

Apprendre la Cyber par les Challenges

Du Lycee a l'Universite

LearnByAoC

Professeur Kali

Base sur les challenges Advent of Code

7 décembre 2025

Table des matières

Preface	vii
Carte des Concepts	ix
I Fondations	1
1 Parsing : Lire et Transformer les Donnees	3
1.1 Qu'est-ce que le parsing ?	3
1.2 Lire un fichier	3
1.2.1 La methode de base	3
1.2.2 Nettoyer les donnees	4
1.3 Decouper une chaîne	4
1.3.1 <code>split()</code> : le couteau suisse	4
1.4 Convertir les types	4
1.5 Patterns de parsing courants	5
1.5.1 Pattern 1 : Liste de nombres	5
1.5.2 Pattern 2 : Valeurs séparées par virgules	5
1.5.3 Pattern 3 : Grille 2D	5
1.5.4 Pattern 4 : Sections séparées par ligne vide	5
1.6 Challenges d'application	6
1.7 Application en cybersécurité	6
1.8 Résumé	7
2 Arithmetique : Compter et Calculer	9
2.1 Les opérations de base	9
2.2 Le modulo : ton nouvel ami	9
2.3 GCD et LCM : cycles et périodicité	10
2.4 Challenges d'application	10
2.5 Résumé	10
3 Recherche : Trouver ce qu'on Cherche	11
3.1 Recherche linéaire	11
3.2 Recherche dans un set : O(1)	11
3.3 Recherche binaire	11
3.4 Challenges d'application	12
3.5 Résumé	12
4 Grilles 2D : Naviguer dans l'Espace	13
4.1 Représenter une grille	13
4.2 Les 4 directions	13
4.3 Distance de Manhattan	13

4.4 Challenges d'application	14
II Structures de Données	15
5 Ensembles (Sets) : L'Unicité	17
5.1 Qu'est-ce qu'un set ?	17
5.2 Opérations ensemblistes	17
5.3 Pattern : Positions visitées	17
5.4 Challenges d'application	18
6 Dictionnaires : Associer Clés et Valeurs	19
6.1 Qu'est-ce qu'un dict ?	19
6.2 defaultdict : le dict intelligent	19
6.3 Counter : compter automatiquement	20
6.4 Challenges d'application	20
7 Piles et Files : L'Ordre de Traitement	21
7.1 La Pile (Stack) : LIFO	21
7.2 La File (Queue) : FIFO	21
7.3 Applications	21
7.4 Challenges d'application	22
8 Arbres : Structures Hiérarchiques	23
8.1 Qu'est-ce qu'un arbre ?	23
8.2 Parcours d'arbre	23
8.3 Challenges d'application	24
III Algorithmes	25
9 Recursion : Se Rappeler Soi-même	27
9.1 Qu'est-ce que la récursion ?	27
9.2 Memoization : éviter les recalculs	27
9.3 Challenges d'application	28
10 Graphes : BFS, DFS et Dijkstra	29
10.1 Qu'est-ce qu'un graphe ?	29
10.2 BFS : Largeur d'abord	29
10.3 DFS : Profondeur d'abord	30
10.4 Dijkstra : Chemins ponderés	30
10.5 Challenges d'application	30
11 Programmation Dynamique : Optimiser par Sous-problèmes	33
11.1 Qu'est-ce que la DP ?	33
11.2 Les deux approches	33
11.3 Challenges d'application	34
12 Simulation : Modéliser le Monde	35
12.1 Automates cellulaires	35
12.2 Détection de cycles	35
12.3 Challenges d'application	36

IV Applications Cyber	37
13 Cryptographie : Cacher et Reveler	39
13.1 XOR : le chiffrement reversible	39
13.2 Hachage : l'empreinte unique	39
13.3 Challenges d'application	39
14 Parsing Avance : Regex et Grammaires	41
14.1 Expressions regulieres (Regex)	41
14.2 Patterns courants	41
14.3 Challenges d'application	42
15 Optimisation : Faire Plus Vite	43
15.1 Identifier les goulots	43
15.2 Complexite algorithmique	43
15.3 Techniques courantes	43
15.4 Challenges d'application	44
V Annexes	45
A Reference Python	47
A.1 Collections	47
A.2 Comprehensions	47
A.3 Fonctions utiles	47
B Complexite Algorithmique	49
B.1 Notation Big-O	49
B.2 Complexites courantes	49
B.3 Regles de calcul	49
B.4 Complexite spatiale	49
C Index des Challenges	51
C.1 Par Concept Principal	51
C.1.1 Parsing et I/O	51
C.1.2 Grilles et Navigation	51
C.1.3 BFS / Pathfinding	51
C.1.4 Recursion et DP	51
C.1.5 Mathematiques	51
C.2 Par Difficulte	52
C.2.1 Niveau Debutant (★)	52
C.2.2 Niveau Intermediaire (★★)	52
C.2.3 Niveau Avance (★★★)	52
C.2.4 Niveau Expert (★★★★)	52
C.3 Challenges Emblematiques	52
Index des Challenges	53

Preface

Ce livre est né d'une conviction : **on apprend mieux en résolvant des problèmes qu'en lisant des cours.**

Les challenges d'Advent of Code, créés par Eric Wastl depuis 2015, offrent une progression idéale pour apprendre la programmation et la pensée algorithmique. Plus de 250 problèmes, du plus simple au plus complexe, couvrant pratiquement tous les concepts fondamentaux de l'informatique.

Comment utiliser ce livre ?

- Chaque chapitre couvre un **concept** (pas une année d'AoC)
- Les concepts sont ordonnés par **prerequisites**
- Chaque concept est illustré par plusieurs **challenges**
- Les explications vont du plus simple (10 ans) au plus technique

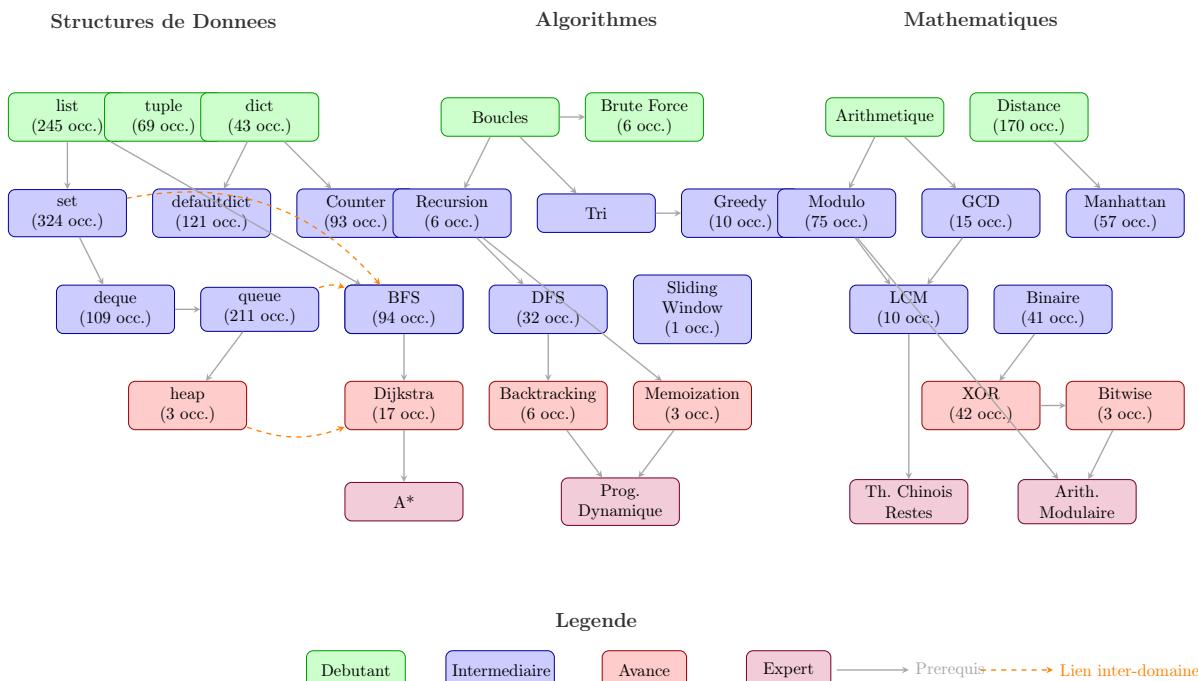
🛡 En Cybersecurité

Chaque concept est relié à une application concrète en cybersecurité. Tu ne fais pas que résoudre des puzzles - tu construis des compétences réelles.

“On n'apprend pas à nager en lisant un livre. On saute dans l'eau, on boit la tasse, puis on comprend.”

Carte des Concepts

Le graphe ci-dessous montre les **dependances entre concepts**. Une flèche de A vers B signifie que tu dois maitriser A avant d'apprendre B.



Legende des couleurs :

- **Vert** : Niveau Débutant (Lycee)
- **Bleu** : Niveau Intermediaire (Bac / L1)
- **Rouge** : Niveau Avance (L2 / L3)
- **Violet** : Niveau Expert (Master)

Les nombres entre parenthèses indiquent combien de fois ce concept apparaît dans les 257 challenges Advent of Code (2015-2025).

Première partie

Fondations

Chapitre 1

Parsing : Lire et Transformer les Donnees

L'Histoire

Imagine que tu receois une lettre codee. Avant de pouvoir comprendre le message, tu dois d'abord **decoder** les symboles. C'est exactement ce qu'on fait en programmation : transformer du texte brut en donnees utilisables.

1.1 Qu'est-ce que le parsing ?

Definition 1.1 (Parsing). Le **parsing** (ou analyse syntaxique) est l'action de lire des donnees brutes (texte, fichier) et de les transformer en une structure exploitable par un programme.

Exemple 1.1. Transformer la chaine "123,456,789" en liste [123, 456, 789].

1.2 Lire un fichier

1.2.1 La methode de base

```
1 # Methode 1 : Lire tout le fichier
2 with open("input.txt") as f:
3     contenu = f.read()
4
5 # Methode 2 : Lire ligne par ligne
6 with open("input.txt") as f:
7     lignes = f.readlines()
8
9 # Methode 3 : Iteration directe (recommande)
10 with open("input.txt") as f:
11     for ligne in f:
12         print(ligne.strip())
```

Listing 1.1 – Lire un fichier ligne par ligne

Concept Cle

Le mot-cle **with** garantit que le fichier sera ferme automatiquement, meme en cas d'erreur. C'est une bonne pratique a toujours utiliser.

1.2.2 Nettoyer les donnees

```

1 ligne = "Hello World \n"
2
3 ligne.strip()      # "Hello World" (supprime espaces + \n)
4 ligne.rstrip()     # "Hello World" (seulement a droite)
5 ligne.lstrip()     # "Hello World \n" (seulement a gauche)

```

Listing 1.2 – Methodes de nettoyage

1.3 Decouper une chaine

1.3.1 split() : le couteau suisse

```

1 # Decouper par espaces (defaut)
2 "a b c".split()      # ['a', 'b', 'c']
3
4 # Decouper par un caractere specifique
5 "a,b,c".split(",")   # ['a', 'b', 'c']
6
7 # Decouper par une chaine
8 "a->b->c".split("->") # ['a', 'b', 'c']
9
10 # Limiter le nombre de decoupages
11 "a,b,c,d".split(",", 2) # ['a', 'b', 'c,d']

```

Listing 1.3 – Utilisation de split()

⚠ Piege a Eviter

split() sans argument decoupe par **tous les espaces blancs** (espaces, tabs, newlines) et supprime les elements vides. split(" ") ne decoupe que par l'espace simple.

```

1 "a b".split()      # ['a', 'b'] (2 espaces ignores)
2 "a b".split(" ")   # ['a', '', 'b'] (element vide!)

```

1.4 Convertir les types

```

1 # String vers entier
2 int("42")           # 42
3 int(" 42 ")         # 42 (strip automatique)
4 int("42abc")        # ERREUR !
5
6 # String vers flottant
7 float("3.14")       # 3.14
8
9 # Entier vers string
10 str(42)            # "42"
11
12 # Conversion en masse avec map()
13 nombres = list(map(int, ["1", "2", "3"])) # [1, 2, 3]
14
15 # Ou avec une comprehension (plus lisible)
16 nombres = [int(x) for x in ["1", "2", "3"]] # [1, 2, 3]

```

Listing 1.4 – Conversions de types

1.5 Patterns de parsing courants

1.5.1 Pattern 1 : Liste de nombres

Entrée :

```
123
456
789
```

```
1 with open("input.txt") as f:
2     nombres = [int(ligne) for ligne in f]
3 # [123, 456, 789]
```

Listing 1.5 – Parser une liste de nombres

1.5.2 Pattern 2 : Valeurs séparées par virgules

Entrée : 1,2,3,4,5

```
1 with open("input.txt") as f:
2     nombres = [int(x) for x in f.read().strip().split(",")]
3 # [1, 2, 3, 4, 5]
```

Listing 1.6 – Parser des valeurs CSV

1.5.3 Pattern 3 : Grille 2D

Entrée :

```
@.@
.@@.
@.@


```

```
1 with open("input.txt") as f:
2     grille = [list(ligne.strip()) for ligne in f]
3 # [['@', '.', '@', '.'],
4 #  ['.', '@', '@', '.'],
5 #  ['@', '.', '@', '.']]
6
7 # Accès : grille[ligne][colonne]
8 grille[0][0] # @@
9 grille[1][1] # @@
```

Listing 1.7 – Parser une grille

1.5.4 Pattern 4 : Sections séparées par ligne vide

Entrée :

```
section1
data1
```

```
section2
data2
```

```

1 with open("input.txt") as f:
2     sections = f.read().strip().split("\n\n")
3 # ['section1\ndata1', 'section2\ndata2']
4
5 # Puis parser chaque section
6 for section in sections:
7     lignes = section.split("\n")
8     # ...

```

Listing 1.8 – Parser des sections

1.6 Challenges d'application

Exercice

AoC 2015 Day 1 - Not Quite Lisp

Parse une chaine de caracteres (et) et compte combien de fois chaque caractere apparait.

Difficulte : ★

Concepts : Iteration sur string, comptage

Exercice

AoC 2020 Day 4 - Passport Processing

Parse des “passeports” avec des champs cle:valeur separees par espaces ou newlines.

Difficulte : ★★

Concepts : Split multiple, dictionnaires

Exercice

AoC 2016 Day 4 - Security Through Obscurity

Parse des noms de salles au format nom-avec-tirets-123[checksum].

Difficulte : ★★★

Concepts : Regex, extraction de patterns

Google Colab

Notebook interactif - Parsing

<https://colab.research.google.com/github/Karlbblock/>

LearnByAoC/blob/master/notebooks/01_parsing.ipynb



1.7 Application en cybersecurite

En Cybersecurite

Le parsing est **fondamental** en cyber :

- **Analyse de logs** : Parser les logs Apache, auth.log, syslog
- **Parsing de packets** : Extraire les headers HTTP, DNS queries
- **Reverse engineering** : Parser des formats binaires
- **OSINT** : Extraire des donnees de pages web

Attention : Un parsing mal fait peut creer des vulnerabilites (injection, buffer overflow). Toujours valider les donnees !

1.8 Resume

Methode	Usage
open() / with	Lire un fichier
.strip()	Nettoyer les espaces
.split()	Dcouper une chaine
int() / float()	Convertir en nombre
map()	Appliquer une fonction
[x for x in ...]	Comprehension de liste

Chapitre 2

Arithmetique : Compter et Calculer

▀ L'Histoire

Tu te souviens quand tu as appris à compter sur tes doigts ? En programmation, on fait pareil - mais avec des millions de nombres, et beaucoup plus vite !

2.1 Les opérations de base

Definition 2.1 (Operateurs arithmetiques). Python utilise les operateurs classiques :

- + : Addition
- - : Soustraction
- * : Multiplication
- / : Division (resultat flottant)
- // : Division entière
- % : Modulo (reste de la division)
- ** : Puissance

```
1 17 / 5    # 3.4    (division flottante)
2 17 // 5   # 3     (division entière)
3 17 % 5    # 2     (reste : 17 = 3*5 + 2)
4 2 ** 10   # 1024  (2 puissance 10)
```

Listing 2.1 – Opérations arithmétiques

2.2 Le modulo : ton nouvel ami

💡 Concept Cle

Le **modulo (%)** donne le **reste** de la division. C'est l'opération la plus utile en programmation !

```
1 # Pair ou impair ?
2 n % 2 == 0  # True si n est pair
3
4 # Position dans un cycle (0 à n-1)
5 jour = total_jours % 7 # Jour de la semaine (0-6)
6
7 # Eviter les dépassemens de liste
```

```

8 index = i % len(liste) # Revient au debut si depasse
9
10 # Extraire le dernier chiffre
11 dernier = n % 10 # Chiffre des unites

```

Listing 2.2 – Usages courants du modulo

Exercice

AoC 2015 Day 1 - Not Quite Lisp

Utilise un compteur simple : (ajoute 1,) soustrait 1. Trouve l'etage final.

Concepts : Iteration, comptage, conditions

2.3 GCD et LCM : cycles et periodicite

Definition 2.2 (PGCD et PPCM). — **PGCD** (GCD) : Plus Grand Commun Diviseur
— **PPCM** (LCM) : Plus Petit Commun Multiple

```

1 import math
2
3 def lcm(a, b):
4     return a * b // math.gcd(a, b)
5
6 # Quand 3 cycles de periodes 4, 6, 9 se synchronisent ?
7 from functools import reduce
8 periodes = [4, 6, 9]
9 sync = reduce(lcm, periodes) # 36

```

Listing 2.3 – Calcul du LCM

En Cybersecurite

Le PGCD est utilise en **cryptographie** (RSA) et le PPCM pour analyser les **cycles** dans les malwares.

2.4 Challenges d'application

Exercice

AoC 2020 Day 13 - Shuttle Search

Trouve quand plusieurs bus avec des periodes differentes passent en meme temps.

Difficulte : ★★★★**Concepts** : LCM, modulo, arithmetique modulaire

2.5 Resume

Concept	Usage principal
% (modulo)	Cycles, parite, position circulaire
// (div. entiere)	Quotient sans decimales
math.gcd()	Plus grand diviseur commun
math.lcm()	Synchronisation de cycles

Chapitre 3

Recherche : Trouver ce qu'on Cherche

L'Histoire

Imagine que tu cherches un mot dans le dictionnaire. Tu ne lis pas toutes les pages depuis le debut! Tu ouvres au milieu, tu regardes si c'est avant ou apres, et tu recommences. C'est la **recherche binaire**.

3.1 Recherche lineaire

Definition 3.1 (Recherche lineaire). Parcourir tous les elements un par un jusqu'a trouver ce qu'on cherche. Complexite : $O(n)$.

3.2 Recherche dans un set : $O(1)$

Concept Cle

Un **set** utilise une **table de hachage**. La recherche est instantanee ($O(1)$).

```
1 # LENT : recherche dans une liste O(n)
2 if x in liste: # parcourt toute la liste
3     ...
4
5 # RAPIDE : recherche dans un set O(1)
6 ensemble = set(liste)
7 if x in ensemble: # acces direct
8     ...
```

Listing 3.1 – Comparaison list vs set

3.3 Recherche binaire

Definition 3.2 (Recherche binaire). Diviser l'espace de recherche en deux a chaque etape. Complexite : $O(\log n)$.

```
1 import bisect
2
3 liste = [1, 3, 5, 7, 9]
4 bisect.bisect_left(liste, 6) # 3 (position d'insertion)
```

Listing 3.2 – Module bisect

3.4 Challenges d'application

📝 Exercice

AoC 2020 Day 1 - Report Repair

Trouve deux nombres dont la somme est 2020. Optimise avec un set.

Concepts : Recherche dans set, complement

🛡 En Cybersecurite

- **Bruteforce intelligent** : Recherche binaire sur l'espace de clés
- **Analyse de logs** : Recherche rapide dans des GB de logs

3.5 Resume

Methode	Complexite	Quand
Liste + <code>in</code>	$O(n)$	Petite liste
Set + <code>in</code>	$O(1)$	Recherches multiples
Recherche binaire	$O(\log n)$	Données triées

Chapitre 4

Grilles 2D : Naviguer dans l'Espace

L'Histoire

Tu connais les échecs ? Une grille 8x8 où chaque pièce peut bouger dans certaines directions. En programmation, on utilise constamment des grilles !

4.1 Representer une grille

```
1 # Parser un fichier en grille
2 with open("input.txt") as f:
3     grille = [list(ligne.strip()) for ligne in f]
4
5 # Accès : grille[ligne][colonne]
6 grille[0][0] # coin haut-gauche
7
8 # Dimensions
9 hauteur = len(grille)
10 largeur = len(grille[0])
```

Listing 4.1 – Créer et parser une grille

⚠ Piege à Eviter

Attention : `grille[ligne][colonne]` = `grille[y][x]` !

4.2 Les 4 directions

```
1 DIRECTIONS = [
2     (-1, 0), # Haut
3     (1, 0), # Bas
4     (0, -1), # Gauche
5     (0, 1) # Droite
6 ]
```

Listing 4.2 – Directions cardinales

4.3 Distance de Manhattan

Definition 4.1 (Distance de Manhattan). Distance en blocs : $d = |x_1 - x_2| + |y_1 - y_2|$

```
1 def manhattan(p1, p2):  
2     return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])
```

Listing 4.3 – Calcul Manhattan

4.4 Challenges d'application

Exercice

AoC 2015 Day 3 - Perfectly Spherical Houses

Suis des directions et compte les maisons visitez.

Concepts : Directions, set de positions

En Cybersecurite

- **Analyse d'images** : Steganographie, detection
- **Cartographie reseau** : Visualisation de topologie

Deuxième partie

Structures de Donnees

Chapitre 5

Ensembles (Sets) : L'Unicite

L'Histoire

Imagine un sac magique où tu ne peux mettre qu'un seul exemplaire de chaque objet. Si tu essaies de mettre deux pommes identiques, le sac n'en garde qu'une. C'est un **ensemble** !

5.1 Qu'est-ce qu'un set ?

Definition 5.1 (Set). Un **set** est une collection **non ordonnee** d'éléments **uniques**. Recherche en $O(1)$.

```
1 # Creation
2 ensemble = {1, 2, 3}
3 ensemble = set([1, 2, 2, 3])  # {1, 2, 3}
4
5 # Ajout / Suppression
6 ensemble.add(4)
7 ensemble.remove(1)          # Erreur si absent
8 ensemble.discard(10)        # Pas d'erreur si absent
9
10 # Test d'appartenance O(1)
11 if x in ensemble:
12     ...
```

Listing 5.1 – Opérations de base

5.2 Opérations ensemblistes

```
1 a = {1, 2, 3}
2 b = {2, 3, 4}
3
4 a | b    # {1, 2, 3, 4} Union
5 a & b    # {2, 3} Intersection
6 a - b    # {1} Difference
7 a ^ b    # {1, 4} Difference symétrique
```

Listing 5.2 – Union, intersection, différence

5.3 Pattern : Positions visitees

```
1 visited = set()
2 position = (0, 0)
3
4 while True:
5     if position in visited:
6         print("Deja visite !")
7         break
8     visited.add(position)
9     position = move(position)
```

Listing 5.3 – Tracker les positions

5.4 Challenges d'application

📝 Exercice

AoC 2015 Day 3 - Perfectly Spherical Houses

Compte les maisons uniques visitees par le Pere Noel.

Concepts : Set de tuples, positions

🛡️ En Cybersecurite

- **Deduplication** : Supprimer les doublons dans les logs
- **Detection d'anomalies** : Nouvelles IPs, nouveaux users

Chapitre 6

Dictionnaires : Associer Cles et Valeurs

L'Histoire

Un dictionnaire, c'est comme ton carnet de contacts : tu cherches un nom (la cle) et tu trouves le numero (la valeur). Instantanément.

6.1 Qu'est-ce qu'un dict ?

Definition 6.1 (Dictionnaire). Un dict associe des **cles** a des **valeurs**. Acces en O(1).

```
1 # Creation
2 scores = {"Alice": 100, "Bob": 85}
3
4 # Acces
5 scores["Alice"]      # 100
6 scores.get("Eve", 0) # 0 (defaut si absent)
7
8 # Modification
9 scores["Alice"] = 110
10 scores["Eve"] = 95   # Nouvelle entree
11
12 # Iteration
13 for nom, score in scores.items():
14     print(f"{nom}: {score}")
```

Listing 6.1 – Opérations de base

6.2 defaultdict : le dict intelligent

```
1 from collections import defaultdict
2
3 # Comptage
4 compteur = defaultdict(int)
5 for mot in mots:
6     compteur[mot] += 1 # Pas besoin d'initialiser !
7
8 # Listes de valeurs
9 graphe = defaultdict(list)
10 graphe["A"].append("B") # Pas besoin de verifier si "A" existe
```

Listing 6.2 – defaultdict pour éviter les KeyError

6.3 Counter : compter automatiquement

```
1 from collections import Counter
2
3 texte = "abracadabra"
4 freq = Counter(texte)
5 # Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
6
7 freq.most_common(3)  # [('a', 5), ('b', 2), ('r', 2)]
```

Listing 6.3 – Counter pour les fréquences

6.4 Challenges d'application

📝 Exercice

AoC 2024 Day 1 - Historian Hysteria

Compte les occurrences et calcule un score de similarité.

Concepts : Counter, multiplication

🛡 En Cybersecurité

- **Analyse de fréquence** : Casser le chiffrement par substitution
- **Cache/Memoization** : Accélérer les calculs

Chapitre 7

Piles et Files : L'Ordre de Traitement

■ L'Histoire

Une pile d'assiettes : tu poses en haut, tu prends en haut (LIFO). Une file d'attente au cinema : le premier arrive est le premier servi (FIFO).

7.1 La Pile (Stack) : LIFO

Definition 7.1 (Pile). Last In, First Out. Le dernier element ajoute est le premier retire.

```
1 pile = []
2 pile.append(1)    # Push
3 pile.append(2)
4 pile.pop()        # 2 (Pop)
5 pile[-1]          # 1 (Peek sans retirer)
```

Listing 7.1 – Pile avec une liste

7.2 La File (Queue) : FIFO

Definition 7.2 (File). First In, First Out. Le premier element ajoute est le premier retire.

```
1 from collections import deque
2
3 file = deque()
4 file.append(1)      # Ajouter a droite
5 file.append(2)
6 file.popleft()     # 1 (Retirer a gauche)
```

Listing 7.2 – File avec deque

💡 Concept Cle

deque est $O(1)$ aux deux extremites. Une liste est $O(n)$ pour pop(0) !

7.3 Applications

- **Pile** : Parentheses équilibrées, DFS, undo/redo
- **File** : BFS, traitement dans l'ordre d'arrivée

7.4 Challenges d'application

Exercice

AoC 2021 Day 10 - Syntax Scoring

Verifie l'équilibrage des parenthèses avec une pile.

Concepts : Pile, matching de caractères

En Cybersecurite

- **Buffer overflow** : Comprendre la pile d'exécution
- **Message queues** : Communication inter-processus

Chapitre 8

Arbres : Structures Hierarchiques

L'Histoire

Un arbre genealogique : grand-parents en haut, parents au milieu, enfants en bas. Chaque personne a un parent (sauf la racine) et peut avoir plusieurs enfants.

8.1 Qu'est-ce qu'un arbre ?

Definition 8.1 (Arbre). Structure hierarchique avec une **racine**, des **noeuds** et des **feuilles** (noeuds sans enfants).

```
1 # Arbre comme dict parent -> enfants
2 arbre = {
3     "COM": ["B"],
4     "B": ["C", "G"],
5     "C": ["D"],
6     "G": ["H"],
7 }
8
9 # Ou enfant -> parent (pour remonter)
10 parents = {
11     "B": "COM",
12     "C": "B",
13     "D": "C",
14 }
```

Listing 8.1 – Representation avec dict

8.2 Parcours d'arbre

```
1 def parcours(noeud, arbre, profondeur=0):
2     print(" " * profondeur + noeud)
3     for enfant in arbre.get(noeud, []):
4         parcours(enfant, arbre, profondeur + 1)
```

Listing 8.2 – Parcours recursif

8.3 Challenges d'application

Exercice

AoC 2019 Day 6 - Universal Orbit Map

Compte les orbites directes et indirectes dans un arbre.

Concepts : Parcours d'arbre, comptage de profondeur

En Cybersecurite

- **Système de fichiers** : Arborescence de répertoires
- **DOM/XML** : Parsing de documents structures

Troisième partie

Algorithmes

Chapitre 9

Recursion : Se Rappeler Soi-même

L'Histoire

Pour comprendre la recursion, il faut d'abord comprendre la recursion. C'est une blague de programmeur ! Une fonction qui s'appelle elle-même pour résoudre un problème plus petit.

9.1 Qu'est-ce que la recursion ?

Definition 9.1 (Recursion). Une fonction qui s'appelle elle-même avec un cas de base pour s'arrêter.

```
1 def factorielle(n):
2     if n <= 1:          # Cas de base
3         return 1
4     return n * factorielle(n - 1)  # Appel recursif
5
6 # 5! = 5 * 4 * 3 * 2 * 1 = 120
```

Listing 9.1 – Exemple : factorielle

9.2 Memoization : éviter les recalculs

```
1 from functools import lru_cache
2
3 @lru_cache(maxsize=None)
4 def fib(n):
5     if n < 2:
6         return n
7     return fib(n-1) + fib(n-2)
8
9 # Sans cache : O(2^n) - EXPLOSIF
10 # Avec cache : O(n) - linéaire
```

Listing 9.2 – Fibonacci avec memoization

Concept Cle

La **memoization** stocke les résultats déjà calculés. Essentiel pour la programmation dynamique !

9.3 Challenges d'application

Exercice

AoC 2015 Day 7 - Some Assembly Required

Evalue des circuits logiques avec des dependances recursives.

Concepts : Recursion, memoization, graphe de dependances

En Cybersecurite

- **Parsing recursif** : Grammaires, expressions
- **Exploration de systemes** : Fichiers, processus

Chapitre 10

Graphes : BFS, DFS et Dijkstra

L'Histoire

Un réseau social est un graphe : les personnes sont des noeuds, les amitiés sont des arêtes.
Comment trouver le chemin le plus court entre deux personnes ?

10.1 Qu'est-ce qu'un graphe ?

Definition 10.1 (Graphe). Ensemble de **noeuds** (sommets) reliés par des **arêtes** (liens).

```
1 from collections import defaultdict
2
3 graphe = defaultdict(list)
4 graphe["A"].append("B")
5 graphe["A"].append("C")
6 graphe["B"].append("D")
```

Listing 10.1 – Représentation par liste d'adjacence

10.2 BFS : Largeur d'abord

```
1 from collections import deque
2
3 def bfs(start, goal, graphe):
4     queue = deque([(start, 0)])
5     visited = {start}
6
7     while queue:
8         node, dist = queue.popleft()
9
10        if node == goal:
11            return dist
12
13        for voisin in graphe[node]:
14            if voisin not in visited:
15                visited.add(voisin)
16                queue.append((voisin, dist + 1))
17
18    return -1
```

Listing 10.2 – BFS - Plus court chemin non pondéré

10.3 DFS : Profondeur d'abord

```

1 def dfs(node, visited, graphe):
2     if node in visited:
3         return
4     visited.add(node)
5
6     for voisin in graphe[node]:
7         dfs(voisin, visited, graphe)

```

Listing 10.3 – DFS recursif

10.4 Dijkstra : Chemins ponderés

```

1 import heapq
2
3 def dijkstra(start, goal, graphe):
4     pq = [(0, start)]
5     distances = {start: 0}
6
7     while pq:
8         dist, node = heapq.heappop(pq)
9
10        if node == goal:
11            return dist
12
13        if dist > distances.get(node, float('inf')):
14            continue
15
16        for voisin, poids in graphe[node]:
17            new_dist = dist + poids
18            if new_dist < distances.get(voisin, float('inf')):
19                distances[voisin] = new_dist
20                heapq.heappush(pq, (new_dist, voisin))
21
22    return -1

```

Listing 10.4 – Dijkstra avec heapq

10.5 Challenges d'application

Exercice

AoC 2016 Day 13 - A Maze of Twisty Little Cubicles

BFS dans un labyrinthe généré par une formule.

Difficulté : ★★

Concepts : BFS, génération procédurale

Exercice

AoC 2021 Day 15 - Chiton

Trouve le chemin de risque minimal dans une grille.

Difficulté : ★★★

Concepts : Dijkstra, grille comme graphe

 En Cybersecurite

- **Scan reseau** : Cartographie de topologie
- **Analyse de malware** : Graphe d'appels de fonctions
- **Pathfinding** : Routage, navigation

Chapitre 11

Programmation Dynamique : Optimiser par Sous-problemes

■ L'Histoire

Tu veux monter un escalier. A chaque marche, tu peux faire 1 ou 2 pas. Combien de façons différentes d'arriver en haut ? Au lieu de tout recalculer, on réutilise les résultats précédents.

11.1 Qu'est-ce que la DP ?

Definition 11.1 (Programmation Dynamique). Technique qui résout un problème en le décomposant en **sous-problèmes** et en stockant leurs solutions pour éviter les recalculs.

```
1 def escalier(n):
2     # dp[i] = nombre de façons d'atteindre marche i
3     dp = [0] * (n + 1)
4     dp[0] = 1  # 1 façon de rester en bas
5     dp[1] = 1  # 1 façon d'atteindre marche 1
6
7     for i in range(2, n + 1):
8         dp[i] = dp[i-1] + dp[i-2]
9
10    return dp[n]
```

Listing 11.1 – Exemple : Escalier

11.2 Les deux approches

- **Top-down** : Recursion + memoization
- **Bottom-up** : Iteration, remplir un tableau

11.3 Challenges d'application

Exercice

AoC 2020 Day 10 - Adapter Array

Compte les arrangements possibles d'adaptateurs.

Concepts : DP, comptage de chemins

En Cybersecurite

- **Alignement de séquences** : Comparaison ADN/malware
- **Optimisation** : Allocation de ressources

Chapitre 12

Simulation : Modeliser le Monde



L'Histoire

Le Jeu de la Vie de Conway : des cellules naissent et meurent selon des règles simples. Pourtant, des structures complexes émergent. Bienvenue dans la simulation !

12.1 Automates cellulaires

```
1 def step(grille):
2     nouvelle = set()
3
4     # Compter les voisins
5     for cell in grille:
6         voisins = count_neighbors(cell, grille)
7         if voisins in [2, 3]:
8             nouvelle.add(cell)
9
10    # Naissance
11    for cell in candidates(grille):
12        if count_neighbors(cell, grille) == 3:
13            nouvelle.add(cell)
14
15    return nouvelle
```

Listing 12.1 – Game of Life simplifié

12.2 Détection de cycles

```
1 def find_cycle(initial_state, step_func):
2     seen = {initial_state: 0}
3     state = initial_state
4     step = 0
5
6     while True:
7         state = step_func(state)
8         step += 1
9
10        if state in seen:
11            cycle_start = seen[state]
12            cycle_length = step - cycle_start
13            return cycle_start, cycle_length
14
15        seen[state] = step
```

Listing 12.2 – Trouver un cycle

12.3 Challenges d'application

Exercice

AoC 2020 Day 17 - Conway Cubes

Game of Life en 3D et 4D.

Concepts : Simulation, dimensions multiples

En Cybersecurite

- **Analyse de malware** : Execution symbolique
- **Fuzzing** : Simulation d'entrees

Quatrième partie

Applications Cyber

Chapitre 13

Cryptographie : Cacher et Reveler

■ L'Histoire

Les espions ont toujours utilise des codes secrets. Aujourd'hui, c'est pareil mais avec des mathematiques. Comment cacher un message pour que seul le destinataire puisse le lire ?

13.1 XOR : le chiffrement reversible

Definition 13.1 (XOR). Operation binaire : $1 \text{ XOR } 1 = 0$, $1 \text{ XOR } 0 = 1$, $0 \text{ XOR } 0 = 0$. Propriete magique : $A \text{ XOR } B \text{ XOR } B = A$

```
1 def xor_encrypt(message, key):
2     return bytes([m ^ k for m, k in zip(message, cycle(key))])
3
4 # Dechiffrement = meme operation !
5 def xor_decrypt(ciphertext, key):
6     return xor_encrypt(ciphertext, key)
```

Listing 13.1 – Chiffrement XOR

13.2 Hachage : l'empreinte unique

```
1 import hashlib
2
3 message = b"secret"
4 hash_md5 = hashlib.md5(message).hexdigest()
5 # '5ebe2294ecd0e0f08eab7690d2a6ee69'
```

Listing 13.2 – MD5 en Python

13.3 Challenges d'application

■ Exercice

AoC 2015 Day 4 - The Ideal Stocking Stuffer

Trouve un nombre qui produit un hash MD5 commençant par des zeros.

Concepts : Hachage, bruteforce

 Exercice**AoC 2016 Day 5 - How About a Nice Game of Chess ?**

Déroule un mot de passe à partir de hash MD5 successifs.

Concepts : Hachage itératif

 En Cybersecurite

- **Cracking de mots de passe** : Rainbow tables, hashcat
- **Intégrité** : Vérification de fichiers
- **Blockchain** : Proof of work

Chapitre 14

Parsing Avance : Regex et Grammaires

L’Histoire

Parfois, les données sont tellement complexes qu’un simple `split()` ne suffit plus. Il faut des outils plus puissants : les expressions régulières.

14.1 Expressions régulières (Regex)

```
1 import re
2
3 texte = "Il y a 42 pommes et 17 oranges"
4
5 # Trouver tous les nombres
6 re.findall(r'\d+', texte) # ['42', '17']
7
8 # Extraire avec groupes
9 match = re.search(r'(\d+) pommes', texte)
10 match.group(1) # '42'
11
12 # Remplacer
13 re.sub(r'\d+', 'X', texte) # 'Il y a X pommes et X oranges'
```

Listing 14.1 – Regex de base

14.2 Patterns courants

Pattern	Signification
\d	Chiffre
\w	Lettre ou chiffre
\s	Espace
.	N’importe quel caractère
+	1 ou plus
*	0 ou plus
?	0 ou 1
()	Groupe de capture

14.3 Challenges d'application

Exercice

AoC 2015 Day 5 - Doesn't He Have Intern-Elves For This ?

Valide des strings avec des règles complexes.

Concepts : Regex, validation

En Cybersecurite

- **Log parsing** : Extraction d'IPs, timestamps
- **Input validation** : Sécuriser les formulaires
- **Grep/sed/awk** : Outils Unix essentiels

Chapitre 15

Optimisation : Faire Plus Vite

L'Histoire

Ton programme fonctionne, mais il met 10 minutes. Comment le faire tourner en 10 secondes ? L'optimisation, c'est l'art de faire plus avec moins.

15.1 Identifier les goulots

```
1 import time
2
3 start = time.time()
4 # Code à mesurer
5 result = slow_function()
6 print(f"Temps: {time.time() - start:.2f}s")
```

Listing 15.1 – Profiling simple

15.2 Complexité algorithmique

Complexité	Exemple	1M éléments
$O(1)$	Accès dict	Instantané
$O(\log n)$	Recherche binaire	20 opérations
$O(n)$	Parcours liste	1M opérations
$O(n \log n)$	Tri	20M opérations
$O(n^2)$	Double boucle	1T opérations
$O(2^n)$	Bruteforce	Impossible

15.3 Techniques courantes

- Utiliser `set` au lieu de `list` pour les recherches
- Memoization pour éviter les recalculs
- Éviter les copies inutiles de listes
- Utiliser des générateurs au lieu de listes

15.4 Challenges d'application

Exercice

AoC 2025 Day 7 - Laboratories

Optimise une simulation exponentielle avec `defaultdict(int)`.

Concepts : Comptage au lieu d'enumeration

En Cybersecurite

- **Timing attacks** : Mesurer les differences de temps
- **DoS** : Exploiter la complexite algorithmique

Cinquième partie

Annexes

Annexe A

Reference Python

A.1 Collections

```
1 from collections import deque, Counter, defaultdict
2 from itertools import permutations, combinations, product
3 from functools import lru_cache, reduce
4 import heapq
5 import math
6 import re
```

Listing A.1 – Imports essentiels

A.2 Comprehensions

```
1 # Liste
2 [x*2 for x in range(10) if x % 2 == 0]
3
4 # Dict
5 {k: v for k, v in items}
6
7 # Set
8 {x % 3 for x in range(10)}
9
10 # Generator
11 (x*2 for x in range(10))
```

Listing A.2 – Syntaxe des comprehensions

A.3 Fonctions utiles

Fonction	Usage
enumerate()	Indices + valeurs
zip()	Parcourir plusieurs listes
map()	Appliquer une fonction
filter()	Filtrer des éléments
sorted()	Trier (renvoie nouvelle liste)
reversed()	Inverser
any() / all()	Tests logiques

Annexe B

Complexite Algorithmique

B.1 Notation Big-O

Definition B.1 (Big-O). La complexite decrit comment le temps d'execution evolue avec la taille de l'entree.

B.2 Complexites courantes

Complexite	Nom	Exemples
$O(1)$	Constante	Acces dict/set, operations mathematiques
$O(\log n)$	Logarithmique	Recherche binaire, operations sur heap
$O(n)$	Lineaire	Parcours de liste, recherche lineaire
$O(n \log n)$	Linearithmique	Tri (merge sort, timsort)
$O(n^2)$	Quadratique	Double boucle imbriquee
$O(n^3)$	Cubique	Triple boucle, multiplication de matrices
$O(2^n)$	Exponentielle	Bruteforce, backtracking naif
$O(n!)$	Factorielle	Permutations completes

B.3 Regles de calcul

1. On garde le terme dominant : $O(n^2 + n) = O(n^2)$
2. On ignore les constantes : $O(3n) = O(n)$
3. Boucles imbriquées : on multiplie
4. Boucles sequentielles : on additionne

B.4 Complexite spatiale

Definition B.2 (Complexite spatiale). Quantite de memoire utilisee en fonction de la taille de l'entree.

- Liste de n elements : $O(n)$
- Grille $n \times n$: $O(n^2)$
- Set de positions visitees : $O(\text{nombre de positions uniques})$

Annexe C

Index des Challenges

C.1 Par Concept Principal

C.1.1 Parsing et I/O

- 2015 Day 1-5 : Parsing basique
- 2020 Day 4 : Parsing complexe (passeports)
- 2016 Day 4 : Regex

C.1.2 Grilles et Navigation

- 2015 Day 3 : Set de positions
- 2015 Day 6 : Grille booleenne
- 2020 Day 11 : Game of Life
- 2016 Day 1 : Distance Manhattan

C.1.3 BFS / Pathfinding

- 2016 Day 13 : BFS dans labyrinthe
- 2022 Day 12 : BFS avec contraintes
- 2021 Day 15 : Dijkstra
- 2018 Day 22 : A* avance

C.1.4 Recursion et DP

- 2015 Day 7 : Recursion avec memoization
- 2020 Day 10 : DP comptage
- 2015 Day 17 : Backtracking

C.1.5 Mathematiques

- 2020 Day 13 : Theoreme chinois des restes
- 2019 Day 12 : LCM pour cycles
- 2015 Day 4 : Hachage MD5

C.2 Par Difficulte

C.2.1 Niveau Debutant (★)

2015 Days 1-5, 2020 Day 1, 2024 Day 1

C.2.2 Niveau Intermediaire (★★)

2016 Day 13, 2015 Day 6-10, 2020 Days 3-8

C.2.3 Niveau Avance (★★★)

2021 Day 15, 2015 Day 7, 2020 Days 10-15

C.2.4 Niveau Expert (★★★★★)

2020 Day 13 (CRT), Days 20-25 de toutes les années

C.3 Challenges Emblematiques

Challenge	Pourquoi c'est un classique
2015 Day 7	Introduction parfaite à la récursion avec memoisation
2016 Day 13	BFS pur dans un espace générique
2021 Day 15	Méilleur challenge pour apprendre Dijkstra
2020 Day 13	Le théorème chinois des restes en pratique
2020 Day 17	Game of Life en N dimensions

Index des Challenges