Chapitre 1 : algorithmes, preuves et complexité

Plan

- 1/ Définitions
- 2/ Écriture d’un algorithme
- 3/ Preuve d’un algorithme
- 4/ Complexité d’un algorithme

1/ Définitions

- Algorithme
  - Définition informelle : étapes à suivre pour obtenir un résultat à partir d'une entrée (exemple : recette de cuisine)
  - Définition formelle : suite finie d'instructions permettant de donner un résultat au bout d'un temps fini, à partir d'une entrée
  - Un algorithme doit être compréhensible par un humain
  - Un algorithme doit être non ambigu (donc clair et précis)

1/ Définitions

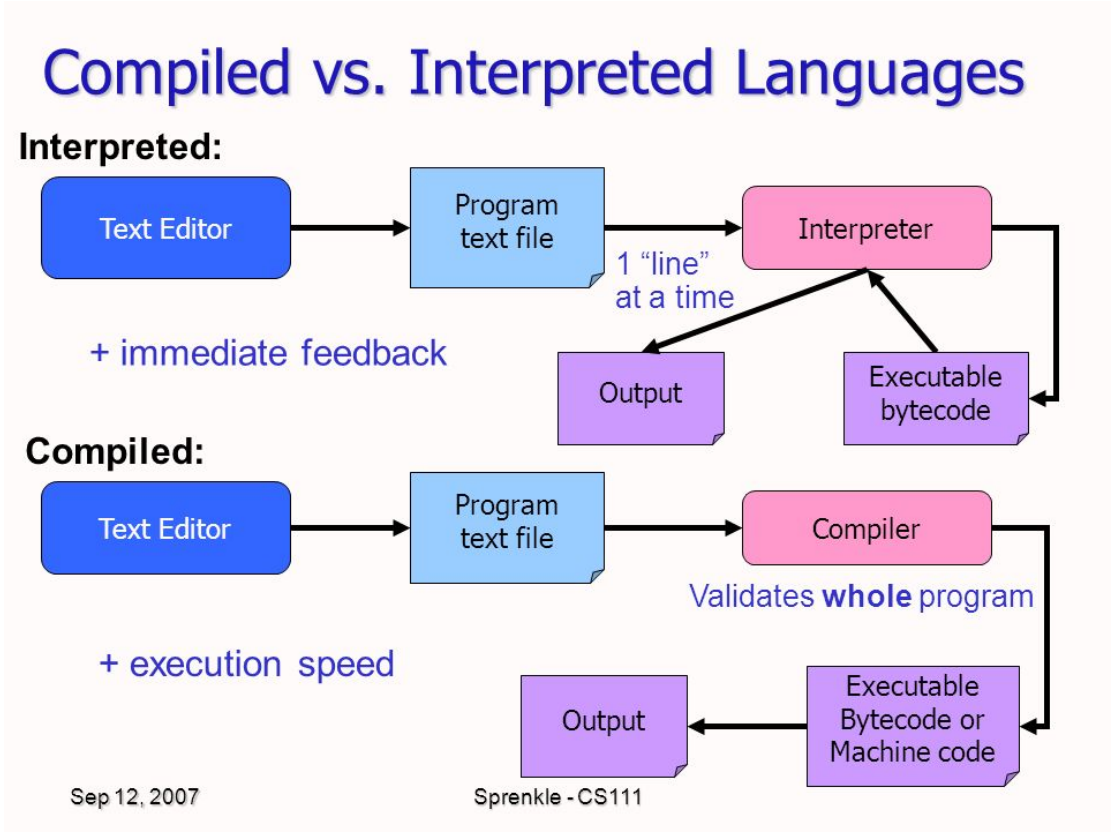
- Programme
  - Traduction d'un algorithme dans un langage compréhensible par un ordinateur
  - Un programme peut être exécuté sur un ordinateur
  - Le choix du langage de programmation dépend de plusieurs paramètres (langages connus par le programmeur, langages adaptés à certains problèmes, langages portables sur différents ordinateurs, etc.)
- Exemples de langages de programmation
  - Impératifs : C, Pascal
  - Fonctionnels : Lisp, Caml
  - Objets : Java, C++, C#, Python

1/ Définitions

- Python
  - Langage de programmation impératif objet
  - Langage multi-plateformes (donc portable)
  - Langage *open-source*
  - Langage le plus populaire au monde depuis 2021 (selon le classement Tiobe)
    - Popularité : 25.85% (11.80% pour le 2ème langage qui est le C, et 10.12% pour le 3ème langage qui est le Java)
    - Source : https://www.tiobe.com/tiobe-index/, consulté le 9/07/2025
- Remarques : Python est relativement simple à apprendre (car sa syntaxe est épurée)

1/ Définitions

- Python est un langage interprété (et non pas compilé)



Source : https://images.slideplayer.com/25/7615243/slides/slide_10.jpg, consulté le 9/07/2025

2/ Écriture d'un algorithme

- Comment écrire un algorithme ?
  - Un algorithme doit être compréhensible et non ambigu => on utilise généralement un langage « un peu » formel (mais pas trop)
- Exemple : calcul du PGCD (v1)
  - Version récursive : l'algorithme du PGCD appelle le PGCD

Algorithme PGCD (v1)
entrée : *a* et *b* deux entiers positifs
sortie : le PGCD de *a* et *b*
si *a*<*b* alors
    retourner PGCD(*b*,*a*)
sinon
    si *a*=*b* alors
        retourner *a*
    sinon
        retourner PGCD(*a*-*b*,*b*)
    finsi
finsi

2/ Écriture d'un algorithme

- Exemple : calcul du PGCD (v1) (suite)
  - L'algorithme définit l'entrée et la sortie attendue
  - L'algorithme décrit les étapes, avec quelques mots-clés simples (si…alors…sinon…finsi, retourner, etc.)

Algorithme PGCD (v1)
entrée : *a* et *b* deux entiers positifs
sortie : le PGCD de *a* et *b*
si *a*<*b* alors
    retourner PGCD(*b*,*a*)
sinon
    si *a*=*b* alors
        retourner *a*
    sinon
        retourner PGCD(*a*-*b*,*b*)
    finsi
finsi

2/ Écriture d'un algorithme

- Exemple : calcul du PGCD (v1) (suite)

Algorithme PGCD (v1)
entrée : *a* et *b* deux entiers positifs
sortie : le PGCD de *a* et *b*
si *a*<*b* alors
    retourner PGCD(*b*,*a*)
sinon
    si *a*=*b* alors
        retourner *a*
    sinon
        retourner PGCD(*a*-*b*,*b*)
    finsi
finsi

- Traduction en Python

```
def pgcd(a, b):
    if a < b:
        return pgcd(b, a)
    else:
        if a == b:
            return a
        else:
            return pgcd(a - b, b)
```

2/ Écriture d'un algorithme

- Exemple : calcul du PGCD (v1bis)
  - Version non récursive de l'algorithme précédent

Algorithme PGCD (v1bis)
entrée : *a* et *b* deux entiers positifs
sortie : le PGCD de *a* et *b*
tantque *a*!=*b* faire
    si *a*<*b* alors
        échanger *a* et *b*
    sinon
        a <— *a*-*b*
    finsi
fintantque
retourner *a*

2/ Écriture d’un algorithme

- Les mots-clés du langage algorithmique sont peu nombreux :
  - Condition : si…alors…sinon…finsi
  - Boucles : pour…faire…finpour, tantque…faire…fintantque
  - Retour : retourner

3/ Preuve d’un algorithme

- Un algorithme peut être prouvé formellement
  - Il ne s’agit pas d’un test (qui est la validation du déroulement d’un programme sur un jeu de données), mais bien la preuve de la validité de l’algorithme sur l’ensemble des données possibles
- On procède généralement en deux temps
  - Preuve de terminaison : l’algorithme produit un résultat en temps fini, quelle que soit l’entrée
  - Preuve de correction partielle : en supposant que l’algorithme s’arrête, le résultat fourni est celui attendu

3/ Preuve d’un algorithme

- Preuve de terminaison : on cherche à déterminer un « variant »
  - Un variant est une expression entière positive qui décroît strictement à chaque itération de la boucle
  - Si on trouve ce variant, la suite des valeurs prises par le variant est forcément finie
- Remarques
  - Pour les boucles « pour », le variant est très simple à trouver
  - Pour les boucles « tantque » et les fonctions récursives, le variant peut être difficile à trouver
  - Généralement, la preuve de terminaison est très simple

3/ Preuve d’un algorithme

- Preuve de correction partielle : on cherche à déterminer un « invariant »
  - Un invariant est une expression qui est vraie avant d’entrer dans la boucle, et à la fin de chaque tour de boucle
  - Comme la boucle termine (par hypothèse), l’invariant sera aussi vrai en fin de boucle
- Remarques
  - Pour prouver que l’invariant reste vrai entre le début de la boucle et la fin de la bouche, on procède instruction par instruction en adaptant l’expression vraie au fur et à mesure (ex : changement de variable, ajout de conditions, etc.), jusqu’à ce que l’expression finale soit l’invariant de la fin de la boucle
  - Généralement, la preuve de correction partielle est complexe

4/ Complexité d’un algorithme

- Un algorithme décrit une méthode de calcul
  - Plusieurs algorithmes peuvent mener au même résultat
  - Un algorithme permettant d’obtenir rapidement un résultat est souvent meilleur qu’un algorithme permettant de l’obtenir lentement

4/ Complexité d'un algorithme

- Exemple : PGCD-v1

```
Algorithme PGCD (v1)
entrée : a et b deux entiers positifs
sortie : le PGCD de a et b
si a<b alors
        retourner PGCD(b,a)
sinon
        si a=b alors
                retourner a
        sinon
                retourner PGCD(a-b,b)
        finsi
finsi
```

- Exemple : PGCD-v2 = algorithme d'Euclide

```
Algorithme PGCD (v2)
entrée : a et b deux entiers positifs
sortie : le PGCD de a et b
si b=0 alors
        retourner a
sinon
        retourner PGCD(b,a modulo b)
finsi
```

4/ Complexité d'un algorithme

- La complexité temporelle d'un algorithme est une mesure du nombre d'opérations nécessaires pour aboutir au résultat
  - Ce nombre d'opérations est indépendant de la machine qui exécutera le programme… Mais il donne néanmoins une estimation du temps de calcul
  - Ce nombre d'opérations dépend de la taille de l'entrée : on exprime donc la complexité en fonction de la taille de l'entrée
  - Ce nombre d'opérations dépend de la valeur de l'entrée : on calcule généralement la complexité dans le pire des cas
- On peut aussi calculer la complexité spatiale (= en mémoire) d'un algorithme, mais c'est plus rare
  - La complexité spatiale est toujours inférieure à la complexité temporelle

4/ Complexité d'un algorithme

- La complexité ne s'exprime généralement pas à l'opération près : on veut plutôt un ordre de grandeur (pour les grandes valeurs)
- Notation « grand-O » de Landau
  - $f(x) = O(g(x))$ si et seulement si il existe C et N tel que pour tout $x>=N$, $|f(x)| <= C.|g(x)|$

Si la complexité temporelle est en…	… alors on dit que l'algorithme est…
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(log(n))$	logarithmique
$O(2^n)$	exponentiel
$O(1)$	en temps constant

4/ Complexité d'un algorithme

- Simplifications
  - Comme on ne s'intéresse qu'au comportement asymptotique du nombre d'opérations, on peut effectuer beaucoup de simplifications
- Exemples :
  - $O(n^2+n)=O(n^2)$
  - $O(n^2+1000n)=O(n^2)$
  - $O(1000)=O(1)$
  - $O(n^3)+1000.O(n)=O(n^3+1000.n)=O(n^3)$

4/ Complexité d'un algorithme

- Calcul de la complexité de l'algorithme PGCD-v1 :
  - L'algorithme est linéaire
- Calcul de la complexité de l'algorithme d'Euclide (PGCD-v2) :
  - Rappels sur la suite de Fibonacci : soit $F_0=0$, $F_1=1$, $F_{n+2}=F_{n-1}+F_{n-2}$ pour tout $n>=0$, alors $F_n=(1/sqrt(5)).or^n$, avec $or=(1+sqrt(5))/2$
  - Théorème de Lamé : Soit $d=PGCD(a,b)$ avec $a>=b$, et soit k le nombre d'étapes pour calculer d avec l'algorithme d'Euclide. Alors $a>=d.F_{k+2}$ et $b>=d.F_{k+1}$. Ce théorème se démontre par récurrence sur k.
  - On en conclut que $k<=2.08ln(b)+1$, donc l'algorithmique est logarithmique en la taille de l'entrée (a et b).

4/ Complexité d'un algorithme

- Théorème de Master : si la complexité d'un algorithme s'écrit $T(n)=aT(n/b)+O(n^d)$, avec $a>=1$, $b>1$, et d un entier, alors l'expression de $T(n)$ avec la notation O est :
  - Si $d<log_b(a)$, alors $T(n)=O(n^{log\_b(a)})$
  - Si $d=log_b(a)$, alors $T(n)=O(n^d.log_b(n))$
  - Si $d>log_b(a)$, alors $T(n)=O(n^d)$