Un cours d'algorithmique pour les gens du TAL

Mathieu Dehouck

April 2, 2026

Ce cours d'algorithmique est en cours de construction. Ainsi il peut contenir des erreurs d'othographe et des imprécisions.

1 Introduction

L'idée de ce cours est de donner une vue d'ensemble des notions principales de l'algorithmiques. Ce cours est destiné en premier lieu aux gens qui font du TAL (NLP), les exemples viendront donc souvent de ce domaine.

L'algorithmique est le domaine de l'informatique théorique et des mathématiques qui s'intéresse à l'étude des suites formelles d'opérations.

La recherche d'élements dans des ensembles (un mot dans une liste de vocabulaire); Le tri d'un ensemble d'éléments; La comparaison de deux ensembles d'éléments...

2 Exemple introductif

Vous faites du TAL. Vous voulez créer un nouveau corpus pour votre prochaine étude. Vous avez collecté des données sur vos sources préférées (internet, grammaires, livres...). Vous avez des dizaines de milliers de phrases (bien trop pour tout lire seul) de longeurs variables. Vous aimeriez avant de commencer vous assurer qu'il n'y a pas de doublons dans votre sélection.

Pour ce faire, vous décidez d'écrire un programme pour comparer vos phrases deux à deux. Pour faciliter le développement de votre programme, vous séléctionner un sous ensemble de 100 phrases.

Vous vous mettez au travail.

```
for ph1 in corpus:
  for ph2 in corpus:
    if ph1 == ph2:
      print('Doublons:', ph1, ph2)
```

Le code ci-dessus compare toutes les paires de phrases, et entre autres les phrases avec elles-mêmes, ce qui renvoie des faux positifs. L'on doit donc s'assurer que si `ph1` et `ph2` sont égales, elles se trouvent à des indices différents.

```
for i, ph1 in enumerate(corpus):
  for j, ph2 in enumerate(corpus):
    if ph1 == ph2 and i != j:
      print('Doublons:', i, ph1, j, ph2)
```

Le nouveau script fonctionne et n'imprime que les vrais doublons. En plus, il termine de manière quasi instantannée. Vous le lancez alors sur votre collection de 500 000 phrases et rien ne semble se passer. Vous ajoutez un compteur de boucle et le verdict tombe. Le programme estime sa durée à 13 heures...

La raison : vous comparez chaque phrase avec toutes les autres, quand vous travailliez sur 100 phrases, cela faisait $100 \times 99 = 9\,900$ comparaisons. Mais avec 500 000 phrases, cela fait $500\,000 \times 499\,999 = 249\,999\,500\,000$ comparaisons soit 25,25 millions de fois plus. La croissance en temps de calcul est quadratique (proportionnelle au carré du nombre de phrases), donc quand on multiplie le nombre de phrases par 5 000, on multiplie le temps de calcul par environ 25 000 000.

On peut s'en doute faire mieux.

D'abord, vous vous rendez compte que vous comparez chaque paire de phrases deux fois, une première fois quand $i > j$ et une seconde quand $i < j$, mais c'est inutile. On peut donc se contenter d'une des deux comparaisons, disons $i < j$. On divise ainsi par deux le temps de calcul nécessaire : 6h30. C'est encore trop long...

Vous réfléchissez un peu et une nouvelle idée vous apparaît : "Si deux phrases sont identiques, alors elles font la même longueur." Autrement dit, il est inutile de comparer des phrases de longueurs différentes.

Pour regrouper les phrases par longueurs, il suffit de les lire une fois chacune. Donc le temps reste proportionnel au nombre de phrase et à leur longueur totale.

Dans votre collection, les phrases ont des longueurs allant de 3 à 150 mots. La longueur la plus fréquente est 18 mots et elle représente 4% des phrases de votre collection. En regroupant les phrases par longueur, le plus gros groupe de phrases fait maintenant environ 20 000, et les autres groupes font en moyenne moins de 5 000 phrases. Le plus gros groupe nécessite 625 fois moins de comparaisons que lorsque l'on compare toutes les phrases les unes avec les autres. Même si l'on doit faire des comparaisons pour environ 150 groupes de phrases, avec un plus grand groupe à 4% on s'attend à une réduction du temps de calcul d'au moins 25 fois, pour une durée de calcul d'au maximum 16 minutes (une grosse pause).

Dans les faits, on peut faire encore mieux. En effet, la longueur n'est qu'un critère possible pour regrouper les phrases ensemble, et d'autres critères mieux choisis pourraient mieux séparer les phrases. Mais on verra ça plus loin dans ce cours.

Le reste de ce cours va explorer différents problèmes, plus ou moins liés au

traitement du langage, mais toujours applicables à celui-ci avec un peu d'adaptation.

3 La Complexité

La complexité d'un algorithme est une représentation des ressources nécessaires à son execution.

Les deux ressources principales disponibles sur un ordinateur étant la mémoire et le processeur, l'on peut s'intéresser à la complexité spaciale (la quantité de mémoire nécessaire pour faire tourner un algorithme) ainsi qu'à la complexité temporelle (le nombre d'opération à exécuter sur le CPU/GPU). Dans ce cours on s'itéressera avant tout à la complexité temporelle qui est la plus importante pour la majorité des algorithmes courants.

Pour la culture, il est bon de savoir qu'il existe d'autres formes de complexité, même si elles ne sont pas toujours appelées ainsi. Pour les applications distribuées/décentralisées dont les opérations sont réparties sur plusieurs machines partangeant un réseau de communication (LAN, Internet...), chaque échange entre deux machines prend du temps et peut limiter l'accès des autres machines au réseau qui devient donc une ressource à part entière. Les specialistes des algorithmes distribués s'intéressent donc aussi au nombre d'échanges nécéssaires pour réaliser une tâche entre plusieurs machines.

> La complexité et l'agorithmique sont importantes quand on travaille avec des grands objets ou quand on répête certaines opérations de nombreuses fois.

3.1 Trade Off

Information/Structure vs Complexité

3.2 La notation en O

Le temps de complétion d'un algorithme dépend d'énormément de facteurs. Il dépend bien entendu de la complexité de l'algorithme et de la quantité de données auquelle, mais également de la qualité de son implémentation, du type de langage de programation qui le fait tourner[1] (compilé ou interprété, machine ou byte-code...) et du nombre de vérifications de sécurité effectuées pendant son exécution. Il dépend également de la vitesse du processeur sur lequel il est exécuté, du type de mémoire vive, de la carte mère, du sysème d'exploitation qui tourne dessus, et des des autres programmes qui tournent dessus en même temps.

Ainsi un même algorithme, appliqué aux mêmes données, peut tourner 100 fois plus vite si écrit en C compilé pour une machine moderne, ou interprété en Python sur une machine vieille de 20 ans. Le temps d'exécution est donc assez peu révélateur de la nature d'un algorithme. Par contre, sa complexité nous renseigne sur son comportement quand on va lui donner plus de données à traiter.

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

Peu importe la machine, l'implémentation et le langage de programmation utilisé, on peut s'attendre à ce que le temps de calcul nécessaire à un algorithme quadratique (proportionnel au carré de la taille de son entrée) soit multipliée par 4 quand on multiplie la taille de son entrée par 2.

C'est ce que représente la notation en O (prononcé "grand O").

Un algorithme qui prend un temps proportionnel à la taille n de son entrée pour s'executer est dit *linéaire* et on note sa complexité $O(n)$. Des exemples d'algorithmes linéaires typiques sont la recherche du maximum ou du minimum dans un jeu de données, le calcul de la somme ou de la moyenne d'un jeu de données. Notons que pour normaliser un jeu de données, il faut le lire deux fois, une fois pour calculer la somme de ses valeurs et une seconde fois pour diviser les valeurs d'entrée par la somme, ainsi la le temps d'execution sera proportionnel à $2n$. Mais comme $2n$ évolue de la même manière que n face à l'augmentation de la taille des données, $2n$ est linéaire et on note aussi $2n = O(n)$.

Pour s'en convaincre, soit k le rapport d'augmentation de la taille de l'entrée entre deux exécutions d'un algorithme. L'augmentation du temps de calcul est $\frac{kn}{n} = k = \frac{2kn}{2n}$, pour un algorithme dont la complexité est proportionnelle à n comme à $2n$.

La comparaison de deux jeux de données non triés a une complexité, quadratique, en $O(n^2)$ comme celle de certains algorithmes de tri de listes.

Les complexités s'organisent comme suit:

$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(2^n).$$

4 La Multiplication

Le premier problème a très peu avoir avec le TAL, mais est très utile pour réfléchir à la manière de penser les algorithmes.

Soient deux nombres entiers a et b ($(a, b) \in \mathbb{N}^2$). On souhaite calculer le produit $a \times b$ en n'utilisant que l'addition $+$, et en en utilisant le moins possible.

4.1 Première idée : répéter l'addition

Par définition : $a \times b = a + a + a + a + ... + a \ \ (b \ fois)$.
Par exemple, $1 \times a = a$, $2 \times a = a + a$, $3 \times a = a + a + a$ et ainsi de suite. On voit donc que l'on peut calculer $a \times b$ en $b - 1$ additions.

Ça peut paraître bien de prime abord, mais en fait c'est très long. Si on veut multiplier un nombre par 1 000 000, on doit faire 999 999 additions...

Mais comment faire mieux ?

4.2 a+a = 2a

En une addition, l'on peut calculer $2a = a + a$.

Si b est un nombre pair, alors $b = 2 \times c$, sinon $b = 2 \times c + 1$. On notera ± 1 le fait que b soit égal à $2c$ ou $2c + 1$. Donc, $a \times b = a \times (2 \times c \pm 1) = 2 \times a \times c \pm a$, d'où $a \times b = 2a + 2a + ... + 2a \ (c \ fois) \pm a$. Or, c est égale à $b/2$ (à plus ou moins 0,5 près). On a donc divisé le nombre d'additions nécéssaires par deux, pour le coup d'une addition en plus.

Si on veut multiplier un nombre par 1 000 000, après avoir fait une addition $2a = a + a$, on n'a plus besoin que d'en faire 499 999... Mais pourquoi s'arrêter là ?

Avec une deuxième addition, on peut avoir $4a = 2a + 2a$, et on applique le même principe que précédement $2a \times c$. Soit $c = 2 \times d \pm 1$, donc $2a \times c = 2a \times (2 \times d \pm 1) = 4a \times d \pm 2a$, il reste donc $d - 1$ additions à faire, soit $b/4$ (a plus ou moins 0,75 près).

4.3 Généralisation et exemple

Si on répéte la même idée encore et encore, on peut pour le coût d'une addition à chaque fois diviser le nombre d'additions nécessaires par deux, jusqu'à ce qu'il faille n'en faire plus qu'une.

Exemple : $487 \times a$.

$$
\begin{array}{rcrll}
487 = 2 \times 243 + 1 & \rightarrow & 487 \times a & = 243 \times 2a + a; & 2a = a + a \\
243 = 2 \times 121 + 1 & \rightarrow & 243 \times 2a & = 121 \times 4a + 2a; & 4a = 2a + 2a \\
121 = 2 \times 60 + 1 & \rightarrow & 121 \times 4a & = 60 \times 8a + 4a; & 8a = 4a + 4a \\
60 = 2 \times 30 & \rightarrow & 60 \times 8a & = 30 \times 16a; & 16a = 8a + 8a \\
30 = 2 \times 15 & \rightarrow & 30 \times 16a & = 15 \times 32a; & 32a = 16a + 16a \\
15 = 2 \times 7 + 1 & \rightarrow & 15 \times 32a & = 7 \times 64a + 32a; & 64a = 32a + 32a \\
7 = 2 \times 3 + 1 & \rightarrow & 7 \times 64a & = 3 \times 128a + 64a; & 128a = 64a + 64a \\
3 = 2 \times 1 + 1 & \rightarrow & 3 \times 128a & = 1 \times 256a + 128a; & 256a = 128a + 128a.
\end{array}
$$

En remettant tout bout à bout, on a :

$$487 \times a = 256a + 128a + 64a + 32a + 4a + 2a + a.$$

On a ainsi calculé $487 \times a$ à l'aide de 14 additions seulement. Notons que $487 = 256 + 128 + 64 + 32 + 4 + 2 + 1$, en binaire (base 2) la représentation de 487 est donc 111100111.

Comme on l'a vu plus haut, pour additioner deux nombres, il suffit d'une addition, et pour n nombres, $n - 1$ additions suffisent. Ainsi, les 14 additions nécessaires à la multiplication par 487 sont les $8 = 9 - 1$ correspondant à la longueur de la représentation de 487 en binaire et les $6 = 7 - 1$ corresondant aux 1 dans cette-même représentation.

De mannière générale, on pourra calculer la multiplication par b en autant d'additions que la somme de la longueure de la représentation de b en binaire et du nombre de 1 dans cette représentation, moins deux. De plus, comme le nombre de 1 dans la représentation binaire d'un nombre, sera au plus égal à la longueur de ce nombre (on ne peut pas mettre dix 1 dans un nombre de longueur 7), le nombre d'additions nécessaires sera toujours au maximum égal à deux fois la longueur du nombre moins deux.

On pourra se poser la question de la pertinence de remplacer la multiplication par une série d'additions, sachant que pour ce faire, on a eu recours à la **division** par deux de nos facteurs.
En fait, du à la manière dont les nombres entiers sont représentés dans la mémoire de l'ordinateur, il est beaucoup plus facile de multiplier et/ou diviser par des puissances de deux, et a fortiori par deux, que par n'importe quelles autres valeurs.
De la même manière, qu'en décimale, pour multiplier par 10, il suffit de rajouter un zéro après le nombre ($10 * 512 = 512\mathbf{0}$), et pour diviser par 10 on enlève un 0 ou on décalle une virgule d'une position, en binaire, la multiplication/division par 2 consiste en le décallage de la représentation d'un bit vers la gauche ou la droite. Cette opération, appelée **bit shift**, est bien plus facile à réaliser pour une machine qu'une vraie division/multiplication.

4.4 Conclusion

On a vu comment réimplémenter la multiplication par $b \in \mathbb{N}$ en utilisant un nombre limité ($\leq 2 \times (\text{len}(b) - 1)$) d'additions.

Bien que ce premier problème est assez loin de ceux rencontrés dans le TAL, il est très représentatif de ce que l'algorithmique permet de faire. Avec un peut d'analyse du problème qui nous intéresse, il est souvent possible de le décomposer de manière à économiser beaucoup de temps.

4.5 Aller plus loin

Comme on l'a vu plus haut, une manière de réduire drastiquement le nombre d'additions nécessaire pour réaliser un multiplication est de suivre la représentation binaire d'un des deux facteurs. Les ordinateurs utilisent surtout le binaire, mais pour facilité la lecture par les humains, l'octal (base $8 = 2^3$) et l'héxadécimal (base $16 = 2^4$) sont couramment utilisées aussi. Le standard Unicode par exemple utilise des codes héxadécimaux, ainsi le A majuscule latin a pour code 0041, mais le k minuscule latin a pour code 006B. Le caractère tsalagi 𐊫 a pour code 13AD.

On pourra essayer d'écrire des scripts pour passer d'une base à l'autre.

5 Trouver un élément dans une liste

Dans cette section, nous nous intéressons à la question de la présence d'un élément dans une liste d'éléments de même type. Dans le TAL, c'est une question qui arrive souvent : un mot est-il présent dans une liste de vocabulaire ou dans une phrase, une phrase est-elle présente dans un corpus, l'alphabet de telle ou telle langue utilise-t-il une certain caractère ?

Soit une liste l de longueur $n \in \mathbb{N}$ et soit un objet x. En principe x et l peuvent être de n'importe quel type, du moment que l'égalité est bien définie sur ce type. Ce sera le cas pour l'énorme majorité des objets que l'on pourra vouloir manipuler en TAL.

Pour le moment, on considérera que l est une liste d'entier et x est un entier aussi. On revisitera plus tard la question de savoir si un élément est dans une liste pour des objets plus complexes plus loin.

5.1 La liste non triée

Si la liste l de longueur $n \in \mathbb{N}$ n'est pas triée, de combien de temps avons-nous besoin pour savoir si x est dedans ?

Dans ce cas, il est assez facile de voir que dans le pire des cas (x n'est pas dans la liste), il faudra parcourir toute la liste pour s'en rendre compte. Cela donne donc une complexité en $O(n)$

```
for i in range(len(l)):
  if l[i] == x:
    print("Trouvé à l'index", i)
```

5.2 La liste triée

Si maintenant on concidère que l est triée par ordre croissant (avec la même relation d'ordre que celle notée par $<$ et $>$[2]). On va pouvoir utiliser cette information supplémentaire pour accélérer la recherche.

Si x est plus petit que le premier élément $l[0]$ de la liste, alors on sait que x n'est pas dans celle-ci car comme elle est triée par ordre croissant, tous les éléments venant après $l[0]$ sont plus grands que $l[0]$ et donc différents de x. Il en est de même si $l[n-1] < x$. On peut ensuite vérifier si x est égale à $l[0]$ ou $l[n-1]$, en une comparaison à chaque fois.

Notez que grâce au tri, quand x est plus grand que le maximum de l ou plus petit que le minimum, on sait que x n'est pas dans la liste en un nombre constant d'étapes (1 ou 2). Alors que quand la liste n'est pas triée, il fallait n étapes.

Si x est en dehors des bornes de l ou si x est égale à une des bornes ($l[0]$ ou $l[n-1]$), on le sait en au plus 4 comparaisons. Sinon, alors l'on sait que x si elle

[2]Pour plus de détail sur la notion de relation d'ordre et son importance en TAL, voir l'encadré dans la section suivante.

se trouve dans l doit se trouver quelque part entre les indices 1 et $n-2$, mais où exactement ?

Sans plus de connaissance sur la distribution des valeurs dans l, la meilleure chose à faire est de vérifier où se situe x par rapport aux valeurs du milieu de la liste. Si la taille n de la liste est impaire, alors il y a un élément exactement au milieu, mais si n est pair alors il faut choisir un des deux éléments centraux. Notons que le plus important est d'être cohérent quant à ce choix, mais comme les programmes sont facilement déterministes, il est assez naturel d'être cohérent.

Soit $c = l\left[\left\lfloor \frac{n}{2} \right\rfloor\right]$ l'élément central de la liste. Si x est égale à c, alors on a trouvé x dans l. Sinon, soit x est plus grand que c ou plus petit. Sans perte de généralité, supposons que $x < c$. Comme l est triée, nous savons que tous les éléments de l au delà de l'index $\left\lfloor \frac{n}{2} \right\rfloor$ (au delà de c) sont plus grand que c, et donc sont nécessairement différents de x, on peut donc les ignorer. Si x est présente dans l, alors elle se trouve entre les indices 0 et $\left\lfloor \frac{n}{2} \right\rfloor$. On peut donc répéter la même procédure sur la demi liste restante, jusqu'à ce que l'on ait trouvé x ou que les bornes inférieure et supérieure des indices de la liste où pourrait se trouver x soit des entiers succéssifs.

Notons a et b les bornes inférieure et supérieure de la zone où pourrait se trouver x dans l. On sait que, $l[a] < x < l[b]$. Si $b = a + 1$, alors il n'y a plus d'élément entre $l[a]$ et $l[b]$ et donc, x n'est pas dans l.

On appelle cette méthode de recherche, la recherche par **dichotomie**.

```
a = 0
b = len(l) - 1

if x < l[a] or x > l[b]:
  print("x n'est pas dans la liste")
  return
if x == l[a]:
  print("Trouvé à l'index", a)
  return a
if x == l[b]:
  print("Trouvé à l'index", b)
  return b

while a != b-1:
  m = (a + b) // 2
  if l[m] == x:
    print("Trouvé à l'index", m)
    return m
  if l[m] < x:
    a = m
  else:
    b = m
```

La complexité de la recherche par dichotomie est $O(\log_2(n))$.

Notons temps d'exécution de la recherche (le nombre de comparaisons) sur

une liste de longueur n par $T(n)$. Comme à chaque tour de boucle, après deux comparaisons, on divise l'espace de recherche restant par deux, on obtient : $T(n) = 2 + T(\frac{n}{2})$.

Supposons que la longueur soit une puissance de 2 : $n = 2^k, k \in \mathbb{N}$, alors on a :

$$\begin{aligned} T(n) &= T(2^k) \\ &= 2 + T(\frac{2^k}{2}) \\ &= 2 + T(2^{k-1}) \\ &= 2 + 2 + T(\frac{2^{k-1}}{2}) \\ &= 2 \times 2 + T(2^{k-2}) \\ &= a \times 2 + T(2^{k-a}). \end{aligned}$$

On atteint $T(1) = \text{cst}$ quand $a = k$, on a alors : $T(2^k) = 2k + \text{cst}$, or $k = \log_2(2^k) = \log_2(n)$. On a donc bien $T(n) = O(\log_2(n))$.

5.3 Conclusion

On verra comment on peut espérer aller plus vite quand on a plus d'information sur la structure des objets de l plus loin.

6 Tris

On a vu dans le chapitre précédent comment le fait d'utiliser une liste triée plutôt qu'une liste d'éléments sans ordre particulier, permet de grandement accélérer la recherche d'élément. La question est alors, en combien de temps peut-on trier une liste ?

6.1 Comment créer un algorithme ?

Avant de parler spécifiquement de tris, nous allons voir comment quelqu'un (vous) peut créer des algorithmes par lui-même.

L'idée au cœur de la création des algorithmes est que les algorithmes prennent des données en entrées et les manipules d'une manière ou d'une autre avant d'effectuer des sorties.

Les sorties des algorithmes peuvent prendre des formes très variées :

- données utilisable dans le programme courant (dans le script Python),
- texte à imprimer dans la console,
- actions à réaliser dans un navigateur web,
- images à afficher à l'écran,
- son à jouer dans un casque ou des haut-parleurs,
- vibrations dans une manette de jeu ou dans un téléphone
- bytes directement dans la mémoire vive...

Mais tous* les algorithmes produisent des sorties. C'est en effet leur moyen d'interagir avec le monde.

*On pourrait argumenter que les algorithmes d'attente (`sleep`) ne produisent pas de sortie, leur rôle étant simplement de mettre les calculs en pause pendant un certain temps. On pourrait aussi argumenter en faveur du fait de considérer le temps d'attente justement comme leur sortie, même si c'est alors une sortie d'un type très particulier.

Quand la sortie correspond à une copie de l'entrée à laquelle on a ajouté de la structure, pour créer un algorithme, il faut identifier les propriétés caractéristiques de la structure souhaitée. Dans le cas d'une liste triée, si on note $<$ la relation d'ordre canonique entre deux éléments de l, plusieurs propriétés équivalentes sont :

- $l[0] = \min(l)$, (le premier élément de l est aussi le plus petit),
- $l[n-1] = \max(l)$, (le dernier élément de l est aussi le plus grand),

- $\forall i \in \mathbb{N},\quad 0 \leq i < n-1,\quad l[i] \leq l[i+1]$, (pour chaque paire d'éléments consécutifs, celui de gauche est inférieur ou égal à celui de droite),
- $\forall (i,j) \in \mathbb{N}^2,\quad 0 \leq i < j < n,\quad l[i] \leq l[j]$, (pour chaque paire d'élément, celui de gauche est inférieur ou égal à celui de droite).

Toutes ces propriétés sont équivalentes et de fait, elles définissent une liste triée. On pourrait en trouver encore d'autres, et chaque propriété nous conduit à créer des algorithmes différents.

6.2 Selection Sort - Tri par sélection

Prenons par exemple la première propriété : $l[0] = \min(l)$, (le premier élément de l est aussi le plus petit). Nous pouvons en dériver l'algorithme suivant:

```
sorted_l = []
for i in range(len(l)):
  m = min(l) # m est le minimum de l
  sorted_l.append(m) # on l'ajoute à la version triée de l
  l.remove(m) # on l'enlève de l
```

A chaque itération, on cherche le plus petit élément de la liste l, on l'enlève de l pour éviter de le reprendre le tour suivant, et on l'ajoute dans la version triée de l. Après le i-ème tour de boucle, *sorted_l* contient les i plus petits éléments de l triés par ordre croissant.

La complexité de cet algorithme de tri est $O(n^2)$, en effet, il faut parcourir la liste en entier en $O(n)$ pour trouver sont plus petit élément, et il y a n éléments qui vont être tour à tour le plus petit. On a donc $O(n) \times O(n) = O(n^2)$.

Le tri par sélection peut-être implémenter en utilisant le maximum au lieu du minimum.

On a vu plus haut que chercher un élément dans une liste non triée prenait un temps proportionnel à n alors que le temps nécessaire pour chercher dans liste triée est proportionnel à $\log_2(n)$.
Si le temps nécessaire pour trier une liste est proportionnel à n^2, est-il vraiment utile de triée une liste ?
L'on a juste besoin de trier la liste une fois avant d'effectuer autant de recherche que l'on veut. Donc, si l'on fait plus de $O(n)$ recherches*, on commence à économiser du temps.
*Le nombre de recherche à partir duquel on commence à économiser du temps dépend fortement de la complexité de l'algorithme de tri. On verra plus tard un algorithme de tri qui tourne en $O(n \log_2(n))$ étapes. Avec un tel algorithme, on commence à économiser du temps de calcul beaucoup plus vite (après seulement $O(\log_2(n))$ recherches).

6.3 Conclusion

7 Récursion

La récursion n'est pas un algorithme, mais une façon de penser et une méthode de création d'algorithmes.

L'idée derrière les algorithmes récursifs est de décomposer les problèmes pour se retrouver dans des cas plus simples à résoudre.

Prenons un exemple simple. Soit x une chaîne de caractères. On aimerait savoir si x est un palindrome (une chaîne de caractères qui se lit de la même manière dans les deux sens). Quels sont les cas où il est le plus facile de répondre à cette question ?

- Si x est la chaîne vide, alors x est autant la chaîne vide de gauche à droite que de droite à gauche et donc x est bien un palindrome;
- Si x ne contient qu'un caractère, alors une fois encore, x se lit de la même manière dans les deux sens, x est bien un palindrome.

Si maintenant, x contient plus de caractères, que faire ? Par définition, un palindrome se lit de la même manière dans les deux sens, et donc il doit contenir les mêmes lettres dans les deux sens, ce qui signifie qu'au minimum, le premier et le dernier caractères doivent être les mêmes. Si ce n'est pas le cas, alors x n'est pas un palindrome, et il n'est pas nécessaire d'aller plus loin. Si par contre $x[0] = x[-1]$ alors, il reste à vérifier que l'intérieur de x est également un palindrome. Mais notons qu'en se débarrassant du premier et du dernier caractères, dont nous venons de vérifier l'égalité, la nouvelle chaîne que nous devons manipuler est strictement plus courte (de deux caractères).

En répétant cette procédure encore et encore, quatre cas sont possibles:

- Si x est la chaîne vide, alors x est un palindrome;
- Si x ne contient qu'un caractère, alors x est un palindrome;
- Si $x[0] \neq x[-1]$, alors x n'est pas un palindrome;
- Si $x[0] = x[-1]$, alors il reste à vérifier que $x[1:-1]$ est un palindrome.

La programmation récursive se base essentiellement sur cette idée. Un problème peut être soit déjà facile a résoudre, ou alors il est possible de le décomposer de tel sorte à ce que les sous problèmes soient eux plus faciles à résoudre, car plus courts ou mieux structurés.

7.1 Merge Sort - Un tri recursif

Le merge sort ou tri par fusion est un algorithme de tri récursif qui se base sur deux propriétés des listes triées :

- Soit l une liste triée, toute sous-liste l' de l est triée, et son complément $\bar{l'} = \{e | e \in l, e \notin l'\}$ (la liste des éléments de l qui ne sont pas dans l') est aussi trié, et on peut fusionner les deux facilement;

- Une liste de longueur $n = 1$ est trivialement triée.

```
def fuse(l1, l2):
  # une fonction récursive pour fusionner deux listes triées

  if l1 == []:
    return l2

  if l2 == []:
    return l1

  # si on est ici, alors ni l1 ni l2 ne sont vides
  if l1[0] < l2[0]:
    return [l1[0]] + fuse(l1[1:], l2)
  else:
    return [l2[0]] + fuse(l1, l2[1:])


def merge_sort(lst):
  if len(lst) == 1: # une liste de longueur 1 est déjà triée
    return lst

  else:
    # on doit couper lst en deux sous-listes
    l1 = []
    l2 = []
    i = 0
    for e in lst:
      if i == 0:
        l1.append(e)
      else:
        l2.append(e)
      i = 1 - i

    # on trie l1 et l2 avec merge_sort de manière récursive
    l1 = merge_sort(l1)
    l2 = merge_sort(l2)

    # maintenant on doit fusionner l1 et l2, mais elles sont triées
    l_srt = fuse(l1, l2)

    return l_srt
```

Le tri par fusion marche comme suit: si la liste à trier est de longueur 1, alors elle est déjà triée et donc on peut la retourner directement. Si au contraire, elle

contient plus d'un élément, alors on la coupe en deux sous-listes de tailles plus ou moins égales (à un élément prés) que l'on va trier puis fusionner. Comme les deux sous-listes sont deux fois moins grandes que la liste de départ, le problème du tri de chacune est strictement plus simple. Ensuite, il suffit de les fusionner de telle sorte à ce que la liste finale soit triée. Pour ce faire, on compare le premier élément de chaque sous-liste triée, c'est aussi le plus petit car elles sont triées, et on choisit le plus petit des deux, on répête jusqu'à ce que l'une des deux listes soit vides.

7.2 Conclusion

8 Comparaison de chaîne de caractères

9 Hachage