ALGORITHMIQUE ET STRUCTURES DE DONNÉES

1. Introduction à l’algorithmie et la complexité

Yoann Coudert--Osmont

26 Janvier 2026

Définition

Algorithme

Un algorithme est une suite finie d'instructions permettant de résoudre un problème.

Définition

Algorithme

Un algorithme est une suite finie d'instructions permettant de résoudre un problème.

Exemples de problèmes

- Trouver le chemin le plus court entre deux points sur une carte (GPS).
- Affecter les nouveaux étudiants aux établissements de l'enseignement supérieur (Parcoursup).
- Chercher les pages web les plus pertinentes pour une séquence de termes donnée (Google, Bing, DuckDuckGo).

Objectif

En algorithmie on cherche généralement à construire des algorithmes efficaces. C'est à dire, des algorithmes ayant une faible complexité.

Complexité

La **complexité temporelle** d'un algorithme est le nombre d'opération élémentaires réalisées durant son exécution.
La **complexité spatiale** d'un algorithme est la quantité de mémoire nécessaire pour son exécution.

Objectif

En algorithmie on cherche généralement à construire des algorithmes efficaces. C'est à dire, des algorithmes ayant une faible complexité.

Complexité

La **complexité temporelle** d'un algorithme est le nombre d'opération élémentaires réalisées durant son exécution.
La **complexité spatiale** d'un algorithme est la quantité de mémoire nécessaire pour son exécution.

Autres objectifs

- Parallélisation pour les ordinateurs récents (CPU multi-cœurs, GPU, ...)
- l'Ergonomie, la lisibilité du code, la compréhensibilité.
- ...

Plan

1. Complexité

2. Exemples
    - Sous-tableau de somme maximale
    - Exponentiation rapide

3. Théorème maître

Complexité

Définitions

Un algorithme $\mathcal{A}$ est en quelque sorte une fonction qui prend une entrée x et retourne une sortie $\mathcal{A}(y)$.

Exemple

Pour la recherche de plus court chemin entre deux points d'une carte, l'**entrée** se compose d'un *graphe* (un ensemble de routes reliant des points) et de 2 points A et B appartenant à ce graphe entre lesquels on cherche un plus court chemin. La **sortie** est un chemin entre A et B (une succession de routes entre A et B).

Définitions

Complexité temporelle

La complexité temporelle $C_{\mathcal{A}}$ d'un algorithme $\mathcal{A}$ est une fonction retournant le nombre d'opérations effectuées par l'algorithme $C_{\mathcal{A}}(x)$ pour une entrée x.

Définitions

Complexité temporelle

La complexité temporelle $C_{\mathcal{A}}$ d'un algorithme $\mathcal{A}$ est une fonction retournant le nombre d'opérations effectuées par l'algorithme $C_{\mathcal{A}}(x)$ pour une entrée x.

Exemple

```python
def somme(L):
    s = 0       # exécutée 1 fois
    for x in L:
        s += x # exécutée len(L) fois
    return s
```

Ici, $C_{\mathcal{A}}(L) = 1 + \mathrm{len}(L)$.

Dépend uniquement de $\mathrm{len}(L)$!!!

Reparamétrisation de la complexité

Complexité temporelle

La complexité temporelle $C_{\mathcal{A}}$ d'un algorithme $\mathcal{A}$ est une fonction retournant le nombre d'opérations effectuées par l'algorithme $C_{\mathcal{A}}(n)$ pour ~~une entrée x~~ **une taille d'entrée** n.

Reparamétrisation de la complexité

Complexité temporelle

La complexité temporelle $C_{\mathcal{A}}$ d'un algorithme $\mathcal{A}$ est une fonction retournant le nombre d'opérations effectuées par l'algorithme $C_{\mathcal{A}}(n)$ pour ~~une entrée x~~ **une taille d'entrée** n.

Exemple

```python
def trouver(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i
    return None
```

Ici, $C^e_{\mathcal{A}}([3,5,4],3) = 1$ et $C^e_{\mathcal{A}}([3,5,4],4) = 3$.
Avec une taille d'entrée $n = \text{len}(L)$, $C_{\mathcal{A}}(3)$ ne semble pas bien défini ...
$C_{\mathcal{A}}(3) = 1$? ou $C_{\mathcal{A}}(3) = 3$?

Reparamétrisation de la complexité

Lorsque la complexité ne dépend pas que de la taille de l'entrée, il est nécessaire d'apporter plus de précision sur ce que l'on mesure.

Reparamétrisation de la complexité

Lorsque la complexité ne dépend pas que de la taille de l'entrée, il est nécessaire d'apporter plus de précision sur ce que l'on mesure.

- **Complexité dans le pire cas** : La façon la plus courante de mesurer la complexité pour une taille d'entrée n est de donner le nombre d'opérations effectuées dans le pire cas. C'est à dire :

$$C_{\mathcal{A}}^{\text{pire}}(n) = \max_{\text{taille}(x)=n} C_{\mathcal{A}}^{e}(x)$$

Reparamétrisation de la complexité

Lorsque la complexité ne dépend pas que de la taille de l'entrée, il est nécessaire d'apporter plus de précision sur ce que l'on mesure.

- **Complexité dans le pire cas** : La façon la plus courante de mesurer la complexité pour une taille d'entrée n est de donner le nombre d'opérations effectuées dans le pire cas. C'est à dire :

$$C_{\mathcal{A}}^{\text{pire}}(n) = \max_{\text{taille}(x)=n} C_{\mathcal{A}}^{e}(x)$$

- **Complexité en moyenne** : Il arrive parfois de considérer l'espérance de la complexité selon une certaine distribution pour une taille donnée.

$$C_{\mathcal{A}}^{\text{moyen}}(n) = \mathbb{E}_x\left[C_{\mathcal{A}}^{e}(x)\right]$$

Reparamétrisation de la complexité

Lorsque la complexité ne dépend pas que de la taille de l'entrée, il est nécessaire d'apporter plus de précision sur ce que l'on mesure.

- **Complexité dans le pire cas** : La façon la plus courante de mesurer la complexité pour une taille d'entrée n est de donner le nombre d'opérations effectuées dans le pire cas. C'est à dire :

$$C_{\mathcal{A}}^{\text{pire}}(n) = \max_{\text{taille}(x)=n} C_{\mathcal{A}}^{e}(x)$$

- **Complexité en moyenne** : Il arrive parfois de considérer l'espérance de la complexité selon une certaine distribution pour une taille donnée.

$$C_{\mathcal{A}}^{\text{moyen}}(n) = \mathbb{E}_x\left[C_{\mathcal{A}}^{e}(x)\right]$$

Par défaut, on utilisera la **complexité dans le pire cas**.

Reparamétrisation de la complexité

Exemple

```python
def trouver(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i
    return None
```

$$C_{\mathcal{A}}^{\text{pire}}(n) = n \qquad et, \qquad C_{\mathcal{A}}^{\text{moyen}}(n) = \frac{n}{2}$$

Complexité asymptotique

- Il est souvent difficile de mesurer exactement le nombre d'opérations effectuées.
- Comment comparer deux algorithmes $\mathcal{A}$ et $\mathcal{B}$? Pour quelles tailles d'entée n faut-il les comparer ?

Complexité asymptotique

- Il est souvent difficile de mesurer exactement le nombre d'opérations effectuées.
- Comment comparer deux algorithmes $\mathcal{A}$ et $\mathcal{B}$? Pour quelles tailles d'entée n faut-il les comparer ?

Solution: Comportement asymptotique lorsque $\boldsymbol{n \to +\infty}$.

Domination (Grand $\mathcal{O}$)

Domination

Soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. On dit que f est **dominée** par g et l'on note $\boldsymbol{f(x) = \mathcal{O}(g(x))}$ lorsqu'il existe deux constantes N et C tel que $\forall x > N,\ |f(x)| \leqslant C|g(x)|$.

Domination (Grand $\mathcal{O}$)

Domination

Soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. On dit que f est **dominée** par g et l'on note $\boldsymbol{f(x) = \mathcal{O}(g(x))}$ lorsqu'il existe deux constantes N et C tel que
$\forall x > N,\ |f(x)| \leqslant C|g(x)|$.

Exemples

- $2x^3 + 20x^2 - 5x + 100 = \mathcal{O}(x^3)$
- $n\log(n) = \mathcal{O}(n^2)$
- $n^{1000} = \mathcal{O}(2^n)$

Notations Ω et Θ

Notations Ω et Θ

Soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$.

- On dit que f est **minorée** par g et l'on note $\boldsymbol{f(x) = \Omega(g(x))}$ lorsque g est dominée par f (*i.e.* lorsque $g(x) = \mathcal{O}(f(x))$).
- On dit que f et g sont **du même ordre de grandeur** et l'on note $\boldsymbol{f(x) = \Theta(g(x))}$ lorsque $f(x) = \mathcal{O}(g(x))$ et $f(x) = \Omega(g(x))$.

Notations Ω et Θ

Notations Ω et Θ

Soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$.

- On dit que f est **minorée** par g et l'on note $\boldsymbol{f(x) = \Omega(g(x))}$ lorsque g est dominée par f (*i.e.* lorsque $g(x) = \mathcal{O}(f(x))$).
- On dit que f et g sont **du même ordre de grandeur** et l'on note $\boldsymbol{f(x) = \Theta(g(x))}$ lorsque $f(x) = \mathcal{O}(g(x))$ et $f(x) = \Omega(g(x))$.

Exemples

- $2x^3 + 5x = \Theta(x^3)$
- $\sqrt{n} = \Omega(\log(n))$
- $n^n = \Omega(n!)$

Complexité asymptotique

On utilisera couramment la notation grand $\mathcal{O}$ pour donner la complexité asymptotique des algorithmes.
Les fonctions `somme` et `trouver` précédentes sont ainsi des algorithmes que l'on qualifie de linéaires avec une complexité $C_{\mathcal{A}}(n) = \mathcal{O}(n)$.

Lien entre complexité et temps d'exécution

L'ordre de grandeur de la fréquence des CPU actuels est le GHz.
C'est à dire qu'ils exécutent environs 10^9 opérations par seconde.

Complexité	$n = 1$	$n = 10$	$n = 10^3$	$n = 10^6$	**Exemple**
$\mathcal{O}(1)$	1 ns	1 ns	1 ns	1 ns	addition
$\mathcal{O}(\log(n))$	1 ns	3 ns	10 ns	20 ns	dichotomie
$\mathcal{O}(\sqrt{n})$	1 ns	3 ns	30 ns	1 μs	test de primalité naïf
$\mathcal{O}(n)$	1 ns	10 ns	1 μs	1 ms	parcours de liste
$\mathcal{O}(n\log(n))$	1 ns	30 ns	10 μs	20 ms	tri de tableau
$\mathcal{O}(n^2)$	1 ns	100 ns	1 ms	20 min	3SUM
$\mathcal{O}(n^3)$	1 ns	1 μs	1 s	30 ans	multiplication matricielle
$\mathcal{O}(2^n)$	1 ns	1 μs	10^{284} ans	...	SAT

Exemples

Sous-tableau de somme maximale

Problème

Étant donné un tableau L, contenant n nombres, notre tâche est de calculer la plus grande somme possible de valeurs consécutives dans le tableau.

-1	2	4	-3	5	2	-5	2

Sous-tableau de somme maximale

Problème

Étant donné un tableau L, contenant n nombres, notre tâche est de calculer la plus grande somme possible de valeurs consécutives dans le tableau.

$$\boxed{-1}\,\underbrace{\boxed{2}\,\boxed{4}\,\boxed{-3}\,\boxed{5}\,\boxed{2}}_{\Sigma = 10}\,\boxed{-5}\,\boxed{2}$$

Algorithme 1

```python
def Algo1(L):
    n = len(L)
    somme_max = 0
    for a in range(n):
        for b in range(a, n):
            somme = 0
            for i in range(a, b+1):
                somme += L[i]
            somme_max = max(somme_max, somme)
    return somme_max
```

Algorithme 1

```python
def Algo1(L):
    n = len(L)
    somme_max = 0
    for a in range(n):
        for b in range(a, n):
            somme = 0
            for i in range(a, b+1):
                somme += L[i]
            somme_max = max(somme_max, somme)
    return somme_max
```

Analyse de la complexité :

3 boucles imbriquées pouvant itérer jusqu'à n valeurs chacune $\Longrightarrow \mathcal{O}(n^3)$.

Algorithme 1

```python
def Algo1(L):
    n = len(L)
    somme_max = 0
    for a in range(n):
        for b in range(a, n):
            somme = 0
            for i in range(a, b+1):
                somme += L[i]
            somme_max = max(somme_max, somme)
    return somme_max
```

Analyse de la complexité plus précise:

$$C_{\mathcal{A}_1}(n) = 2 + \sum_{a=0}^{n-1} \sum_{b=a}^{n-1} \left(2 + \sum_{i=a}^{b} 1 \right) = \frac{1}{6} \left[n^3 + 9n^2 + 8n + 12 \right]$$

Algorithme 2

```python
def Algo2(L):
    n = len(L)
    somme_max = 0
    for a in range(n):
        somme = 0
        for b in range(a, n):
            somme += L[b]
            somme_max = max(somme_max, somme)
    return somme_max
```

Algorithme 2

```python
def Algo2(L):
    n = len(L)
    somme_max = 0
    for a in range(n):
        somme = 0
        for b in range(a, n):
            somme += L[b]
            somme_max = max(somme_max, somme)
    return somme_max
```

Analyse de la complexité :

2 boucles imbriquées pouvant itérer jusqu'à n valeurs chacune $\Longrightarrow \mathcal{O}(n^2)$.

Algorithme 3

Algorithme 3

```python
def Algo3(L):
    n = len(L)
    somme_max = 0
    somme = 0
    for i in range(n):
        somme = max(0, somme) + L[i]
        somme_max = max(somme_max, somme)
    return somme_max
```

Algorithme 3

```python
def Algo3(L):
    n = len(L)
    somme_max = 0
    somme = 0
    for i in range(n):
        somme = max(0, somme) + L[i]
        somme_max = max(somme_max, somme)
    return somme_max
```

Analyse de la complexité :

Une seule boucle de taille $n \Longrightarrow \mathcal{O}(n)$.

Calcul de x^n

Problème

Étant donné une valeur x, et un entier n, notre tâche est de calculer x^n.

Algorithme naïf

```python
def exponentiation_naive(x, n):
    y = 1
    for _ in range(n):
        y *= x
    return y
```

Algorithme naïf

Algorithme naïf

```python
def exponentiation_naive(x, n):
    y = 1
    for _ in range(n):
        y *= x
    return y
```

Analyse de la complexité :

Une seule boucle de taille $n \Longrightarrow \mathcal{O}(n)$.

Exponentiation rapide

```python
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    else:
        y = exponentiation_rapide(x, n//2)
        z = y * y
        if n%2 == 1:
            z *= x
        return z
```

Exponentiation rapide

Exponentiation rapide

```
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    else:
        y = exponentiation_rapide(x, n//2)
        z = y * y
        if n%2 == 1:
            z *= x
        return z
```

Analyse de la complexité :

$$C_{\mathcal{A}}(0) = 0 \qquad et \qquad C_{\mathcal{A}}(n) = 2 + C_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \text{ si } n > 0$$

Théorème maître

Théorème maître

Si une complexité $C(n)$ vérifie la relation de récurrence suivante :

$$C(n) = a \cdot C\left(\frac{n}{b}\right) + f(n), \quad \text{avec } a \geqslant 1 \text{ et } b > 1$$

Alors, en notant $\boldsymbol{c = \log_b(a)}$:

- Si $f(n) = \mathcal{O}(n^c)$, alors $C(n) = \Theta(n^c)$
- Si $f(n) = \Theta\left(n^c \log^k(n)\right)$, alors $C(n) = \Theta\left(n^c \log^{k+1}(n)\right)$
- Si $f(n) = \Omega(n^c)$, et si il existe $k < 1$ et N tel que $\forall n > N, af\left(\frac{n}{b}\right) < kf(n)$ alors $C(n) = \Theta(f(n))$

Exponentiation rapide

```python
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    else:
        y = exponentiation_rapide(x, n//2)
        z = y * y
        if n%2 == 1:
            z *= x
        return z
```

Analyse de la complexité :

$$C_{\mathcal{A}}(n) = C_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \qquad \Longrightarrow \qquad a = 1,\ b = 2,\ f(n) = 2$$

Exponentiation rapide

```
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    else:
        y = exponentiation_rapide(x, n//2)
        z = y * y
        if n%2 == 1:
            z *= x
        return z
```

Analyse de la complexité :

$$C_{\mathcal{A}}(n) = C_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \qquad \Longrightarrow \qquad a = 1,\ b = 2,\ f(n) = 2$$

Exposant critique : $c = \log_b(a) = \log_2(1) = 0 \quad \Longrightarrow \quad f(n) = \Theta(1) = \Theta\left(n^c \log^0(n)\right)$

Exponentiation rapide

```python
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    else:
        y = exponentiation_rapide(x, n//2)
        z = y * y
        if n%2 == 1:
            z *= x
        return z
```

Analyse de la complexité :

$$C_{\mathcal{A}}(n) = C_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \qquad \Longrightarrow \qquad a = 1,\ b = 2,\ f(n) = 2$$

Exposant critique : $c = \log_b(a) = \log_2(1) = 0 \quad \Longrightarrow \quad f(n) = \Theta(1) = \Theta\left(n^c \log^0(n)\right)$

$$C_{\mathcal{A}}(n) = \Theta\left(n^c \log^1(n)\right) = \Theta\left(\log(n)\right)$$