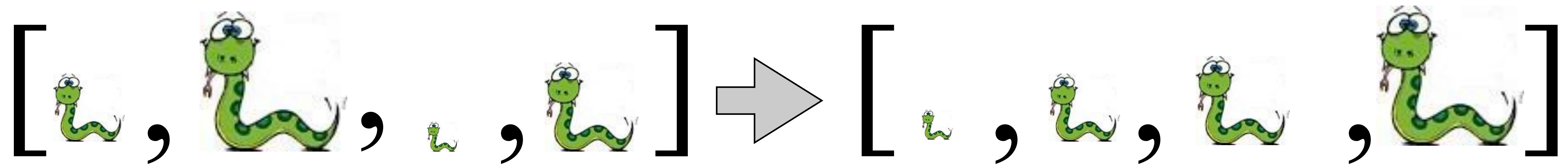
Université Nice Sophia Antipolis
Faculté des Sciences

Programmation avec Python
Etienne Lozes (remerciements: Jean-Paul Roy)
L1-Sciences 2019

Cours n°9

http://deptinfo.unice.fr/~elozes

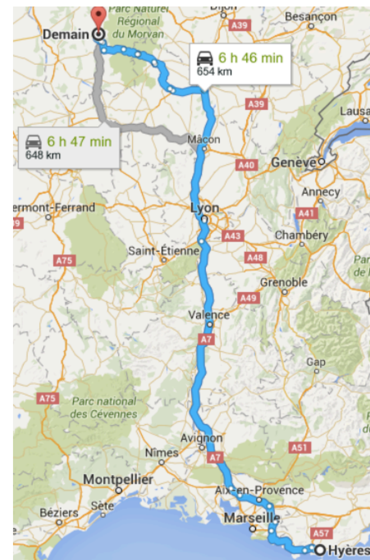
Algorithmes de recherche et de tri



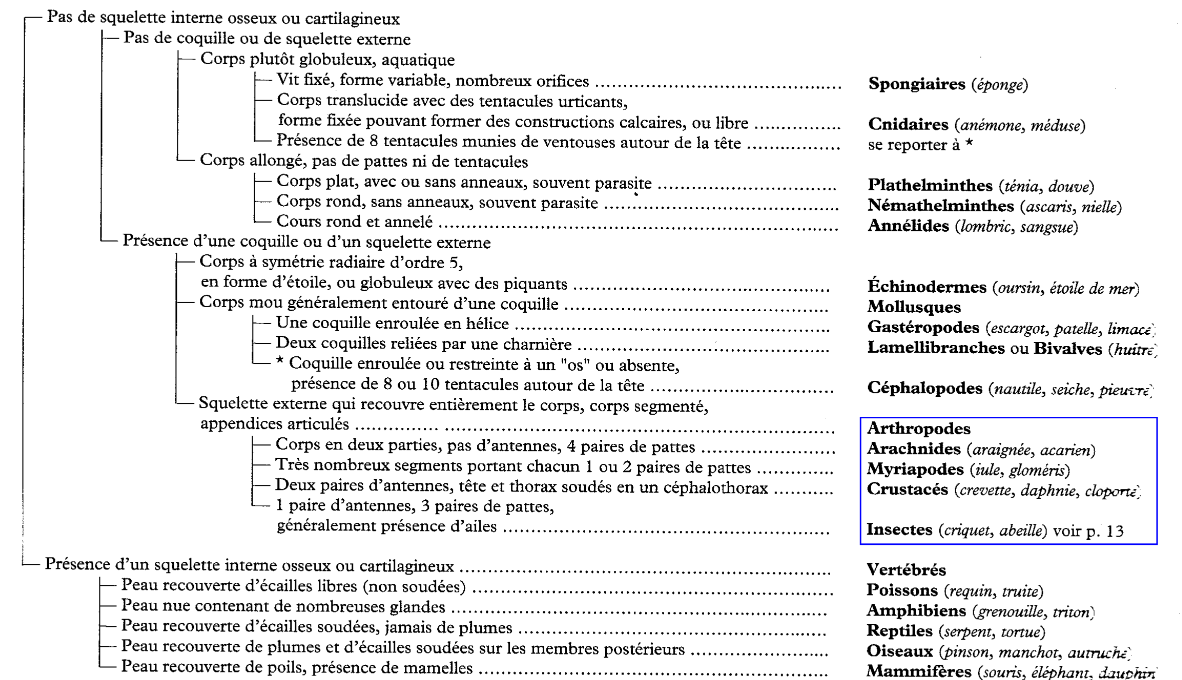
Qu'est-ce qu'un algorithme?

un algorithme est une description non ambiguë, destinée aux humains, d'un procédé permettant de réaliser une tâche à partir d'opérations élémentaires

1. Prendre la direction ouest sur Rue des Porches vers Rue Portalet
2. Prendre à droite sur Rue Portalet
3. Prendre à gauche sur Place Massillon
4. Prendre légèrement à droite sur Rue du Vieux Cimetière
5. Prendre à droite sur Rue Sainte-Catherine
6. Prendre à gauche sur Place Saint-Paul
7. Place Saint-Paul tourne à droite et devient Rue de la Croix
8. Prendre à droite sur Montée Sainte-Croix
9. Prendre à droite sur Avenue Edith Wharton
10. Au rond-point, prendre la 2e sortie sur Rue Seré de Rivières
11. Prendre à gauche sur Avenue Alexis Godillot/D554
    Traverser le rond-point



des suites d'instructions



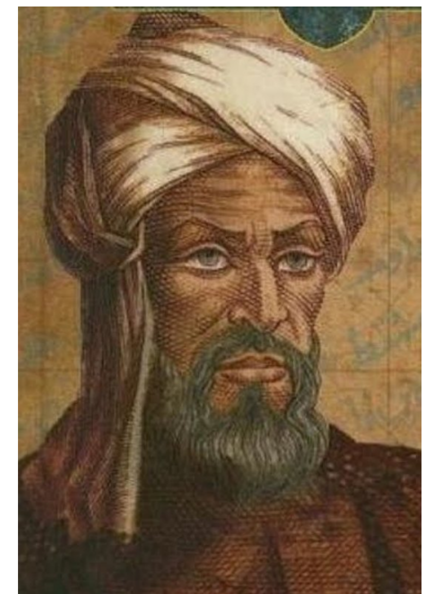
1. Une clé de détermination simplifiée des grands groupes d'animaux

- Pas de squelette interne osseux ou cartilagineux
  - Pas de coquille ou de squelette externe
    - Corps plutôt globuleux, aquatique
      - Vit fixé, forme variable, nombreux orifices .... **Spongiaires** (*éponge*)
      - Corps translucide avec des tentacules urticants, forme fixée pouvant former des constructions calcaires, ou libre .... **Cnidaires** (*anémone, méduse*)
      - Présence de 8 tentacules munies de ventouses autour de la tête .... se reporter à *
    - Corps allongé, pas de pattes ni de tentacules
      - Corps plat, avec ou sans anneaux, souvent parasite .... **Plathelminthes** (*ténia, douve*)
      - Corps rond, sans anneaux, souvent parasite .... **Némathelminthes** (*ascaris, nielle*)
      - Cours rond et annelé .... **Annélides** (*lombric, sangsue*)
  - Présence d'une coquille ou d'un squelette externe
    - Corps à symétrie radiaire d'ordre 5, en forme d'étoile, ou globuleux avec des piquants .... **Échinodermes** (*oursin, étoile de mer*)
    - Corps mou généralement entouré d'une coquille .... **Mollusques**
      - Une coquille enroulée en hélice .... **Gastéropodes** (*escargot, patelle, limace*)
      - Deux coquilles reliées par une charnière .... **Lamellibranches** ou **Bivalves** (*huître*)
      - * Coquille enroulée ou restreinte à un "os" ou absente, présence de 8 ou 10 tentacules autour de la tête .... **Céphalopodes** (*nautile, seiche, pieuvre*)
    - Squelette externe qui recouvre entièrement le corps, corps segmenté, appendices articulés .... **Arthropodes**
      - Corps en deux parties, pas d'antennes, 4 paires de pattes .... **Arachnides** (*araignée, acarien*)
      - Très nombreux segments portant chacun 1 ou 2 paires de pattes .... **Myriapodes** (*iule, gloméris*)
      - Deux paires d'antennes, tête et thorax soudés en un céphalothorax .... **Crustacés** (*crevette, daphnie, cloporte*)
      - 1 paire d'antennes, 3 paires de pattes, généralement présence d'ailes .... **Insectes** (*criquet, abeille*) voir p. 13
- Présence d'un squelette interne osseux ou cartilagineux .... **Vertébrés**
  - Peau recouverte d'écailles libres (non soudées) .... **Poissons** (*requin, truite*)
  - Peau nue contenant de nombreuses glandes .... **Amphibiens** (*grenouille, triton*)
  - Peau recouverte d'écailles soudées, jamais de plumes .... **Reptiles** (*serpent, tortue*)
  - Peau recouverte de plumes et d'écailles soudées sur les membres postérieurs .... **Oiseaux** (*pinson, manchot, autruche*)
  - Peau recouverte de poils, présence de mamelles .... **Mammifères** (*souris, éléphant, dauphin*)

des tests et des prises de décision

Qu'est-ce qu'un algorithme?

mais aussi :
des boucles, du calcul en parallèle, des échanges de messages, de la tolérance aux pannes,...

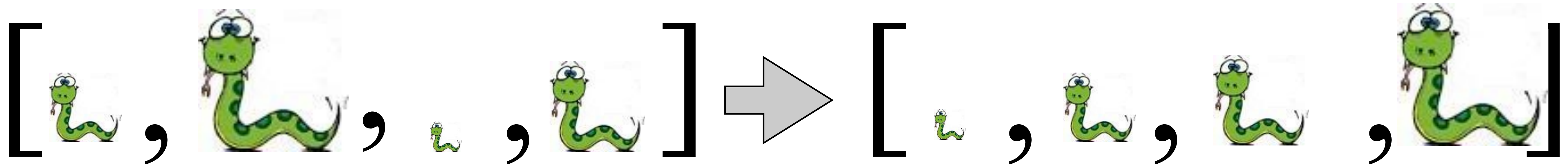


Le mot algorithme dérive du nom du mathématicien perse Al Khwarizmi. Son kitab al-mukhtasar fi hisab al-jabr wa-l-muqabala a donné le mot algèbre.

Algèbre signifie réduction. Un algorithme peut simplifier ou transformer un problème de sorte que l'on se ramène à un problème que l'on sait résoudre.

La notion de réduction est au coeur de la théorie de la complexité algorithmique.

1.
Algorithmes de tri



Un tri est-il performant ?

- Le tri prédéfini `sort` de Python est très efficace. Pour trier une liste aléatoire de n éléments. il a un « coût » en $O(n \log(n))$...

- Le tri par sélection (cours 5) est peu efficace. Pour trier une liste de n éléments, il a un coût proportionnel à n^2...

n	n log(n)	n^2
1000	6907	1000000
2000	15201	4000000

- Le programmeur s'efforcera toujours de minimiser le coût :

  n^2 est meilleur que 2^n

  n est meilleur que n log(n) qui est meilleur que n^2

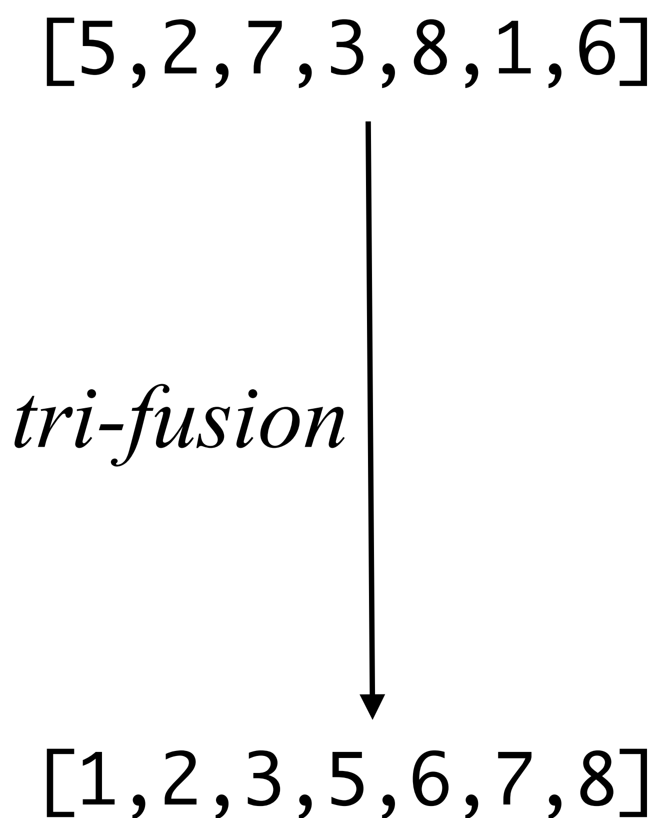
  log(n) est meilleur que n

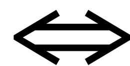
  1 = Cte est meilleur que log(n)

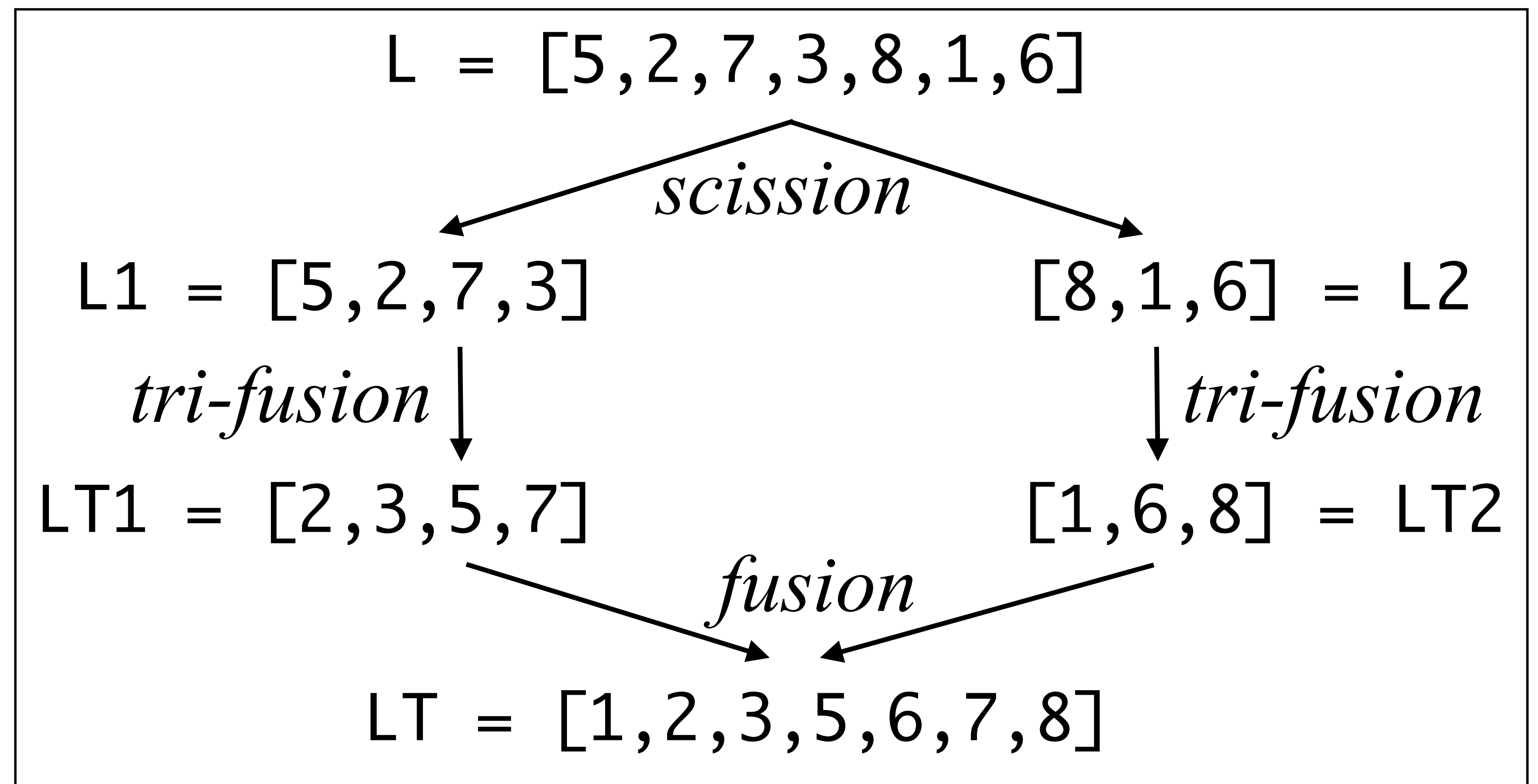
Le tri d'une liste par fusion

Le **tri fusion** est un premier exemple d'algorithme de type **DIVISER POUR REGNER :**

- je commence par scinder la liste L en deux sous-listes L1 et L2 de même longueur, ou presque.
- je trie récursivement L1 et L2, pour obtenir LT1 et LT2.
- je fusionne LT1 et LT2 en une seule liste triée.







```
def tri_fusion(L) :
  if len(L) < 2 : return L
  l = len(L) // 2
  LT1 = tri_fusion(L[:l])
  LT2 = tri_fusion(L[l:])
  return fusion(LT1, LT2)
```

Tout repose donc sur l'opération de fusion. Intuitivement, on va répéter:

- comparer les deux premiers éléments de LT1 et LT2
- prendre le plus petit, le sortir de sa liste, et l'ajouter à la fin de la liste résultat

jusqu'à ce qu'une des deux listes LT1 ou LT2 soit vide.

cf TP !

Un tri rapide : quicksort

• Sans vouloir battre la méthode sort de Python,programmons un tri performant: le tri rapide (***quicksort***) ou ***tri par pivot***. Son principe illustre à nouveau la stratégie DIVISER POUR RÉGNER.

De même que pour le tri fusion, on a 3 étapes

1. On partage la liste L en deux sous-listes L1 et L2.
2. On trie séparément L1 et L2 pour obtenir LT1 et LT2.
3. On réunit les listes triées LT1 et LT2.

• **Etape 1** : stratégie du **pivot**. On choisit un élément p dans la liste (par exemple le premier si la liste est en vrac) et on pose L* = L-{p}.

$$L1 = \{x \in L^* : x < p\} \quad \text{et} \quad L2 = \{x \in L^* : x \geq p\}$$

• **Etape 2** : la récurrence !

• **Etape 3** : on recolle les morceaux. $LT = LT1 \sqcup \{p\} \sqcup LT2$

• Exemple : L = [5,3,8,1,7,2] et le **pivot** p = 5. Alors :

(p = 5 ; L* = 3,8,1,7,2)

1. **Séparer** L en deux **par comparaison avec le pivot** 5

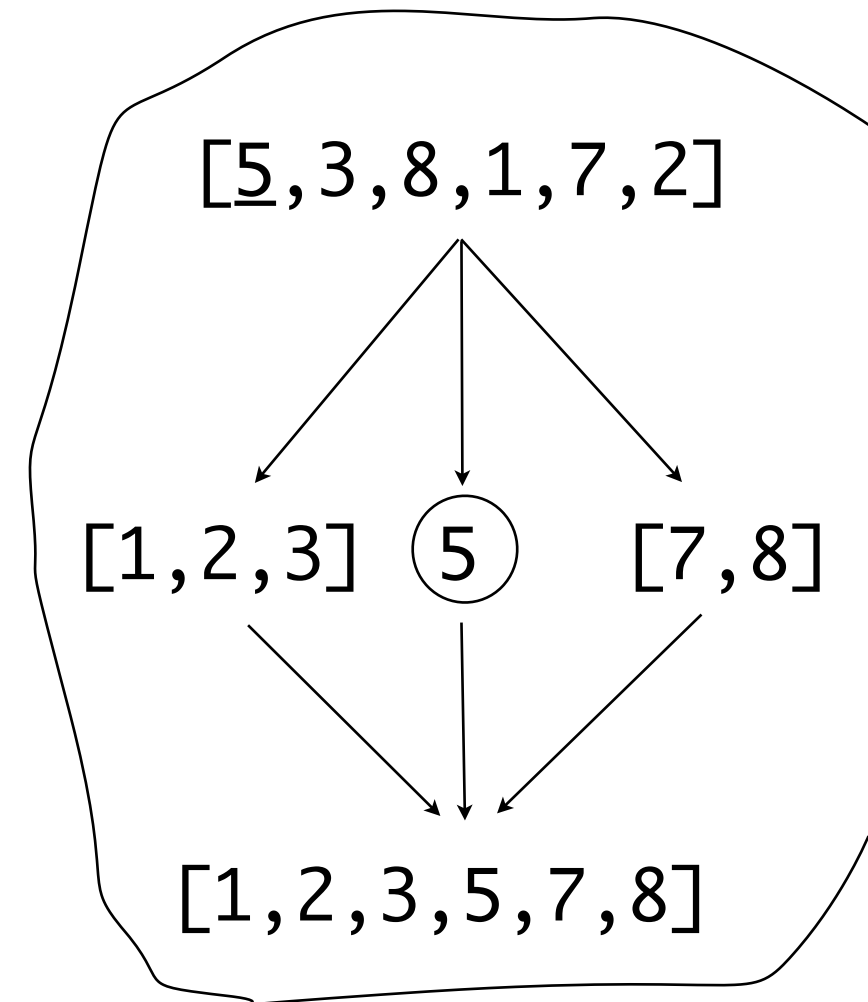
```
L1 = {x ∈ L* : x < 5} == [3,1,2]
L2 = {x ∈ L* : x ≥ 5} == [8,7]
```

2. **Trier par récurrence** L1 et L2

```
LT1 = quicksort(L1) == [1,2,3]
LT2 = quicksort(L2) == [7,8]
```

3. **Rassembler** les résultats obtenus

```
LT = [1,2,3] + [5] + [7,8]
LT == [1,2,3,5,7,8]
```



• Une **compréhension de listes** facilite l'écriture de la scission !

```
def quicksort(L) :
    '''Quicksort par récurrence'''
    if L == [] :
        return L
    p = L[0]      # le pivot
    L1 = [x for x in L[1:] if x < p]
    L2 = [x for x in L[1:] if x >= p]
    LT1 = quicksort(L1)
    LT2 = quicksort(L2)
    return LT1 + [p] + LT2
```

Trier

n	temps
10000	0.05
20000	0.12

1. Séparer

2. Trier par récurrence

3. Rassembler

• Le choix du premier élément L[0] comme **pivot** est excellent lorsque la liste est dans le désordre, mais devient pénalisant si elle est presque triée... Dans ce cas, on choisit le pivot au hasard et on le supprime aussitôt :

```
pivot = L.pop(randint(0,len(L)-1))
```

Comment comparer des algorithmes?

Empiriquement: on peut tester les algorithmes sur de grands jeux de données et regarder lequel est le plus rapide le plus souvent

C'est une approche tout à fait valable, mais le jeu de données retenu implique un biais

Théoriquement: on peut calculer la complexité dans le pire cas de l'algorithme.

La complexité dans le pire cas du tri rapide est $O(n^2)$, celle du tri fusion est $O(n\log(n))$.

En pratique, si l'on travaille sur les versions « en place » de ces algorithmes, le tri rapide tend à battre le tri fusion.

Comment évaluer la complexité d'un algorithme?

On va compter le nombre cn d'opérations élémentaires (lecture, écriture) pour une liste de n éléments. Pour le tri fusion, on a

c_n = *<coût de scission>* + *<coût des AR>* + *<coût de fusion>*

- Vous produirez en TD un algorithme de cout linéaire - en O(n) - pour la fonction `fusion`. Comment calculer la complexité de `tri-fusion` ?

$c_n = O(n) + 2\, c_{n/2} + O(n)$

$c_n = 2\, c_{n/2} + O(n)$

$c_n = 2\, c_{n/2} + n$ *pour simplifier !*

N'hésitons pas à simplifier le raisonnement, nous faisons des mathématiques d'ingénieur :-)

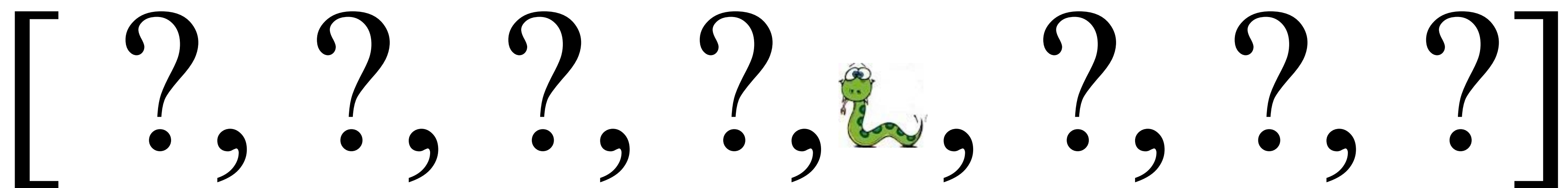
- Comment diable résoudre cette ***équation récursive*** $c_n = 2\, c_{n/2} + n$?

- Vous le verrez rigoureusement en math, nous nous contenterons de

$c_n = 2\, c_{n/2} + n = 2\,(2\, c_{n/4} + n/2) + n = 4\, c_{n/4} + n + n = \ldots = \underbrace{n + n + .. + n}_{\log(n) \text{ fois}}$

autrement dit $c_n = n\log(n)$

2.
Algorithmes de recherche



Recherche séquentielle dans une liste

• Problème fréquent en programmation : **rechercher la position** d'un élément dans une séquence (ici une liste).

• Soit L une liste de nombres, mais **sans ordre** : les nombres sont donnés *en vrac*. Il peut y avoir des **répétitions**. Par exemple :

```
L = [3, 2, 8, 6, 2, 4, 7, 6, 5]
```

• Je cherche l'élément `x`, par exemple x = 6. Il peut apparaître plusieurs fois. Je propose de chercher de la gauche vers la droite. Si $x \notin L$, je retourne -1, sinon je retourne le premier `i` tel que `L[i] == x`.

```
def chercher(x, L):
  for i in range(len(L)):
    if L[i] == x :
      return i
  return -1
```

```
>>> chercher(6,L)
3
>>> chercher(9,L)
-1
```

```
>>> L.index(6)
3
>>> L.index(9)
ValueError
```

Coût en $O(n)$, si $n = len(L)$

Recherche dichotomique dans une liste triée

• Lorsque la liste est dans le **désordre**, la recherche est **séquentielle** et coûte **O(n)** si n = len(L) : on *paye n* dans le pire des cas...

• Lorsque la liste est **triée**, donc dans l'ordre (plusieurs ordres possibles), on peut **accélérer la recherche**. La stratégie consiste à **abandonner la moitié de la liste chaque fois** !

1	2	5	7	8	10	12	13	15
0				*m*				

• En pseudo-langage, en supposant la liste croissante →

- *soit m l'indice du milieu de la liste*
- *si x == L[m], alors :*
  - *le résultat est m*

  sinon :
  - *si x < L[m], alors :*
    - *continuer dans la moitié gauche*
  - *sinon :*
    - *continuer dans la moitié droite*

N.B. Si l'élément x est répété, je ne garantis plus de trouver la première apparition !

• **Complexité dans le pire des cas** : combien de fois puis-je diviser n = len(L) par 2 ? Exemple avec n = 1000 :

$$\underbrace{n=1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0}_{k}$$

• En remontant de la droite vers la gauche, cela revient à effectuer k multiplications par 2, donc $n \simeq 2^k$, d'où $k \simeq \log_2(n)$.

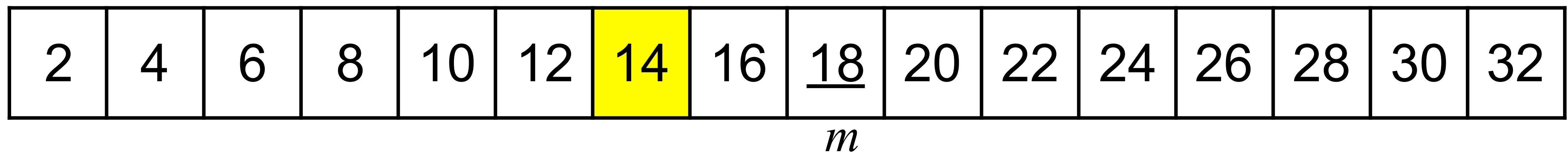
```
>>> log(1000) / log(2)
9.965784284662087
```

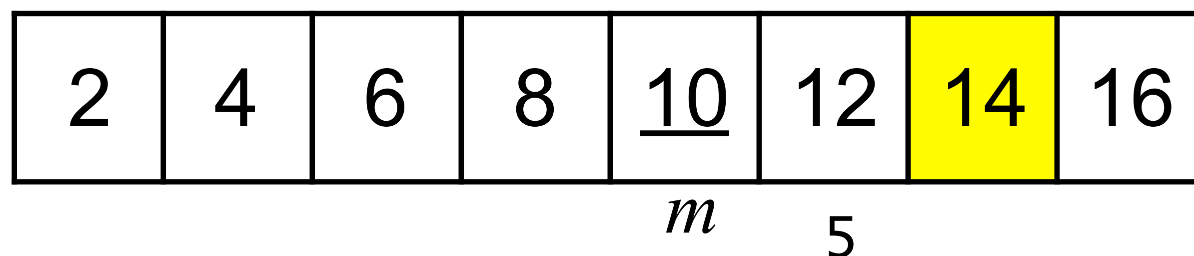
• Donc **bien meilleur que la recherche séquentielle** en O(n)...

```
def rech_dicho(x,L):      # retourne un i tel que L[i] == x
    if L == [] :
        raise ValueError # pas trouvé!
    m = len(L) // 2
    if x == L[m] :
        return m
    elif x < L[m] :
        return rech_dicho(x,L[:m])
    else :
        return m + 1 + rech_dicho(x,L[m+1:])
```

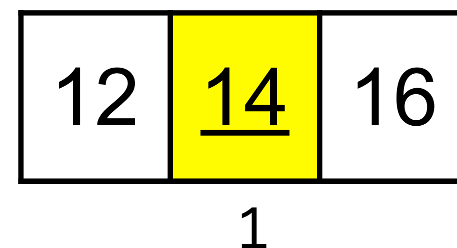
```
>>> L = [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34]
>>> rech_dicho(14,L)
6
>>> rech_dicho(7,L)
ValueError
```

- Etapes de la recherche de x = 14 :





Trouvé en position 5 + 1 = 6



N.B. Inconvénient du programme : beaucoup de calculs de tranches !

Exercice à faire à la maison: éliminez toutes ces tranches!

Indice: généralisez votre fonction en rech_dicho(x,L,deb,fin), et renvoyez un indice entre deb et fin

Attention : un coût sera-t-il amorti ?

- Donc deux types de recherches dans une liste :

Liste triée	Liste non triée
Recherche dichotomique	Recherche séquentielle
Coût en O(log n)	Coût linéaire en O(n)
Rapide !	Lent.

- Si une liste n'est pas triée, j'ai donc deux possibilités :

  - faire une recherche séquentielle ⇒ coût = O(n)
  - la trier d'abord puis faire une recherche dichotomique. ⇒ coût = O(n log(n) + log(n))
    = O (n log(n))

- Alors, quelle est la meilleure méthode ?

- Si je n'effectue la recherche qu'**une seule fois**, le tri n'est pas intéressant car $n \log(n)$ est plus cher que n.

- Ceci nous conduit à l'idée d'**amortissement** qui prend aussi en compte le nombre de fois que je vais utiliser un programme !

    - si je procède k fois à une recherche séquentielle (sans tri), j'aurais un coût de **kn**.

    - si je procède k fois à une recherche dichotomique (avec un seul tri), j'aurais un coût total de **n log(n) + k log(n)**.

- Mais si n est fixe et k assez grand (combien ?), il vaut mieux payer **n log(n) + k log(n)** que **kn**...

MORALE : lorsqu'on s'intéresse à la complexité, il faut à la fois tenir compte de l'algorithme et du nombre de fois qu'on l'utilisera.
Un algorithme peut être un ***one shot*** !

Recherche de motif dans un texte

Problème fréquent en informatique: **rechercher la position** du début d'une sous-séquence dans une séquence
(ici une chaîne de caractères) .

```
>>> motif = 'bab'
>>> texte = 'baobab'
>>> texte.index(motif)
3
```

b	a	o	b	a	b
			b	a	b

Nombreuses applications : traitement de texte, bio informatique, détection de plagiat, travail collaboratif, etc

On ne va pas chercher à faire mieux que la méthode string.index, on va simplement réfléchir à des algorithmes qui résolvent ce problème

Recherche naïve pas à pas

On répète les étapes suivantes :

1. on fait l'hypothèse que l'index est i
2. on vérifie que
   - 2.1 texte[i] == motif[0], et
   - 2.2 texte[i+1] == motif[1], et [...]
   - 2.n texte[i+n-1] == motif[n-1]

si l'un des test échoue, on repart à l'étape 1 avec i + 1

si tous les tests passent, on renvoie i

Complexité dans le pire cas :

O(len(texte) * len(motif))

```
>>> index('baobab', 'bab')
[b]aobab
[b]ab

b[a]obab
b[a]b

ba[o]bab
ba[b]

b[a]obab
 [b]ab

ba[o]bab
  [b]ab

bao[b]ab
   [b]ab

baob[a]b
   b[a]b

baoba[b]
   ba[b]

3
>>>
```

Recherche naïve pas à pas

```
def index(texte, motif):
    for i in range(len(texte)-len(motif)+1):
        for j in range(len(motif)) :
            affiche(texte, motif, i, j)
            if texte[i+j] != motif[j] :
                break
            elif j == len(motif) - 1 :
                return i
    raise ValueError
```

```
def marque(s, i) :
    return '{}[{}]{}'.format(s[:i],s[i],s[i+1:])

def affiche(texte, motif, i, j) :
    print('{}\n{}{}\n'.format(marque(texte,i+j),\
                              ' ' * i,\
                              marque(motif,j)))
```

```
>>> index('baobab', 'bab')
[b]aobab
[b]ab

b[a]obab
b[a]b

ba[o]bab
ba[b]

b[a]obab
 [b]ab

ba[o]bab
  [b]ab

bao[b]ab
   [b]ab

baob[a]b
   b[a]b

baoba[b]
   ba[b]

3
>>>
```

Peut-on faire mieux?

De nombreux algorithmes exploitent astucieusement le résultat des comparaisons précédentes pour écarter rapidement des positions à considérer. On arrive parfois à des algorithmes sous-linéaires (certaines lettres ne sont jamais lues).

Allez voir Knuth-Morris-Pratt et Boyer-Moore sur Wikipedia!

Relativement facile à comprendre, plus difficile à programmer…

```
>>> index_KMP('barbarbala', 'barbala')
[b]arbarbala
[b]arbala

b[a]rbarbala
b[a]rbala

ba[r]barbala
ba[r]bala

bar[b]arbala
bar[b]ala

barb[a]rbala
barb[a]la

barba[r]bala
barba[l]a

barbar[b]ala
   bar[b]ala

barbarb[a]la
   barb[a]la

barbarba[l]a
   barba[l]a

barbarbal[a]
   barbal[a]
```

Je n'essaie même pas d'aligner le premier a du mot avec le premier b du motif, je sais déjà que cela échouera

L'algorithme de Rabin-Karp

C'est un algorithme qui fait appel à une **fonction de hachage glissante** (rolling hash). Un exemple en est la fonction hash ci-dessous.

```
def str2int(s) : # str -> int
    return sum(ord(s[-i]) * 256 ** (i-1) for i in range(1,len(s)+1))

def hash(s) : # str -> {0,1,…,100}
    return str2int(s) % 101
```

La fonction str2int(s) renvoie l'entier dont l'écriture en hexadécimal est la même que la suite d'octets s, où s est une chaîne de caractères ASCII.

```
>>> n = str2int('abc')
>>> hex(n)
'0x616263'
>>> 'abc'.encode().hex()
'616263'
```

La fonction hash renvoie cet entier modulo 101. Le nombre 101 n'a aucune importance: simplement, pour éviter les collisions, il ne faut pas qu'il soit trop petit, et pour ne pas perdre de temps à hacher, il ne faut pas qu'il soit trop grand.

L'algorithme de Rabin-Karp

On va utiliser la fonction de hachage pour **hacher toutes les sous-chaînes** du texte de même longueur que le motif.

Si l'on trouve une sous-chaîne qui a le même digest que le motif, on la compare caractères par caractères au motif, sinon on passe à la position suivante.

On économise ainsi de nombreuses comparaisons, au prix du calcul du digest. Cela devient intéressant lorsque la fonction hash permet un **calcul amorti du digest**.

```
>>> index_RK('baobab', 'bab')
hash('bab') = 22

[bao]bab?
hash('bao') = 35

b[aob]ab?
hash('aob') = 84

ba[oba]b?
hash('oba') = 7

bao[bab]?
hash('bab') = 22

bao[b]ab
   [b]ab

baob[a]b
   b[a]b

baoba[b]
   ba[b]

3
>>>
```

Calcul amorti du digest

```
def update(h, n, s_i, s_j):
    # calcule hash(s[i+1:j+1] à partir de
    # h = hash(s[i:j]), s_i = s[i], et s_j = s[j] (et n = j-i)
    return ((h - ord(s_i) * 256 ** (n-1)) * 256 + ord(s_j)) % 101
```

Je retire l'octet de poids fort

Je décale les octets

J'ajoute l'octet de poids faible

```
>>> h = hash('abc')
>>> update(h, 3, 'a', 'd')
31
>>> hash('bcd')
31
```

IMPORTANT

le nombre d'opérations est fixe, il ne dépend pas de n. On calcule donc le digest suivant **en temps constant O(1)**. Si on avait voulu calculer le digest suivant en appelant la fonction hash, on aurait dû effectuer une somme de n termes, soit un temps O(n)

L'algorithme de Rabin-Karp

```
def index_RK(texte, motif) :
    hm = hash(motif)
    m = len(motif)
    for i in range(len(texte) - m + 1) :
        # je calcule le digest h = hash(texte[i:i+m])
        if i == 0 :
            h = hash(texte[:m])
        else :
            h = update(h, m, texte[i-1], texte[i+m-1])
        # je le compare à hash(motif)
        if h == hm :
            # s'ils sont égaux, je compare texte[i:i+m] et motif
            # caractère par caractère
            for j in range(m) :
                if texte[i+j] != motif[j] :
                    break
                if j == m - 1 :
                    return i
    raise ValueError
```

Conclusion

Les algorithmes sont souvent un mélange d'astuces élégantes et de grands principes universels (diviser pour régner, programmation dynamique, algorithmes gloutons, etc). Leur étude est passionnante!

Même des algorithmes relativement simples (ex: quicksort) font l'objet de recherches mathématiques poussées encore aujourd'hui, en particulier en combinatoire (on y a recours aux fonctions holomorphes). C'est un des nombreux exemples où l'informatique théorique et les mathématiques se rejoignent.

En TD vous verrez d'autres algorithmes proches de ceux vu dans le cours d'aujourd'hui, mais pour vraiment approfondir le domaine, je compte sur vous pour suivre les cours d'algorithmique de L2 et L3!