NSI 1ère - Algorithmique - 1 - 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.
- Dans une machine un programme est un fichier (code source, exécutable etc.)
- Un algorithme est la description de ce que fait le programme.

calculabilité

- (*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'aura jamais 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 *polynomial* 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 connaît des algorithme *exponentiels* en temps 2^n.

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

Algorithme : vitesse

1 million de $ si vous résolvez la question !

4^{eme} 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 comparaisons

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
```

```python
        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

```python
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