NSI 1ère - Algorithmique - Introduction

QK

Algorithmique

« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des téléscopes »

Michael R. Fellows et Ian Parberry

« Que nul n'entre ici s'il n'est géomètre »

Platon (427 av. J.-C., 348 av. J.-C.)

« Quand c'est qu'on joue à Fortnite ? »

Jean-Killian, 2019

l'Algorithmique, une science très ancienne.

- **Antiquité.** *Euclide* (calcul du pgcd), *Archimède* (approximation de π)
- le mot **Algorithme** vient du mathématicien arabe du 9ème siècle *Al Khou Warismi*
- L'algorithmique est, avec l'electronique, **la base scientifique de l'informique.** Elle intervient dans
  - le *logiciel* (software)
  - le *matériel* (hardware). Un processeur est un câblage d'algorithmes simples (addition, multiplication, portes logiques etc.)

Algorithme : une définition.

- Un algorithme prend en entrée des données et fournit, en un nombre fini d'étapes, la réponse à un problème.
- Un algorithme est une série d'opérations à effectuer :
  - Opérations en séquence (algorithme séquentiel)
  - Opérations en parallèle (algorithme parallèle)
  - Opérations exécutées sur un réseau de processeur (algorithme réparti / distribué)

- Mise en oeuvre de l'algorithme

  1. implémentation (plus général que le codage)
  2. écriture de ces opérations dans un langage de programmation (= codage)

  On obtient un programme

Algorithme vs Programme

- Un programme implémente un algorithme
- (*Turing-Church*) : les problèmes ayant une solution algorithmique sont ceux résolvables par une **machine de Turing** (théorie de la calculabilité)
- On ne peut pas résoudre tous les problèmes avec des algorithmes (indécidabilité algorithmique)
  - problème de l'arrêt (*Turing* 1936) - cet algorithme va-t-il s'arrêter ?
  - cet algorithme est-il juste ? (*Rice* 1951)

Complexité

Qualité d'un algorithme

- Deux algorithmes résolvants le même problème ne sont pas équivalents. 2 critères :
  - Temps de calcul : lents vs rapides
  - Mémoire utilisée : peu vs beaucoup
- On parle de compléxité en temps (vitesse) ou en espace (mémoire)

Pourquoi faire des algorithmes rapides ?

- Pourquoi faire des algos efficaces ? Les ordinateurs vont très vite !
- C'est faux, on n'a toujours pas assez de puissance de calcul (météo, mécanique des fluides, IA, trafic routier, séquençage du génôme etc.)

Quelques précisions

Seconde Loi de Moore ***(Gordon Moore - 1975)*** :

- le nombre de transistors des microprocesseurs sur une puce double tous les deux ans.

Elle est restée vraie de ~1960 à ~2000.

Depuis la dissipation thérmique limite la taille des puces, on a plutôt tendance à multiplier le nombre de coeurs de processeurs.

La **puissance de calcul** est la capacité à effectuer un certain nombre de calcul en un temps donné.

Nous discuterons plus en détail des conséquences économiques et environnementales.

Exemple

- Hypothèse optimiste (Loi de Moore) : la puissance de calcul double tous les deux ans.
- Mon programme en n^2 se termine en un temps satisfaisant avec $n = 10.000$

  Quand pourrais-je le faire avec $n = 1.000.000$?

Exemple - quelques précisions

- *Mon programme en* n^2 :

  cela signifie qu'il réalise n^2 opérations pour une donnée de taille n.
- *se termine en un temps satisfaisant avec* $n = 10.000$:

  il a donc réalisé $n^2 = (10.000)^2 = 10^8 = 100.000.000$ opérations en un temps satisfaisant.

Exemple - correction

- Hypothèse optimiste (Loi de Moore) : la puissance de calcul double tous les deux ans.
- *Mon programme en* n^2 *se termine en un temps satisfaisant avec* $n = 10.000$

  Quand pourrais-je le faire avec $n = 100.000$ *?*

  - $p = 100.000 \quad q = 10.000$; $p = 10 \times q$ donc $p^2 = 100 \times q^2$.

    On a besoin de 100 fois plus de puissance.
  - $2^6 = 64 \quad 2^7 = 128$. Elle sera obtenue dans $7 \times 2 = 14$ ans !!!

Exemple - correction - suite

Mon autre algorithme est en $n \log n$

- $\log n$ (logarithme de n) est une fonction croissante vers l'∞, comme n mais "très lente"

Entre $q = 10.000$ et $p = 100.000$

On passe de 100.000 opérations à 1.000.000 d'opérations.

Il ne faudra que 4 ans ($2^3 = 8 \quad 2^4 = 16$).

Complexité des algorithmes

- **But** :
  - **Avoir une idée de la difficulté des problèmes**
  - **Donner une idée du temps de calcul ou de l'espace nécessaire pour résoudre un problème**
- Cela va permettre de comparer des algorithmes.
- La complexité est exprimée en fonction du nombre de données et de leur taille.
- C'est très difficile…
  - On considère que toutes les opérations sont équivalentes
  - Seul l'ordre de grandeur nous intéresse.
  - On considère uniquement le *pire* des cas (souvent ~ cas *moyen*)

Pourquoi faire des algos rapides ?

- Dans la vie réelle, ça n'augmente pas toujours !
- OUI et NON :
  - Certains problèmes sont résolus.
  - Pour d'autres, on simplifie moins et donc la taille des données à traiter augmente !

    Réalité virtuelle : de mieux en mieux définie.
- Dès que l'on résout un problème on le complexifie !

Algorithme : vitesse

Rapide un algorithme qui met un temps *polynômial* en n (nombre de données) pour être exécuté : n^2, n^8

Lent un algorithme qui met un temps *exponentiel* : 2^n

Pour certains problèmes (voyageur de commerce, remplissage de sac à dos de façon optimale) on **ne sait même pas s'il existe** un algorithme rapide.

On connait des algorithme *exponentiels* en temps 2^n.

- $100^2 = 10.000$ $100^3 = 1.000.000$
- $2^{100} = 1267650600228229401496703205376$

Algorithme : vitesse

1 million de $ si vous résolvez la question !

4eme problème du millénaire : $P = NP$?

Exemples en Python :

Tri à bulle vs tri implémenté dans Python3

Tri à bulle

On veut trier (= ranger par ordre croissant) `[3, 2, 1]`

```
  On prend 3. On parcourt à partir de 3.
  À chaque étape, on échange si 3 est plus grand que l'élément :
    1.  3 > 2 on échange. [2, 3, 1]
    2.  3 > 1 on échange. [2, 1, 3]
  On recommence avec le 2ème élément :
    1. 1 < 3 on ne fait rien.
On recommence depuis le départ
jusqu'à ce qu'on n'ait plus aucun échange.
```

Tri à bulle (suite)

```
[2, 1, 3]
    1. 2 > 1 on échange. [1, 2, 3]
    2. 2 < 3 on ne fait rien.
    On recommence avec le 2ème élément :
    1. 2 < 3 on ne fait rien.
On recommence depuis le départ
  1. 1 < 2 on ne fait rien.
  2. 2 < 3 on ne fait rien.
C'est terminé.
```

4

4 comparaisons effectuées après avoir terminé le tri.

Pour une liste de 3 éléments.

Dans Python

```python
def bubble_sort(l):
    while True:
        echange = False
        for i in range(len(l) - 1):
            if l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
                echange = True
        if not echange:
            return l
```

Python Tutor

Comparaison de vitesse

```
from time import time
from random import sample

n = 1000  # on mélange 2 listes de 1 à 1000
l1, l2 = sample(range(n),n), sample(range(n),n)

start = time() # on note l'heure de départ
bubble_sort(l1)
print(time() - start) # 0.3104 secondes

start = time()
sorted(l2) # tri interne
print(time() - start)
# 0.0003 sec : 1000 fois plus rapide.
```

Conclusion

- Le tri à bulle est bien gentil. On en verra de meilleurs.
- Il est important d'utiliser de bons algorithmes pour réaliser des tâches coûteuses.

Preuve

Algorithme : preuve

- On peut *prouver* les algorithmes !
- Un algorithme est dit **totalement correct** si, pour tout jeu de données, il termine et rend le résultat attendu.
- C'est difficile (très) mais c'est important.
  - Encodage, décodage des données (compression, cryptographie etc.)
  - Centrale nucléaire
  - Airbus
- Attention : un algorithme juste peut être mal implémenté.

Algorithme : résumé

- C'est ancien
- C'est fondamental en informatique
- Ça se prouve
- On estime le temps de réponse et la mémoire occupée
- Algorithme $\neq$ Programme

Structures de données

2 types de personnes

- En cuisine il y a deux types de personnes :
  - les chefs écrivent les recettes
  - et les cuisiniers qui préparent les plats.

  Les chefs ne font pas la cuisine.
- En informatique c'est pareil, il y a :
  - ceux qui écrivent les algorithmes
  - et ceux qui les implémentent
- Problème : il faut se comprendre.

Se comprendre...

- Pour se comprendre on va **améliorer le langage.**
  - On définit des choses bien précises (comme en cuisine)
  - On essaie de regrouper certaines méthodes ou techniques
- En informatique :
  - Langage : variable, boucle, incrémentation...
  - Regroupement : structures de données, types abstraits...

Variable

- Une variable sert à mémoriser de l'information
- Ce qui est dans une **variable** est mis dans **une partie de la mémoire**

Type de données

Type de données : le genre de contenu d'une donnée et les opérations pouvant être effectuées sur la variable correspondante

- Par exemple :
  - **int** représente un entier. On peut faire des additions, multiplications...
  - **Date** représente une date. On peut aussi faire des additions, on peut l'écrire d'une certaine manière (jj/mm/aaaa) ou "10 septembre 2049" (ma date de naissance).

- Les types que nous rencontrerons souvent sont : int, float (~ nombre réel), booléen (V / F), chaîne etc.

Exemples en Python

Accéder au type d'un objet avec `type( )`

```
>>> n = 5; f = 3.14; b = True; nom = "Robert"
>>> type(n); type(f); type(b); type(nom)
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

Python a un typage dynamique

Il n'est pas nécessaire de préciser le type de données.

En C ou en java, on écrirait :

```
int n = 5;
```

Dans Python il suffit d'écrire `n = 5`

- Avantage : simplicité
- Inconvénients : nombreux

Un danger du typage dynamique en Python

```
b = 5
if b == 1:
  a = 1 - "marcel" # jamais exécutée
else:
  a = "bonjour " + "marcel"
```

L'opération `1 - "marcel"` est impossible.

Pas d'erreur tant que cette instruction n'est pas **exécutée**

Structure de données

Structure de données : une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir être utilisées efficacement

Différentes structures de données existent pour résoudre des problèmes précis :

- tableaux (listes en Python)
- B-arbres dans les bases de données (BTrees en Python)
- tables de hachage dans les compilateurs (~ dictionnaires en Python)

Structures de données

- Ingrédient essentiel pour l'efficacité des algorithmes
- Permettent d'organiser la gestion des données
- Une structure de données ne regroupe pas nécessairement des objets du même type

Type abstrait de donnée

- Un **type abstrait** de données ou une **structure de données abstraite** est une spécification mathématique d'un ensemble de données et des opérations qu'elles peuvent effectuer. C'est un cahier des charges qu'une structure de données doit ensuite implémenter.

Exemple de type abstrait : la pile

Défini par sa structure et 2 opérations :

- Une pile contient des éléments.
- **Push** : insère un élément à la fin
- **Pop** : extrait le dernier élément inséré de la structure

La pile en Python

Les **listes** permettent d'effectuer les opérations des piles :

Contenir, Push et Pop :

```
>>> pil = [1, 2, 3]
>>> pil.append(4) # push en Python
>>> pil
[1, 2, 3, 4]
>>> pil.pop()
4
>>> pil
[1, 2, 3]
```

Structures de données : résumé

- Permettent de gérer et d'organiser des données
- Sont définies à partir d'un ensemble d'opérations qu'elles peuvent effectuer sur les données

Structures de données et langage à objets

- Les structures de données permettent d'organiser le code.
- une SDD correspond à une **classe** contenant 2 parties
  - Une **visible** : les opérations que la structure peut effectuer. En langage objets on parle de **partie publique**

  - Une **cachée** : les méthodes internes permettant de réaliser les opérations de la SDD. En langage objets on parle de **méthode privée**

- La partie visible de la SDD est parfois appelée **API** de la SDD :

  Application Programming Interface, autrement dit l'interface de programmation de la SDD.

Exemple en Python : la classe `str`

Extrait de la doc : *Les données textuelles en Python sont manipulées avec des objets str ou strings. Les chaînes sont des listes immuables* (on ne peut les modifier) *de caractères Unicode.* (lettres, nombres, symboles etc.)

Quelques méthodes

`find` : renvoie la première position de la sous chaîne, `-1` sinon

```
>>> 'Python'.find('yth')
1
>>> 'Py' in 'Python' # test d'appartenance
True
```

`format` : permet de formater une chaîne :

```
>>> name = "Robert"
>>> 'bonjour {}'.format(name)
'bonjour Robert'
```

Méthodes publiques, privées

Les méthodes présentées ci-dessus sont toutes **publiques**

Pour connaître les méthodes privées il faut consulter le code source de Python

La deuxième phrase de la doc nous dit : **ce sont des listes !**

Voilà pourquoi on peut tester `'Py' in 'Python'` (on en reparlera plus tard)

Pseudo langage

Notions de pseudo langage

- langage formel minimal pour décrire un algorithme
- un langage de programmation (Python, java, C) est trop contraignant
- dans les livres les algos. sont écrits en pseudo langage.

Eléments communs

Tous recouvrent les mêmes concepts :

- variables, affectations
- structure de contrôle : séquence, condition, itération
- découpage de l'algo. en sous-programmes (fonctions)
- structures de données (tableaux, dictionnaires etc.)

Variables, affectations

- Les variables sont indiquées avec leur type : booléen b, entier n etc.
- on affecte avec = ou ← ← illustre bien l'affectation : "mettre dedans."

Structures de données

- Les tableaux sont utilisés. Si A est un tableau, $A[i]$ est le i^{eme} élément de ce tableau.
- Les structures sont utilisées. Si P est une structure modélisant un point et x un champ modélisant l'abscisse alors $P.x$ est l'abscisse de ce point.

Séquencement des instructions

- si nécessaire, on termine une instruction avec ;
- Les blocs d'instructions sont entourés de
  - {...}
  - début ... fin
- En Python, ce n'est nécessaire que sur un même ligne.

```
a = 3; print(a)
```

Conditionnelle

La conditionnelle est donnée par :

```
si (condition){
  instruction1;
}sinon{
  instruction2;
}
```

Python et l’indentation

Indenter :

Mettre des espaces en début de ligne

- Pseudo code, langage avec syntaxe inspirée du C :

  La structure est indiquée par les `{ }`

  indentation optionnelle, pour éclairer le lecteur

- Python :

  Dans Python la **structure est donnée par l’indentation**

  Elle est **OBLIGATOIRE**

Conditionnelle en Python

```python
a = 5
if a > 4: # apres : on indente
  print("plus grand que 4") # dans le bloc if
else:
  print("inférieur ou égal à 4") # dans le bloc else
print("le if est terminé") # sera executé
```

qui affiche :

```
plus grand que 4
le if est terminé
```

Python : if, elif, else

`a % 3` le reste dans la division par 3 de `a`

`%` se lit “modulo”

```python
a = 16
if a % 3 == 0: # si a est divisible par 3
  print("divisible par 3, le reste vaut 0")
elif a % 3 == 1:
  print("le reste vaut 1")
else:
  print("le reste vaut 2")
```

Itérations

Nous utiliserons plusieurs types de boucles :

Les boucles `while`

```
tant que (condition){ ... }
```

En Python :

```
n = 0
while n < 5:
  print(2*n + 1)
  n += 1
```

1, 3, 5, 7

Contexte d'utilisation : `while`

On emploie principalement les boucles `while` quand **on ne sait pas combien d'étapes seront nécessaires**.

Itérations

Les boucles `for`

```
pour i allant de min à max { ... }
```

```
for i in range(5):
  print( 2 * i + 1)
```

Itérer dans une liste en Python.

```
for mot in ["une", "liste"]:
  print(mot)
```

```
une
liste
```

Rq : Il est possible d'itérer sur de nombreux objets dans Python :

liste, tuples, dictionnaires, set, fichiers, générateurs etc.

Ici on **itère** sur la liste,

i référence succéssivement ses éléments.

Contexte d'utilisation : `for`

- Quand on sait combien d'étapes seront nécessaires
- Quand on veut parcourir tous les éléments d'un "paquet" (liste, tableau, dictionnaire etc.).

Fonctions

- Une fonction est un “petit” programme qui renvoie une valeur.
- Elles permettent
  - un découpage qui **facilite la compréhension**.
  - de **factoriser** : on évite ainsi d’écrire plusieurs fois la même série d’instruction.

Fonctions en Python

Elles sont définies avec **`def`** En sortie, elles revoient des variables avec **`return`**

```
def carre(x):
  return x ** 2

a = carre(9) # 81
```

Fonction sans sortie

```
def direBonjour(texte):
  print("Bonjour je m'appelle {}".format(texte) )

direBonjour("Henri")
```

Rq : `.format(...)` est une **méthode** de la classe **`str`**

Intérêt des fonctions :

Les fonctions facilitent le développement :

Il est plus facile de trouver une erreur parmi 10 fonctions de 3 lignes que dans un bloc de 30 lignes.

Portée des variables

On distingue les variables **locales** et **globales**

```
a = 5 # variable globale
def maFonction():
  a = 3 # variable locale à la fontion
  print(a)
maFonction()
print(a) # a vaut 5 !!!
```

```
3
5
```

En dehors des fonctions : variables **globales**

Dans les fonctions : variables **locales**

Attention aux arnaques : Python Tutor - portée des variables

Python : utiliser une variable globale

```
a = 5 # variable globale
def maFonction():
  global a # on rend la variable globale
  a = 3
maFonction()
print(a)
```

```
3
```

Cette fois, on a spécifié `global a` !!

En pseudo code

On évite cette difficulté en précisant les paramètres (entrées et sorties) de la fonction.

- maFonction(**e** int **i**, **s** int **j**, **es** int **k**)
- en entrée : avec **e** on passe à la fonction la valeur i mais elle ne la changera pas globalement
- en sortie : la fonction écrit dans **j** mais elle n'en connait pas la valeur extérieure
- en entrée/sortie : on lui passe la valeur de **k** et elle écrit dedans.
- Sans précision, on suppose que c'est en entrée.
- passage par entrée/sortie = **passage par référence** ou **passage par variable**

Exemple

- code appelant :

  i ← 3

  j ← 5

  k ← 8

  maFonction(e int i, s int j, es int k);

  Afficher(i, j, k);

- maFonction(e int i, s int j, es int k){

  i ← i +1

  j ← 6

  k ← k + 2 }
- résultat de Afficher(i, j, k) : ? ? ?

Exemple - solution

- code appelant :

  i ← 3

  j ← 5

  k ← 8

  maFonction(e int i, s int j, es int k);

  Afficher(i, j, k);
- maFonction(e int i, s int j, es int k){

  i ← i +1

  j ← 6

  k ← k + 2 }
- résultat de Afficher(i, j, k) : 3 6 10

fonctions : différentes implémentations

- En Java certains objets n'ont de paramètres qu'en entrée (int, float etc.)

Python : prudence !

- En Python il faut faire attention, à nouveau…

```
def f(x):
  x = 5
a = 0
print(a) # 0
f(a)
print(a) # 0
```

a été passé en entrée

Tandis que…

```python
def f(l):
  l[0] = 5
a = [0, 1, 2]
print(a[0]) # 0
f(a)
print(a[0]) # 5
```

Cette fois on modifie l'élément 0 de la liste a !

Tableaux

Tableaux

- Un tableau (*array* en anglais) est une SDD qui contient un ensemble d'éléments auquel on accède avec un numéro d'indice.
- Le temps d'accès à un élément par son indice est *constant*
- Les éléments sont contigus dans l'espace mémoire. Avec l'indice on sait à combien de cases mémoire se trouve l'élément en partant du début du tableau
- Souvent désignés par une majuscule : T est un tableau, $T[i]$ est son élément d'indice i

Avantages / Inconvénients

- **Avantages** : accès direct au ième élément
- **Inconvénients** : les opérations d'insertion et de suppression sont impossibles.
  - Il faut créer un nouveau tableau, de taille plus grande ou plus petite (selon l'opération). Il faut alors copier tous les éléments du tableau original dans le nouveau tableau. Cela fait beaucoup d'opérations.

Tableaux en Python : des listes !

Dans Python les tableaux sont des objets *listes*.

On peut les construire de plusieurs manières :

- `[]` est une liste vide
- `l = [""abc", "def", "ghi""]` et on accède avec `l[1]` `"def"`
- Par compréhension : `[2 * x + 1 for x in range(4)]` `[1, 3, 5, 7]`

Listes par compréhension, suite

On peut itérer dans une chaîne de caractères donc facilement la découper

```
>>> lettres = "abcdefg"
>>> [ i for i in lettres ]
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

- Il y a bien d'autres façons de faire.

La grande différence entre les tableaux et les listes ?

Les listes sont **mutables** :

```
>>> l = ['abc', 'def', 'ghi']
>>> l[1]
'def'
>>> l[1] = 'xyz' # modifier un élément
>>> l
['abc', 'xyz', 'ghi']
>>> l.insert(2, "mno") # insérer
>>> l
['abc', 'xyz', 'mno', 'ghi']
>>> l.remove('abc') # supprimer l'élément 'abc'
>>> l
['xyz', 'mno', 'ghi']
```

Affecter : le même objet !

Dans Python quand on affecte un objet à un autre, ils sont identiques !

```
>>> l = ['abc', 'def', 'ghi']
>>> m = l # m et l : les mêmes objets
>>> m[1] = "xyz" ; l[0] = "pqr" # l et m modifiés
>>> l, m
(['pqr', 'xyz', 'ghi'], ['pqr', 'xyz', 'ghi'])
```

Copier : une copie de l'objet

On contourne cette difficulté avec un "slice"

```
>>> l = ['abc', 'def', 'ghi']
>>> l[1:] # une copie de l à partir de l'élément 1
>>> n = l[:] # une COPIE complète de l
>>> l[0] = "pqr" # l est modifée, pas n
>>> l, n
(['pqr', 'def', 'ghi'], ['abc', 'xyz', 'ghi'])
```

Retour sur les tableaux

- On met ce qu'on veut dans un tableau

  $T = [1.44,\ 3.14,\ 2.72]$

- Pas forcement des objets de même nature

  $T = ["abc",\ 3.14,\ True]$

Élément $\neq$ indice

Ne pas confondre l'élément et l'indice

$T = [1.44,\ 3.14,\ 2.72]$

3.14 est l'élément, son indice est 1

$T[1]$ est 3.14

Tableaux multidimensionnels

Tableau bidimensionnel

Un **tableau bidimensionnel** ou **matrice** est un tableau qui contient des tableaux.

Tableaux de tableaux...

- Par exemple :

$$T \leftarrow [\ [a, b, c], [d, e, f], [m, n, o]\ ]$$

l/c	0	1	2
0	a	b	c
1	d	e	f
2	m	n	o

- On dit que T est une matrice à 3 lignes et 3 colonnes

Accéder à un élément

- Comment accéder à f ?

$$T \leftarrow [\ [a, b, c], [d, e, f], [m, n, o]\ ]$$

l/c	0	1	2
0	a	b	c
1	d	e	f
2	m	n	o

- f est l'élément $T[1][2]$

  ligne 1, colonne 2

Python : listes de listes

0	1
2	3

Pour présenter facilement on s'aligne au même niveau

```
T = [[0, 1],
     [2, 3]] # [[0, 1], [2, 3]]
T[1][0] # 2
```

Python : listes de listes par compréhension

Une méthode efficace :

```
>>> T = [[j for j in range(3*i, 3*i+3)]
... for i in range(3)]
```

Qu'obtient-on dans `T` ?

Python : listes de listes - solution

```
>>> T = [[j for j in range(3*i, 3*i+3)]
... for i in range(3)]
>>> T
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

l/c	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

Itérer dans une matrice.

pour itérer dans une matrice il faut 2 boucles *imbriquées*

```
Pour i allant de 1 à n {
  Pour j allant de 1 à n {
```

```
    faire... T[i][j] ...
  }
}
```

Calcul d'une moyenne des éléments d'un tableau.

Le tableau T comporte n lignes et p colonnes.

On calcule la moyenne habituelle donnée par la formule par

$$\frac{1}{n \times p} \sum_{i=0}^{n-1} \sum_{j=0}^{p-1} T[i][j]$$

- $\sum$ repésente la somme
- $\sum_{i=0}^{n-1}$: pour i allant de 0 à $n-1$ ajouter…
- Les deux $\sum$ sont l'une dans l'autre.
- On divise la somme des termes par $n \times p$

Exemple en Python.

Créer une fonction qui prenne une matrice en entrée et renvoie la moyenne de ses valeurs en sortie

Solution "naturelle"

```python
def moyenne(T):
  s = 0; n = len(T); p = len(T[0])
  for i in range(n):
    for j in range(p):
      s += T[i][j]
  return s / (n * p)
```

Solution avec des outils de python

```python
def moyenne(T):
  s = 0; k = len(T)*len(T[0])
  for ligne in T:
    for x in ligne:
      s += x
  return s / k
```

Solutions encore plus radicale…

```python
def moyenne(T):
    s = 0; k = len(T) * len(T[0])
    for ligne in T:
```

```
        s += sum(ligne)
    return s/k
```

Python Tutor Et la plus courte à laquelle j'ai pensé :

```
def moyenne(T):
    return sum([sum(row) for row in T])\
      / (len(T) * len(T[0]))
```

Il y a encore plus court…

Celles là ne sont pas de moi

```
def moyenne(T):
  return sum(map(sum, T)) / (len(T) * len(T[0]))
```

```
def moyenne(T):
  return sum(sum(T,[])) / (len(T) * len(T[0]))
```

Exercices

Une étape de 2048

On considère une liste de nombres (exemple : `[2, 0, 4, 8]`)

Programmer une fonction `zeroADroite(liste)` qui renvoie une liste de même taille mais avec tous les 0 qu'elle contenait déplacés à droite.

Exemples :

```
>>> zeroADroite([2, 0, 4, 8])
[2, 4, 8, 0]
>>> zeroADroite([0, 0, 4, 0])
[4, 0, 0, 0]
>>> zeroADroite([2, 0, 4, 8])
[2, 4, 8, 0]
>>> zeroADroite([4, 0, 0, 16])
[4, 16, 0, 0]
```

Tuples et dictionnaires

p-uplets

- En mathématiques on connait les "couples" de valeurs :

  Le point A d'abscisse 3, d'ordonnée 4 se note $A(3;4)$.

- Quand on dispose de p données on parle de p**-uplet**.

  En anglais ça donne un "tuple".

Les tuples en Python

- Dans Python les tuples sont des séries de données, séparées de virgules.

```
>>> un_tuple = (3, 4, 5)
>>> un_tuple[1] # comme pour une liste
4
>>> type(un_tuple)
<type 'tuple'>
>>> for x in un_tuple: # on peut itérer sur un tuple
      print(x**2)
9
16
25
>>> autre_tuple = ("pomme", 1, True)
```

Les tuples ne sont pas mutables !

```
>>> un_tuple[1] = 2 # on ne peut pas changer un élément
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Une fois un tuple défini il n'est plus possible de le transformer (**non mutable**).

- C'est comme une liste, mais plus rapide et pas modifiable.
- Pratique quand on sait qu'une liste de valeur ne changera plus.

Tuples et fonctions

- Une fonction peut renvoyer un tuple

```
>>> def f(x, y):
      return x**2, y**3
>>> f(3, 2)
(9, 8)
```

p-uplet nommés

- *Remarque importante* : ici le programme officiel de la spécialité NSI est imprécis :

  "En Python, les p-uplets nommés sont implémentés par des dictionnaires."

- C'est faux. Les dictionnaires en Python sont des objets particuliers et les "p-uplets nommés" sont implémentés par des *namedtuples* auxquels on accède via la librairie "collections"

- Je présenterai donc *les dictionnaires en Python* et pas les *namedtuples en Python*

  Nous parlerons toujours de dictionnaires et jamais plus de p-uplets nommés. Les idées dont nous avons besoin sont les mêmes.

et donc : les dictionnaires en Python

Les dictionnaires sont des "tableaux associatifs" indexés par des *clés*

```
>>> scores = {"Jean": 145, "Paul": 200, "Emy": 345 }
>>> scores["Emy"] = 364
>>> scores
{"Jean": 145, "Paul": 200, "Emy": 364 }
>>> del scores["Paul"]
>>> scores["Téo"] = 308
>>> scores
{"Jean": 145, "Emy": 364, "Téo": 308}
```

Les dictionnaires sont **mutables**. On accède à une **valeur** à l'aide de sa **clé**

Element d'un dictionnaire, conversions, tris

On peut tester l'appartenance, trier ou convertir un dictionnaire :

```
>>> scores = {"Jean": 145, "Emy": 364, "Téo": 308}
>>> "Jean" in scores
True
>>> list(scores)
["Jean", "Emy", "Téo"] # extraire la liste des clés
>>> sorted(scores)
["Emy", "Jean", "Téo"] # liste triée des clés
```

Dictionnaires par compréhension

On peut créer de plusieurs manières un dictionnaire par compréhension :

Par exemple, pour stocker les valeurs d'une fonction :

```
>>> {x: x**2 for x in  (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

On via une série d'associations :

```
>>> Quentin = dict(age=71, taille=180, poids=150)
>>> Quentin
{'age': 71, 'taille': 180, 'poids': 150}
```

Et avec `zip` :

```
>>> dict(zip(["x", "y", "z"], [18, 25, 32]))
{"x": 18, "y": 25, "z": 32}
```

Itérer dans un dictionnaire

On peut boucler dans un dictionnaire et récupérer les clés et les valeurs :

```
>>> persos = {'Robert': 'le musclé',
'Nadine': 'la rusée'}
>>> for key, value in persos.items():
        print(key, value)
Robert le musclé
Nadine la rusée
```

Clés et valeurs

On peut aussi itérer sur la liste des clés ou des valeurs :

```
>>> ingredients = {'poules': 2, 'coqs':4,
'cochons': 8}
>>> for x in ingredients.keys():
        print(x)
'poules'
'coqs'
'cochons'
>>> for y in ingredients.values():
        print(y)
2
4
8
```

Compléments algorithmique

Recherche dichotomique

x est-il un élément de ma *liste* ?

On considère une liste de nombres **rangés par ordre croissant**

Ex : $l = [0, 1, 5, 17]$.

On cherche à savoir si un nombre est dedans.

Solutions natives en Python

Python le fait nativement :

```
>>> l = [0, 1, 5, 17]
>>> 5 in l, 8 in l
(True, False)
>>> l.index(5), l.index(8)
(2, -1) # 5 est l'élément d'indice 2, 8 n'est pas dedans
```

Oui mais comment ?

solution naïve

On parcourt 1 par 1 les éléments :

```
Pour i allant de 0 à la longueur -1 {
  si l[i] vaut 5 {
    renvoyer Vrai
  }
}
Renvoyer Faux
```

En python :

```
def Recherche(l, x):
  for i in range(len(l)):
    if l[i] == x:
      return i
  return -1 # x n'est pas dedans
```

Dans le pire des cas, **on parcourt tout le tableau.**

Recherche dichotomique

Disons qu'on cherche 5.

- On regarde l'élément central m.
  - Si $m = 5$ c'est gagné. On renvoie sa position.
  - Sinon, si $m > 5$ alors 5 est **à sa gauche.**

    notre nouvelle liste est constituée des **éléments à gauche de** m
  - Sinon, c'est que 5 est à droite de m.

    notre nouvelle liste est constituée des **éléments à droite de** m
- On recommence depuis le départ avec la nouvelle liste.

Exercice

- Proposer un algorithme en pseudo code avec une boucle tant que.
- Compter les nombre maximal d'étapes pour une liste de taille 16, 32, 64 etc.

- Que remarque-t-on ?
- Coder cet algorithme en Python

Solution - pseudo code

```
rechercheDicho(liste, clé){
  bas = 0
  haut = longueur(liste) - 1
  Tant que (bas < haut){
    med = (bas + haut) // 2
    si clé == liste[med] {bas = med ; haut = med}
    Sinon{
      si clé > liste[med] {bas = med + 1}
      sinon {haut = med - 1}
    }
  }
  si cle == liste[bas]{renvoyer Vrai}
  sinon {renvoyer Faux}
}
```

Solution (suite)

- Le pire des cas est atteint quand la clé n'est pas dans la liste.
- Alors, on divise par 2 la taille de la liste jusqu'à ce que sa taille soit 1.
- S'il y a r division, alors $2^r < n < 2^{r+1}$
- On note $r \leq \log_2(n)$ - le logarithme évoqué plus tôt…
- La recherche dichotomique est un algorithme en $O(\log_2 n)$
- Pour une taille ~1000, la recherche séquentielle effectue au pire 1000 étapes,

  la recherche dichotomique au pire 10.

Solution en python 3 (P. Tutor)

```python
def rechercheDicho(liste, cle):
    bas = 0
    haut = len(liste) - 1
    while bas < haut:
        med = (bas + haut) // 2
        if cle == liste[med]:
            bas = med
            haut = med
        else:
```

```
            if cle > liste[med]:
                bas = med + 1
            else:
                haut = med - 1
    if cle == liste[bas]:
        return True
    else:
        return False
```

Algorithmes gloutons

Les algorithmes gloutons (*greedy*)

Classe d'algorithme qui font, étape par étape, un choix optimum local.

On cherche à optimiser une réponse selon un critère.

Le problème du sac à dos

Un voleur, parvenu dans le coffre doit choisir comment remplir son sac pour dérober un maximum.

Il peut porter 50 kg.

Il cherche à le remplir pour optimiser le montant dévalisé.

Oui mais comment choisir ?

Algorithme glouton

La solution simple consiste à remplir son sac en prenant toujours en premier l'objet de plus grande valeur qui entre dans le sac.

Sac à dos : glouton, exemple

Poids : 1, 2, 4. Capacité du sac : 5

Toutes les valeurs sont égales aux poids.

1. On commence par 4.

   $4 \leq 5$ donc on prend 4. Il reste 1

2. On continue avec 2.

   $2 > 1$ on ne prend pas 2.

3. On continue avec 1. $1 \leq 1$ donc on prend 1. Il reste 0. Terminé.

$5 = 1 \times 4 + 0 \times 2 + 1 \times 1$

Est-ce le meilleur algorithme ?

NON !

Par exemple :

3 objets dans le coffre, capacité du sac : 5

- Objet A : masse 4
- Objet B : masse 3
- Objet C : masse 2

L'algorithme glouton va choisir l'objet A (valeur la plus élevée) et s'arrêter.

La solution optimale est de prendre les objets B et C.

3 objets et ça échoue déjà !

Algorithme glouton : inutile ?

Pas du tout ! Il existe un cas simple dans lequel il donne toujours la meilleure réponse :

suite super croissante :

- Les poids sont *super croissants* si, une fois triés, la somme des k premiers est inférieure au poids $k+1$.
- $1,\ 3,\ 5,\ 10,\ 21$ est une suite de poids super croissante.

  $1 < 3$, $1+3 < 5$, $1+3+5 < 10$ etc.
- $2,\ 3,\ 4$ n'en est pas une : $2+3 > 4$
- Dans le cas "super croissant", l'algorithme glouton rend toujours la combinaison optimale.

Sac à dos : un problème fréquent

On retrouve ce problème un peu partout :

- Gestion de portefeuille en finance : quelles actions choisir ?
- Chargement d'avions : tous les bagages doivent être amenés, sans surcharge
- Découpe des matériaux : réaliser un maximum de pièces en limitant les chutes

Sac à dos : un problème NP-complet

- On dit d'un problème informatique qu'il est de classe P s'il existe un algorithme "polynomial" (en n^2, n^8 etc) qui renvoie une solution.

- Il est NP si on peut **vérifier** les solutions de manière polynomiale mais qu'il **n'existe pas** de moyen de les obtenir qui soit polynomiale.

Autrement dit : *NP-complet* = facile à vérifier, quasi impossible à résoudre.

L'algorithme du sac à dos est *NP-complet.* Il n'existe aucun algorithme "rapide" qui renvoie toujours la meilleure solution. On peut toujours la vérifier (il suffit d'ajouter les poids)

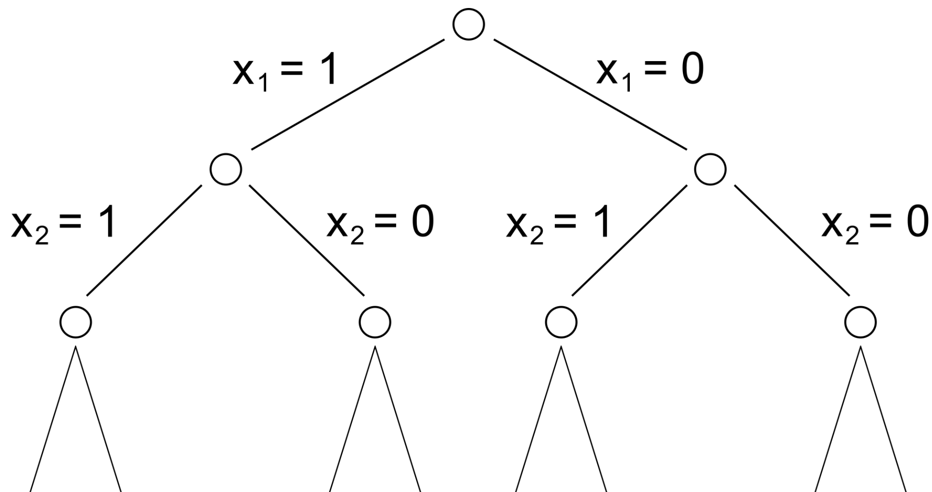
Recherche d'une solution

Si l'algorithme glouton ne donne pas la meilleure solution, comment faire ?

Une solution naïve consiste à explorer toutes les combinaisons possibles.

Arbre binaire

- rond : un poids, branche : une décision, triangle : un "sous-arbre"
- $x_1 = 1$ je prends le poids 1. $x_1 = 0$, je ne le prends pas.
- On note le poids total dans le rond et on explore le tout.



coût exponentiel

- À chaque fois qu'un objet est ajouté à la liste des objets disponibles, un niveau s'ajoute à l'arbre d'exploration binaire, et le nombre de cases est multiplié par 2.
- L'exploration de l'arbre et le remplissage des cases ont donc un coût qui croît exponentiellement avec le nombre n d'objets.
- 100 objets, 2^{100} solutions possibles…

  $2^{100} = 1267650600228229401496703205376$

$P = NP$?

Voilà une des questions à 1 million de $ du Clay Institute

Y répondre par l'affirmative (et donner les algorithmes polynomiaux…) changerait l'informatique actuelle. En particulier, la cryptographie qui repose essentiellement sur la difficulté qu'ont les machines à **trouver** des solutions et la grande simplicité qu'elles ont à les **vérifier**

- Donner deux entiers p et q tels que $p \times q = N$ avec $N = 60.186.563$.

  Vous avez 10 secondes…

Cryptographie

- $60.186.563 = 7.757 \times 7.759$

Comment le sais-je ? Et bien c'est facile je suis parti de $p = 7757$ et $q = 7759$…

- Ce sont deux *nombres premiers* (leurs diviseurs sont 1 et eux mêmes).
- Maintenant imaginez que p et q comportent 100 chiffres.

Clé publique, clé privée

- A et B (les gentils) doivent s'envoyer un message **privé**
- C (le méchant) veut **lire le message**.
- Un des (nombreux) procédés pour échanger des messages chiffrés repose sur les nombres premiers :
  - Facile de **vérifier** $p \times q = N$,
  - Difficile de **trouver** p et q avec seulement N.
- Il consiste à encoder un message avec la paire de clés p et q (qui sont très grands !). Ce sont les **clés privées.**
- A et B connaissent ces clés privées, ils n'ont pas de mal à chiffrer et déchiffrer.
- Le message qui transite sur le réseau, intercepté par C, ne fait apparaître que le nombre N (la **clé publique**).
- Mais, pour le déchiffrer, il doit connaître p et q.

  Il parviendra à factoriser, mais peut-être en 2000 ans…

Ça sert à quelque chose tout ça ?

C'est le fondement des échanges sur internet et donc de l'économie moderne.

Achat en ligne, services bancaires, e mail, chat, streaming… sont (ou devraient…) être sécurisés.

Un pirate n'a **aucun mal** à intercepter le flux de données. Mais, si celui-ci est convenablement chiffré, il ne peut rien en faire.

Et ça fonctionne ?

Oui… mais…

- Oui, ça fonctionne. Si vous envoyez via une messagerie sécurisée un message à votre voisin(e), personne ne pourra le lire.
- Mais il existe des entrées dérobées dans certains algorithmes. Et aussi des lois qui limitent la taillé des clés privées afin de laisser aux états la possibilité d'accéder aux contenus sensibles si nécessaire.
- Exemple : en 2015 le FBI a cherché pendant 6 mois à déverrouiller un iPhone 5C. Apple refusait de le faire (mais **pouvait** le faire) et le FBI a dû trouver un ingénieur compétent autrement.

Entrée dérobée ?

- C'est un moyen de contourner le problème brutal "trouver p et q tels que $N = p \times q$" et de déchiffrer l'information directement. Certains algorithmes en ont dès leur conception.
- Parfois c'est l'implémentation du programme qui en laisse (on parle alors de faille de sécurité)…
- Mais… dans $N = p \times q$, p et q sont des entrées dérobées !

  Oui ! Tout chiffrement comporte une entrée dérobée très difficile à obtenir.

Justement, les sacs à dos !

Le problème du sac à dos est $NP - complet$, c'est un bon candidat pour un algorithme de chiffrement !

Il fut le premier candidat (*Martin Hellman, Ralph Merkle et Whitfield Diffie - 1976*) pour ce rôle. Hélas aucune solution n'est fiable.

La difficulté repose dans le déchiffrement. Si le destinataire ne parvient pas à décoder, ça ne sert à rien. Il faut donc trouver un moyen de se ramener à une situation simple, où l'algorithme glouton fonctionne (poids super croissants !).

Et nous n'y sommes jamais parvenu. Chaque algorithme proposé qui résolve les 2 problèmes (*difficile à décoder sans la clé, facile à décoder avec la clé*) laisse derrière lui une *autre porte dérobée* qui rend le décodage facile pour un pirate.