

Cours 1

I Thème du cours

Définition : Par **algorithmique**, on entend la conception et l'analyse des algorithmes.

Définition : Un **algorithme** est une méthode systématique de résolution d'un problème.

Ce concept n'est pas limité à l'informatique (de nombreux algorithmes ont été écrits avant l'invention des ordinateurs).

La notion est devenue centrale en informatique avec l'apparition de machines capables d'exécuter **fidèlement** et **rapidement** une suite d'opérations prédéfinie.

Exemple : Plusieurs exemples d'algorithmes célèbres :

- Des algorithmes de calcul : *opérations arithmétiques, approximation de pi, $\sqrt{2}$, ...*
- Des constructions géométriques : *milieu d'un segment, triangle équilatéral, pentagone régulier, centre d'un cercle, ...*
- Des recettes de cuisine
- Des manuels de construction : *IKEA, lego, ...*

Trois axes d'études dans ce cours :

1. Conception d'algorithmes
y'a-t-il des techniques générales ?
2. Preuve de correction (ie. l'algorithme fait bien ce qu'on attend de lui).
Un algorithme est **correct** si pour chaque entrée, il **termine (1)** en produisant la **bonne sortie (2)**.
(on peut donc distinguer la preuve de terminaison et la preuve de correction proprement dite (appelée aussi correction partielle))
3. Étude de l'efficacité.
les ressources nécessaires (temps, mémoire) sont-elles raisonnables ? Est-il possible de faire mieux ?

Exemple : Addition de deux entiers : 1357 + 2468. On dessine ça :

$$\begin{array}{r} 1 & 3 & 5 & 7 \\ + & 2 & 4 & 6 & 8 \\ \hline \end{array}$$

Et on utilise des retenues et on additionne les colonnes de droite à gauche.

C'est un algorithme correct (on peut le prouver) et efficace (il est linéaire en la taille des entiers : 4 opérations au lieu de 3825 pour une addition naïve).

En Python, ça donne ça :

```

1 def addition(nb1, nb2):                      # deux entiers sous forme de tableau de taille égale
2     en commençant par les unités
3     res = []
4     retenue = 0
5     for(chiffre1, chiffre2) in zip(nb1, nb2) : # parcours en parallèle les tableaux
6         tmp = chiffre1 + chiffre2 + retenue
7         retenue = tmp // 10;                   # division euclidienne
8         res.append(tmp % 10)                  # ajout à la fin du tableau
9     return res + [retenue]                   # concaténation de deux tableaux
10
11 print(addition([7,5,3,1], [8,6,4,2]))    # affiche [5,2,8,3] pour 1357 + 2468 = 3825

```

On sait que le programme termine, car la boucle for itère un nombre fini de fois (la taille des tableaux).

Correction: en montrant l'invariant : "après i tours de boucle, $res \equiv n_1 + n_2[10^i]$ " (ie. les i premiers chiffres de la somme sont corrects). On peut faire une preuve par récurrence sur i .

Complexité en temps: autant d'additions élémentaires (ie. de chiffres) que de chiffres dans les nombres. Donc **linéaire EN la taille des entrées** (préciser linéaire en quoi est important, éviter les sous-entendus).

Donc ici, dire que " n_1 et n_2 sont de taille au plus l " signifie que $n_1, n_2 \in O(10^l)$, ou encore que $l = 1 + \lfloor \max(\log_{10}(n_1), \log_{10}(n_2)) \rfloor$.

💡 **Exemple :** Multiplication de deux entiers.

```

1 def multiplication_naive(nb1, nb2) :
2     # nb1 tableau de chiffres représentant un entier n1
3     # nb2 entier n2 représenté de manière usuelle (type int)
4     res = [0] * len(nb1) # tableau de 0 de longueur identique que nb1
5     for i in range(1, nb2+1) : # de i=1 à i=nb2, donc nb2 tours
6         res = addition(res, nb1)
7     return res

```

Correction: similaire à l'addition, en utilisant l'invariant "après l'étape i , $res \equiv n_1 \times i[10^l]$ " où l est la taille de n_1 .

Complexité en temps: n_2 additions, de grands entiers. Chaque addition est de coût linéaire en la taille du résultat, donc en $\log(n_1 \times n_2) = \log(n_1) + \log(n_2)$.

Complexité en $O(n_2 \times (\log(n_1 \cdot n_2)))$, soit $O(l \times 10^l)$ si n_1 et n_2 sont de taille l .

💡 **Remarque :** On manipulera toujours la partie entière supérieure de $\log_{10}(n)$ pour un entier n comme la "taille" de n .

sez z