sur des données numériques, entiers ou flottants, correspond à l'ordre numérique bien connu.

Les entiers et les flottants sont comparables entre eux :

```
1 <= 3.5
```

```
>>> 1 <= 3.5
True
```

Car la méthode `__le__` des entiers accepte en argument les flottants.

3.3 Les booléens

Le booléen `False` est «plus petit» que le booléen `True`.

```
>>> False <= True
True
>>> True <= False
False
```

Remarque

Rares sont les occasions pour lesquelles on a besoin de trier des booléens.

3.4 Les caractères

L'ordre des caractères est déterminé par celui des numéros Unicode associés. Plus précisément un caractère c_1 est «plus petit» qu'un caractère c_2 si le numéro Unicode du premier est plus petit que celui du second.

```
>>> ord('T')
84
>>> ord('a')
97
>>> 'T' <= 'a'
True
```

3.5 Les chaînes de caractères

Pour les chaînes de caractères, l'opérateur `<=` correspond à l'ordre *lexicographique*, qui est la généralisation de l'ordre alphabétique utilisé par exemple pour classer les mots dans un dictionnaire.

L'ordre lexicographique consiste à comparer les premiers caractères. En cas d'égalité des premiers caractères on passe aux deuxièmes. Etc …

```
>>> 'Timoleon' <= 'calbuth'
True
```

La chaîne vide est inférieure à toutes les autres chaînes de caractères.

3.6 Les tuples, les listes

Les tuples et les listes sont elles aussi comparables. L'ordre défini par l'opérateur `<=` est aussi l'ordre lexicographique. Bien entendu, il est nécessaire que les éléments de mêmes rangs soient comparables.

```
>>> ('A', 1) <= ('A', 2)
True
>>> ('A', 1, 3) <= ('A', 1)
False
```

Le tuple vide est inférieur à tous les autres tuples. De même pour la liste vide.

3.7 Définir ses propres fonctions de comparaisons

Nous l'avons vu en td, il est parfois nécessaire au programmeur de définir ses propres fonctions de comparaison. C'est le cas s'il veut comparer deux objets d'une même classe.

Pour cela, on peut :

- implanter les méthodes `__lt__` (less than) et `__eq__` (equal) et/ou
- définir une fonction de comparaison.

Rappel

Les fonctions de comparaison que nous concevrons et que nous nommerons `compare` auront donc toutes deux paramètres `x` et `y`, les deux valeurs à comparer, et renverront l'un des trois nombres -1, 0 ou 1 selon la spécification suivante :

- -1 si `x` est plus petit que `y` ;
- 0 si `x` est égal à `y` ;
- 1 si `x` est plus grand que `y`.

:::

Pour cela, il faut que :

- les listes à trier soient *homogènes*, c'est-à-dire que leurs éléments soient tous d'un même type [4], par exemple des listes de nombres, ou de chaînes, ou d'étudiants, de caractères …
- les éléments de cette liste puissent être ordonnés selon un ordre que l'on notera $<$.

Trier une liste c'est obtenir, à partir d'une liste ℓ, une liste contenant les mêmes éléments mais rangés par ordre croissant.

Du point de vue du traitement des données, cette définition n'est pas suffisante :

- doit-on construire une nouvelle liste et laisser la liste ℓ inchangée ? (comme la fonction `sorted`)
- ou bien doit-on transformer la liste de sorte qu'elle soit triée ? (comme la méthode `sort`)

Dans ce cours nous traiterons le second problème. Le premier problème peut aisément être traité en adaptant les algorithmes proposés.

Voici donc la **spécification du problème de tri** que nous nous fixons :

> **Entrée :** ℓ une liste de longueur n, homogène, d'éléments ordonnables.
>
> **Sortie :** Aucune
>
> **Effet de bord :** la liste ℓ est triée.

Nous ne répéterons plus cette spécification du problème dans les algorithmes qui suivront.

[4] En effet, quel sens donner à l'affirmation *cette carotte est plus petite que ce salsifis* ?

4 Tri par sélection

4.1 Sélection du minimum

Commençons par un algorithme permettant de rechercher l'indice d'un élément de valeur minimale dans une tranche de liste.

4.1.1 L'algorithme

Entrée : une liste ℓ de longueur n, homogène, d'éléments ordonnables, et a et b deux indices, $0 \leq a < b \leq n$

Sortie : indice d'un élément minimal de la tranche $\ell[a : b]$.

- $ind_{min} \leftarrow a$
- $i \leftarrow a + 1$
- **Tant que** $i < b$
  - **Si** $\ell[i] < \ell[ind_{min}]$
    - $ind_{min} \leftarrow i$
  - **Fin Si**
- **Fin Tant que**
- **Renvoyer** ind_{min}

4.1.2 Terminaison

On peut prendre par exemple $b - i$ comme variant de boucle.

4.1.3 Correction

Un peut considérer l'invariant
"$\ell[ind_{min}]$ est inférieur ou égal aux éléments de la tranche $\ell[a : i]$"
(en considérant qu'avant la boucle $i = a + 1$)".

4.1.4 Son coût

Notons $c_{\texttt{select}}(k)$ le nombre de comparaisons d'éléments de la liste nécessaires pour trouver le plus petit élément dans une tranche de longueur $k = b - a$.

À chaque étape de la boucle tant que, on effectue une comparaison (ligne 4 de l'algorithme). Le nombre de comparaisons est donc égal au nombre d'itérations de cette boucle. Autrement dit

$$\mathrm{c}_{\mathrm{select}}(k) = \sum_{i=a+1}^{b-1} 1 = (b-1) - (a+1) + 1 = b - a - 1 = k - 1.$$

4.2 Algorithme du tri par sélection

4.2.1 L'algorithme

Tri d'une liste

- **Pour** i variant de 0 à n − 2
  - $ind_{min} \leftarrow \texttt{select}_{\texttt{m}}\texttt{in}(\ell, i, n)$
  - `echanger`$\ell[i]$ et $\ell[ind_{min}]$
- **Fin Pour**

4.2.2 Un exemple

Le tableau ci-dessous montre l'évolution de la tranche triée lors du tri par sélection de la liste des lettres du mot TIMOLEON.

i	ℓ
0	**E** TIMOLON
1	**EI** TMOLON
2	**EIL** TMOON
3	**EILM** TOON
4	**EILMN** TOO
5	**EILMNO** TO
6	**EILMNOOT**

4.2.3 Terminaison

$n - i$ est un variant possible

4.2.4 Correction

Pour démontrer la correction, on peut considérer l'invariant :
"$\ell[a : i]$ est triée ET tous ses éléments sont inférieurs à ceux de $\ell[i : b]$ ET la liste est une permutation de la liste initiale."

4.2.5 Son coût

Notons $c_{\texttt{tri-select}}(n)$ le nombre de comparaisons d'éléments d'une liste de longueur n lors du tri par sélection de cette liste.

L'algorithme du tri par sélection se limitant à une seule boucle pour, le nombre de comparaisons est égal à la somme des nombres de comparaisons effectuées à chaque itération lors de la sélection du minimum.

À l'étape numéro i de cette boucle, on sélectionne le minimum dans une tranche de liste de longueur n − i. On a donc

$$\mathrm{c}_{\mathrm{tri-select}}(n) = \sum_{i=0}^{n-2} \mathrm{c}_{\mathrm{select}}(n-i).$$

Compte-tenu du coût de la sélection établie plus haut, on obtient :

$$\begin{aligned}\mathrm{c}_{\mathrm{tri-select}}(n) &= \sum_{i=0}^{n-2}(n-i-1) \\ &= \frac{n(n-1)}{2} \\ &\sim \frac{n^2}{2}.\end{aligned}$$

Le nombre de comparaisons croît donc quadratiquement (degré 2) avec la longueur de la liste à trier.

Autrement dit, lorsqu'on multiplie la longueur de la liste d'un facteur c, le nombre de comparaisons est multiplié par c^2.

5 Tri par insertion

5.1 Insertion dans une tranche de liste triée

5.1.1 L'algorithme

Entrée : ℓ une liste de longueur n, homogène, d'éléments ordonnables, et $i < n$ un indice, tel que $\ell[0 : i]$ est trié

Sortie : aucune

Effet de bord la tranche $\ell[0 : i + 1]$ est triée.

- $aux \leftarrow \ell[i]$
- $k \leftarrow i$

- **Tant que** $k \geq 1$ et $aux < \ell[k-1]$
  - $\ell[k] \leftarrow \ell[k-1]$
  - $k \leftarrow k-1$
- **Fin TQ**
- $\ell[k] \leftarrow aux$

5.1.2 Terminaison

On peut prendre comme variant de boucle $v(k) = k$

5.1.3 Correction

On peut prendre comme propriété invariante :

"avec $\ell_k = \ell[0:k] + \ell[k+1:]$, ℓ_k est triée ET tous les éléments de $\ell[k+1:]$ sont supérieurs à aux ET $\ell_k + [aux]$ est une permutation de la liste initiale"

5.1.4 Son coût

Notons $c_{\texttt{insert}}(i)$ le nombre de comparaisons d'éléments de la liste nécessaires pour insérer l'élément d'indice i dans la tranche $\ell[0:i+1]$.

Ce nombre de comparaisons ne dépend pas uniquement de l'indice i mais aussi du contenu de la liste.

- Dans le meilleur des cas, une seule comparaison suffit. Cela arrive lorsque l'élément d'indice i est supérieur ou égal à celui qui le précède, et donc est déjà à sa place dans la liste.
- Dans le pire des cas, il faut comparer cet élément à chacun de ceux qui le précède, ce qui arrive lorsqu'il est plus petit que chacun d'eux. Et dans ce cas on effectue i comparaisons.

Ainsi on a

$$1 \leqslant c_{\texttt{insert}}(i) \leqslant i$$

5.2 Le tri

5.2.1 L'algorithme

Tri d'une liste

- **Pour** i variant de 1 à n-1
  - `insert`(ℓ, i)
- **Fin Pour**

Un exemple

Le tableau ci-dessous montre l'évolution de la tranche triée lors du tri par sélection de la liste des lettres du mot TIMOLEON.

étape	ℓ
1	**T** IMOLEON
2	**IT** MOLEON
3	**IMT** OLEON
4	**IMOT**LEON
5	**ILMOT**EON
6	**EILMOT**ON
7	**EILMOOT**N
8	**EILMNOOT**

Son coût

Notons $c_{\texttt{tri-insert}}(n)$ le nombre de comparaisons d'éléments d'une liste de longueur n lors du tri par insertion de cette liste.

Compte-tenu de l'algorithme de ce tri, ce nombre est égal à la somme des comparaisons effectuées pour chacune des insertions :

$$\mathrm{c}_{\mathrm{tri-insert}}(n) = \sum_{i=1}^{n-1} \mathrm{c}_{\mathrm{insert}}(i)$$

À la différence du tri par sélection, le nombre de comparaisons va dépendre du contenu de la liste à trier, et compte-tenu de ce qui a été dit pour le coût de l'insertion, il y a lieu de distinguer deux cas :

- dans le meilleur des cas, la liste est déjà triée, et on est dans le meilleur des cas de l'insertion à chaque étape de la boucle. Et on a ainsi :

$$\mathrm{c}_{\mathrm{tri-insert}}(n) = \sum_{i=1}^{n-1} 1 = n - 1.$$

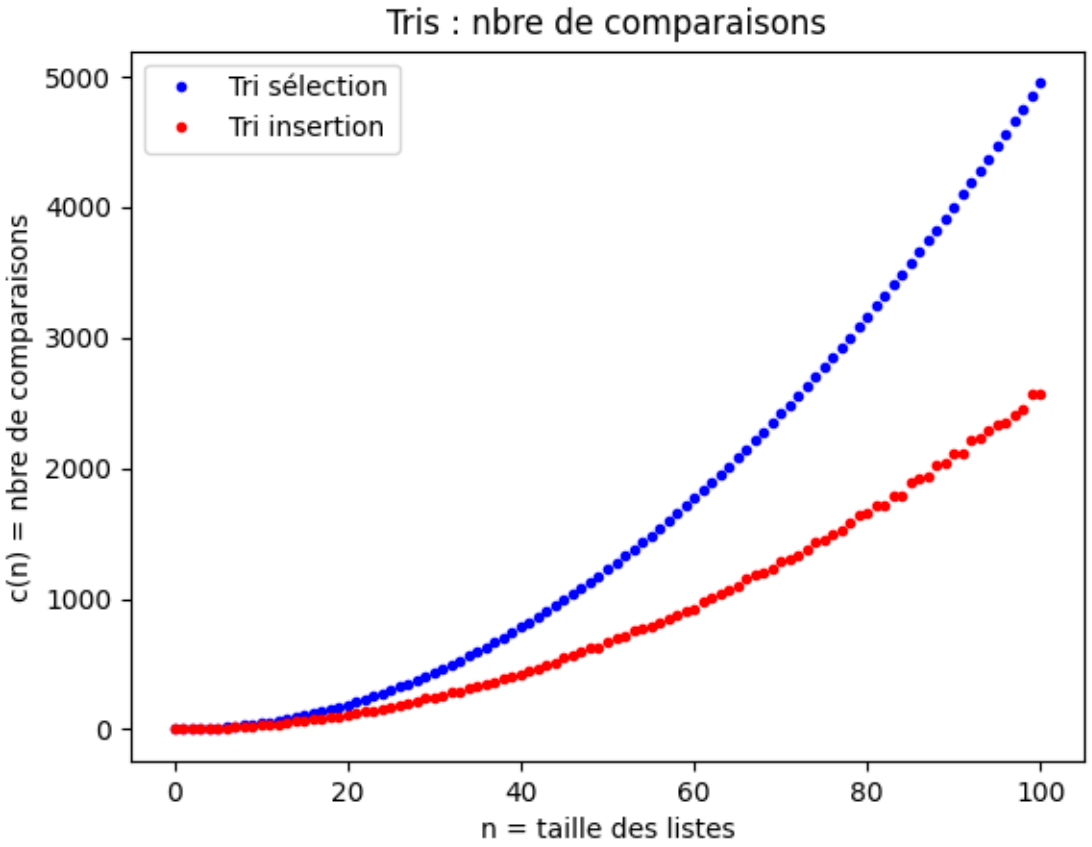
- dans le pire des cas, la liste est triée dans l'ordre inverse et on est alors dans le pire des cas de l'insertion à chaque étape de la boucle. On a alors :

$$\begin{aligned} \mathrm{c}_{\mathrm{tri-insert}}(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \\ &\sim \frac{n^2}{2}. \end{aligned}$$

Ainsi dans le pire des cas, l'algorithme du tri par insertion a le même coût que celui par sélection : il est quadratique en fonction de la longueur de la liste.

Mais dans le meilleur des cas, il est linéaire (degré 1). Le tri par insertion possède de très bonnes performances pour trier des listes *presque* triées.

Qu'en est-il en moyenne pour des listes quelconques ?



La figure ci-dessus montre l'évolution du nombre de comparaisons dans les tris par sélection et par insertion en moyenne.

Le calcul de ce nombre moyen a été obtenu en triant 100 fois par insertion des listes générées aléatoirement avec des longueurs comprises entre 1 et 100 et en totalisant les comparaisons effectuées, puis en divisant par 100 ce total. Pour le tri par sélection, comme le nombre de comparaisons dépend uniquement de la longueur de la liste, il est inutile de moyenner. (On peut d'ailleurs observer que la courbe correspondant au tri par sélection est lisse, alors que celle du tri par insertion ne l'est pas).

On peut constater qu'en moyenne le tri par insertion est deux fois moins coûteux que le tri par sélection. Et, effectivement, on peut prouver qu'en moyenne le nombre de comparaisons du tri par insertion est équivalent à

$$c_{\text{tri-insert}}(n) \sim \frac{n^2}{4}.$$

Il existe des tris plus performants dont le coût en moyenne, ou dans le pire des cas, n'est pas quadratique mais de l'ordre de $n * log(n)$. Citons à titre d'exemple le tri rapide (quicksort) et le tri fusion (mergesort).

Il existe aussi des algorithmes de tri qui n'opèrent pas par comparaison d'éléments.

6 Liens

- Algorithmes de tri en ligne sur le site Interstices
- Tris chorégraphiés
  - tri par sélection
  - tri par insertion

7 Notes