Analyse des algorithmes

Terminaison des algorithmes

Lycée Fabert – PCSI2

2025–2026

Trois questions fondamentales

Tout algorithme doit répondre à trois questions :

- **Terminaison** : s'arrête-t-il toujours ?
- **Correction** : fournit-il le bon résultat ?
- **Complexité** : quelles ressources sont nécessaires ?

Objectifs du cours

- Comprendre ce qu'est la terminaison d'un algorithme
- Savoir prouver la terminaison :
  - des boucles `for`
  - des boucles `while`
  - des algorithmes récursifs
- Introduire les notions de variant et d'invariant de boucle

Exemple introductif : les jetons

- Une boîte contenant des jetons rouges et bleus
- À chaque étape :
  - on retire deux jetons
  - on en remet un seul selon des règles précises
- Question :
  - le jeu se termine-t-il toujours ?
  - peut-on prévoir la couleur du dernier jeton ?

Idée clé : variant de boucle

Définition

Un **variant de boucle** est une expression :

- à valeurs dans $\mathbb{N}$
- strictement décroissante à chaque itération

Idée clé : variant de boucle

Définition

Un **variant de boucle** est une expression :

- à valeurs dans $\mathbb{N}$
- strictement décroissante à chaque itération

Conséquence

S'il existe une suite infinie strictement décroissante d'entiers naturels, alors la boucle termine.

Pour savoir quelle sera la fin : Invariant de boucle

Définition

Un **Invariant de boucle** est une propriété

- qui est vraie au départ
- qui reste vraie à chaque itération

Boucle par boucle

On peut montrer la terminaison d'un algorithme, en montrant la terminaison boucle par boucle. Il faut alors montrer que le nombre de boucle est fini.
Dans le cas d'un algorithme récursif, on doit vérifier que le nombre d'appels est fini.

Boucle `for`

- Nombre d'itérations fixé à l'avance, donc fini.
- Chaque itération contient un nombre fini d'instructions

Conclusion

Une boucle `for` termine toujours, sauf modification de l'objet itéré, c'est à dire erreur manifeste de code.

Exemple : factorielle

```
def fact_iter(n):
    res = 1
    for i in range(2, n+1):
        res = res * i
    return res
```

- Le nombre d'itérations est fini
- La boucle termine donc toujours

Boucle `while`

- Le nombre d'itérations n'est pas connu à l'avance
- Il faut exhiber un **variant de boucle**

Exemple : puissance

```
def puissance(a, n):
    p = 1
    c = n
    while c > 0:
        p = p * a
        c = c - 1
    return p
```

- Le compteur c est un variant
- Il est positif et décroît strictement

Exemple : PGCD

On veut faire un algorithme qui calcule le PGCD de $a, b \in \mathbb{N}$.

Exemple : PGCD

On veut faire un algorithme qui calcule le PGCD de $a, b \in \mathbb{N}$.
On utilise évidemment l'algorithme d'Euclide !

Exemple : PGCD

On veut faire un algorithme qui calcule le PGCD de $a, b \in \mathbb{N}$.
On utilise évidemment l'algorithme d'Euclide !

- Ni a ni b ne sont des variants

Exemple : PGCD

On veut faire un algorithme qui calcule le PGCD de $a, b \in \mathbb{N}$.
On utilise évidemment l'algorithme d'Euclide !

- Ni a ni b ne sont des variants
- Le variant est la somme $u + v$ des deux entiers, suite des divisions euclidiennes successives.

$u + v$ est positif et décroît strictement.

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Recherche dichotomique

- Recherche dans une liste triée

Recherche dichotomique

- Recherche dans une liste triée
- Variant : taille de l'intervalle de recherche

Recherche dichotomique

- Recherche dans une liste triée
- Variant : taille de l'intervalle de recherche $N = d - g + 1$.
- N décroît strictement à chaque itération

Recherche dichotomique

- Recherche dans une liste triée
- Variant : taille de l'intervalle de recherche $N = d - g + 1$.
- N décroît strictement à chaque itération

```
def rechercheDicho(L, x):
    g = 0
    d = len(L) - 1

    while g <= d:
        m = (g + d) // 2
        if L[m] == x:
            return m
        elif L[m] < x:
            g = m + 1
        else:
            d = m - 1
    return -1
```

Suite de Syracuse

- Définition simple
- Observation expérimentale : la suite atteint toujours le cycle 4-2-1.
- Mais

Aucun variant connu $\Rightarrow$ terminaison non démontrée

Algorithmes récursifs

- Une fonction s'appelle elle-même
- Nécessité d'un **cas d'arrêt**

Idée

Un paramètre doit décroître strictement à chaque appel récursif

Exemple : factorielle récursive

```
def fact_rec(n):
    if n == 0:
        return 1
    else:
        return n * fact_rec(n-1)
```

- Le paramètre n décroît
- L'appel récursif s'arrête en $n = 0$

À retenir

- La terminaison doit être prouvée
- Outils principaux :
  - variant de boucle
  - cas d'arrêt pour la récursivité
- Tester un algorithme ne suffit pas à prouver sa terminaison

Exercice 1 – Terminaison

Étudier la terminaison des algorithmes suivants et déterminer, le cas échéant, le nombre d'itérations de la boucle.

Algorithme 1

```
n = 100
a = 1
while a < n:
    a = a + 5
```

Exercice 1 – Terminaison

Étudier la terminaison des algorithmes suivants et déterminer, le cas échéant, le nombre d'itérations de la boucle.

Algorithme 1

```
n = 100
a = 1
while a < n:
    a = a + 5
```

Algorithme 2

```
n = 10**8
a = 1
while a < n:
    a = 2 * a
```

Exercice 1 – Suite

Algorithme n°3

```
a = 100
while a >= 0:
    a = a // 2
```

- La boucle termine-t-elle ?
- Identifier un éventuel variant de boucle.

Exercice 2 – Variant de boucle

On considère l'algorithme suivant, où p est un entier donné :

```
c = 0
while p > 0:
    if c == 0:
        p = p - 2
        c = 1
    else:
        p = p + 1
        c = 0
```

1. Tester l'exécution pour $p = -3$ puis $p = 5$.
2. La variable p ou c est-elle un variant de boucle ?
3. Montrer que $3p + 2c$ est un variant de boucle.

Exercice 3 – Non terminaison

On considère la fonction suivante

```
def func(c):
    p = 1
    if c > 0:
        while c != 0:
            p = p * 2
            c = c - 2
    return p
```

- Montrer que c n'est pas un variant de boucle.
- Déterminer les valeurs de c pour lesquelles l'algorithme ne termine pas.

Exercice 4 – Boucles `for` en Python

Les algorithmes suivants terminent-ils ? Justifier puis tester.

Algorithme n°1

```
n = 5
for i in range(n):
    print(i)
    n = n + 1
```

Exercice 4 – Boucles `for` en Python

Les algorithmes suivants terminent-ils ? Justifier puis tester.

Algorithme n°1

```
n = 5
for i in range(n):
    print(i)
    n = n + 1
```

Algorithme n°2

```
n = 5
for i in range(n):
    print(i)
    i = i - 1
```

Exercice 4 – Recherche dans une chaîne

On considère la fonction suivante

```
def recherche(ch, car):
    i = len(ch) - 1
    while ch[i] != car:
        i = i - 1
    return i
```

1. Tester pour `"abracadabra"` avec `"d"`, `"r"`, `"e"`.
2. La boucle termine-t-elle toujours ?
3. Corriger l'algorithme en utilisant le caractère paresseux de `and`.

Trois questions fondamentales

Tout algorithme doit répondre à trois questions :

- **Terminaison** : s'arrête-t-il toujours ?
- **Correction** : fournit-il le bon résultat ?
- **Complexité** : quelles ressources sont nécessaires ?

Objectifs du chapitre

- Comprendre la notion de correction d'un algorithme
- Distinguer correction partielle et correction totale
- Utiliser les invariants de boucle pour raisonner

Invariant de boucle

Définition

- vraie avant l'entrée dans la boucle
- conservée à chaque itération

Correction partielle / totale

- Correction partielle : résultat correct, quand l'algorithme termine
- Correction totale : terminaison + résultat conforme

Exemple : puissance

```
def puissance(a, n):
    p = 1
    c = n
    while c > 0:
        p = p * a
        c = c - 1
    return p
```

Exemple : puissance

```
def puissance(a, n):
    p = 1
    c = n
    while c > 0:
        p = p * a
        c = c - 1
    return p
```

Quel est l'invariant ?

Exemple : division euclidienne

```
def division(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

Exemple : division euclidienne

```
def division(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

Quel est l'invariant ?

Exemple : division euclidienne

```
def division(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

Quel est l'invariant ?
C'est la propriété

$$a = bq + r \quad (r \geq 0)$$

Exemple : minimum d'une liste

```
def minimum(L):
    assert L!=[]
    m = 0
    for i in range(1, len(L)):
        if L[i] < L[m]:
            m = i
    return m
```

Exemple : minimum d'une liste

```
def minimum(L):
    assert L!=[]
    m = 0
    for i in range(1, len(L)):
        if L[i] < L[m]:
            m = i
    return m
```

Quel est l'invariant ?

Exemple : minimum d'une liste

```
def minimum(L):
    assert L!=[]
    m = 0
    for i in range(1, len(L)):
        if L[i] < L[m]:
            m = i
    return m
```

Quel est l'invariant ?

C'est la propriété

$$L[m_k] = \min(L[0], \ldots, L[k])$$

Exemple : tri par sélection

```
def tri_selection(L):
    n = len(L)
    for j in range(n-1):
        m = j
        for i in range(j+1, n):
            if L[i] < L[m]:
                m = i
        if m != j:
            L[j], L[m] = L[m], L[j]
```

Invariant du tri

Après k itérations :

- les k premiers éléments sont triés
- ils sont inférieurs aux suivants

Jeux de tests

- Vérifier le comportement du programme
- Tester cas généraux et cas limites

Exemple de tests

```
assert division(30, 7) == (4, 2)
assert division(120, 5) == (24, 0)
```

Conclusion

- Les invariants sont l'outil clé pour la correction
- Une preuve se fait par initialisation et hérédité

Exercice 1 – Puissance

On considère l'algorithme suivant :

```
def puissance(a, n):
    p = 1
    c = n
    while c > 0:
        p = p * a
        c = c - 1
    return p
```

1. Proposer un invariant de boucle.
2. Montrer qu'il est vérifié à l'initialisation.
3. Montrer qu'il est conservé.
4. Conclure sur la correction.

Exercice 2 – Division euclidienne

```
def division(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

1. Donner un invariant de boucle.
2. Montrer qu'il est conservé.
3. Montrer que le résultat vérifie :

$$a = bq + r \quad \text{et} \quad 0 \leq r < b$$

Exercice 3 – Algorithme incorrect

```
def puissance(a, n):
    p = 1
    c = n
    while c > 0:
        p = p * a
    return p
```

- Cet algorithme est-il correct ?
- Identifier le problème.
- Corriger l'algorithme.

Exercice 4 – Minimum

```
def minimum(L):
    m = 0
    for i in range(1, len(L)):
        if L[i] < L[m]:
            m = i
    return m
```

1. Proposer un invariant de boucle.
2. Interpréter cet invariant.
3. Conclure sur la correction.

Exercice 5 – Tri par sélection

```
def tri_selection(L):
    n = len(L)
    for j in range(n-1):
        m = j
        for i in range(j+1, n):
            if L[i] < L[m]:
                m = i
        L[j], L[m] = L[m], L[j]
```

1. Donner un invariant pour la boucle externe.
2. Interpréter cet invariant.
3. Conclure sur la correction.

Objectifs du chapitre

- Comprendre la notion de complexité d'un algorithme
- Évaluer le coût en temps d'exécution
- Comparer différents algorithmes
- Introduire la notation asymptotique

Complexité d'un algorithme

Définition

La complexité mesure les ressources nécessaires à l'exécution d'un algorithme.

Complexité d'un algorithme

Définition

La complexité mesure les ressources nécessaires à l'exécution d'un algorithme.

- Temps d'exécution (complexité temporelle)
- Mémoire utilisée (complexité spatiale)

Opérations élémentaires

On compte le nombre d'opérations élémentaires :

- affectation
- comparaison
- opération arithmétique
- accès à un tableau (lecture) et ajout (append)
- opération logique
- affichage (print)

Opérations élémentaires

On compte le nombre d'opérations élémentaires :

- affectation
- comparaison
- opération arithmétique
- accès à un tableau

Idée

Toutes ces opérations sont supposées de coût constant.

Cela reste une approximation.

Paramètre de taille

- La complexité dépend de la taille des données
- On note généralement cette taille n

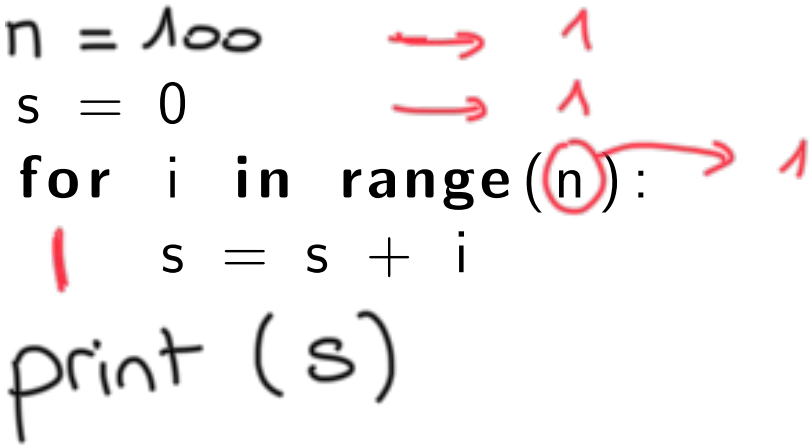
Paramètre de taille

- La complexité dépend de la taille des données
- On note généralement cette taille n

Examples

- longueur d'une liste
- nombre de bits d'un entier

Exemple



n
4 x n
1

5n + 4

Exemple

```
s = 0
for i in range(n):
    s = s + i
```

- boucle exécutée n fois
- nombre d'opérations proportionnel à n

$$T(n) \sim n$$

Notation O

Définition

On écrit :

$$T(n) = O(f(n))$$

si $T(n)$ est majorée par $f(n)$ à constante près.

Notation O

Définition

On écrit :

$$T(n) = O(f(n))$$

si $T(n)$ est majorée par $f(n)$ à constante près.

- on garde le terme dominant
- on néglige constantes et termes faibles

Exemple

$$T(n) = 5n^2 + 3n + 2$$

Exemple

$$T(n) = 5n^2 + 3n + 2$$

$$T(n) = O(n^2)$$

Exemple

$$T(n) = 5n^2 + 3n + 2$$

$$T(n) = O(n^2)$$

Idée

Le terme dominant est n^2.

Ordres de grandeur

- $O(1)$: constant
- $O(\log n)$: logarithmique
- $O(n)$: linéaire
- $O(n \log n)$: linéarithmique.
- $O(n^2)$: quadratique
- $O(2^n)$: exponentielle

Types de complexité

- pire cas (majoration)
- meilleur cas
- cas moyen

Types de complexité

- pire cas (majoration)
- meilleur cas
- cas moyen

Remarque

On s'intéresse le plus souvent au pire cas.

Recherche séquentielle

L triée

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

elif L[i] > x
return -1

Complexité

- pire cas : on parcourt toute la liste

$$O(n)$$

Recherche dichotomique

```
def rechercheDicho(L, x):
    g = 0
    d = len(L) - 1
    while g <= d:
        m = (g + d) // 2
        if L[m] == x:
            return m
        elif L[m] < x:
            g = m + 1
        else:
            d = m - 1
    return -1
```

Complexité

- taille divisée par 2 à chaque étape

$$O(\log n)$$

Double boucle

```
for i in range(n):
    for j in range(n):
        pass
```

Double boucle

```
for i in range(n):
    for j in range(n):
        pass
```

$$O(n^2)$$

Tri à bulles

```
def tri_bulles(L):
    n = len(L)
    for k in range(1, n):
        for i in range(n-k):
            if L[i] > L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]
```

Tri à bulles

```
def tri_bulles(L):
    n = len(L)
    for k in range(1, n):
        for i in range(n-k):
            if L[i] > L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]
```

$$O(n^2)$$

Tri fusion

- Diviser la liste en deux
- Trier récursivement
- Fusionner

Tri fusion

- Diviser la liste en deux
- Trier récursivement
- Fusionner

$$O(n \log n)$$

À retenir

- La complexité permet de comparer les algorithmes
- On raisonne en ordre de grandeur
- Les bons algorithmes font une énorme différence

Exercice 1 – Complexité

```
s = 0
for i in range(n):
    s = s + i
```

- Déterminer le nombre d'itérations.
- Donner la complexité en $O(\cdot)$.

Exercice 2 – Boucles imbriquées

```
for i in range(n):
    for j in range(n):
        s = s + 1
```

- Combien d'itérations au total ?
- En déduire la complexité.

Exercice 3 – Variante

```
for i in range(n):
    for j in range(i):
        s = s + 1
```

- Calculer le nombre total d'itérations.
- En déduire la complexité.

Exercice 4 – Recherche

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

- Complexité dans le pire cas ?
- Complexité dans le meilleur cas ?

Exercice 5 – Dichotomie

On considère la recherche dichotomique.

- Comment évolue la taille du problème ?
- Combien d'itérations au maximum ?
- En déduire la complexité.

Exercice 6 – Comparaison

Comparer les complexités suivantes :

- n
- $\log n$
- n^2
- $n \log n$

Exercice 6 – Comparaison

Comparer les complexités suivantes :

- n
- $\log n$
- n^2
- $n \log n$

- Classer par ordre de croissance
- Discuter l'impact pour n grand