Algorithmique et Structures de Données

PSI/PSI⋆

I Algorithmique—Rappels et Culture

1.1 Algorithmes

❦ Définition.

Un *algorithme* est une séquence d'instructions, destinées à être *exécutées* par une machine de calcul, en vue de *résoudre un problème*.

⚛ Exemples.

Algorithme E

entrées : a et b deux entiers
sorties/effets : $p = \text{PGCD}(a, b)$

```
1. tant que a ≠ b faire
2.     si a ≥ b alors
3.         a ← a − b
4.     sinon
5.         b ← b − a
6.     fin si
7. fin tant que
8. renvoyer a
```

Algorithme P

entrées : n un entier
sorties/effets : $p \iff n$ est premier

```
1. p ← VRAI
2. d ← 2
3. tant que d² ≤ n faire
4.     si n mod d = 0 alors
5.         p ← FAUX
6.     fin si
7.     d ← d + 1
8. fin tant que
9. renvoyer p
```

❧ ❧ ❧

Un algorithme prend des *entrées* et renvoie une *sortie*.

❦ Définition.

La *spécification* regroupe les informations suivantes :

- ▷ nom de l'algorithme
- ▷ *entrées* : **type**, **pré-condition** — éventuellement vide
- ▷ *sortie* : **type**, **post-condition** — éventuellement `None`
- ▷ et éventuellement : *complexité*, *dépendances*, *description*, …

La *signature* est un sous-ensemble de ces informations, regroupant uniquement le nom de l'algorithme, le type de la sortie et le type de chaque entrée.

⚛ Exemples.

Spécification d'un algorithme.

Algorithme `racine_entiere`

ENTRÉES : n un entier, $n \geq 0$
SORTIES/EFFETS : r un entier, $r = \min\{m \in \mathbb{N} \mid m > \sqrt{n}\}$
/* *Complexité :* $O(\sqrt{n})$ */

Signature en *Python*.

```
def racine_entiere(n : int) -> int
```

🐭🐭🐭

❦ DÉFINITION.

Une *instruction* d'un programme dicte à la machine ce qu'elle doit faire lors d'une étape de l'exécution.

Une *expression* est une instruction qui renvoie une valeur.

Un *effet de bord* est une modification de l'état du système, hors de l'environnement local de la fonction qui réalise cette modification.

⚛ EXEMPLES.

Expressions *Python* :

- `a + 3`
- `Euclide(12,21)`
- `L.pop()`
- `a < b`

Instructions avec effet de bord :

- `print("Nadine")`
- `a = a + 3`
- `L.append(12)`
- `L.pop()`
- `tracer_courbe(f,0,100)`
- `fichier.write("Nadine")`

🐭🐭🐭

Une fonction renvoie toujours une valeur, éventuellement `None`. Un appel de fonction est donc toujours une expression.

1.2 Paradigmes et méthodes de programmation

Hors Programme

La *programmation* consiste à *implémenter* un algorithme, *i.e.* à l'écrire dans un langage compréhensible par la machine. Il existe différents paradigmes auxquels sont associés différentes méthodes de programmation.

Les deux paradigmes utilisés dans ce cours sont :

- ▷ Programmation *Impérative*
  - — Programmation Structurée vs Programmation Spaghetti
  - — Programmation Procédurale (fonctions sans retour, avec effet de bords)
  - — Élément de base : *les boucles*.
  - — Méthode : **itérative**
- ▷ Programmation *Fonctionnelle*
  - — Élément de base : *les fonctions*.
  - — Méthode : **récursive**

Il existe aussi d'autre paradigmes :

- — Programmation Orientée Objet (POO)
- — Programmation Logique
- — Paradigmes Multiples
- — Programmation Orientée Processus, Programmation Concurrente, …

⚛ Exemples.

Méthode de Héron

$$\forall n \in \mathbb{N} \quad x_{n+1}^a = \frac{x_n^a + \frac{a}{x_n^a}}{2}, \; x_0^a = b$$

Algorithme `heron_imperatif`

entrées : a, b et n entiers, $a > 0$, $b > 0$, $n \geq 0$
sorties/effets : un entier égal à x_n^a

```
1. r ← b
2. pour i ← 0 à n faire
3.     r ← (r + a/r) / 2
4. fin pour
5. renvoyer r
```

Algorithme `heron_recursif`

entrées : a, b et n entiers, $a > 0$, $b > 0$, $n \geq 0$
sorties/effets : un entier égal à x_n^a

```
1. si n = 0 alors
2.     renvoyer b
3. fin si
4. r ← heron(a, b, n − 1)
5. renvoyer (r + a/r) / 2
```

1.3 Analyse d'algorithme

1.3.1 Terminaison

❦ Définition.

> On dit qu'un algorithme termine s'il s'arrête sur toutes ses entrées.

Cas itératif : Pour prouver la terminaison d'un algorithme, on doit montrer que toutes ses boucles terminent. Pour montrer qu'une boucle termine, il suffit d'exhiber une quantité entière qui décroît après chaque passage de la boucle. On définit pour se faire un *variant de boucle* ou *fonction de terminaison*. Un variant de boucle est une fonction dont les arguments sont les variables de l'algorithme et à valeur dans $\mathbb{N}$, qui décroît strictement à chaque itération de la boucle.

⚛ Exemples.

Algorithme Recherche_Dichotomique

entrées : T un tableau d'entiers croissants, x un entier
sorties/effets : $-1 \leq i < |\mathrm{T}|$ | si $i = -1, x \notin \mathrm{T}$ sinon $\mathrm{T}[i] = x$

```
1 . d ← 0, f ← |T|
2 . tant que d < f faire
3 .     m ← (d+f)/2
4 .     si x = T[m] alors
5 .         renvoyer m
6 .     sinon si x < T[m] alors
7 .         f ← m
8 .     sinon
9 .         d ← m + 1
10 .    fin si
11 . fin tant que
12 . renvoyer −1
```

Le variant $f - d$ diminue strictement jusqu'à 0, condition d'arrêt de la boucle.

🐁🐁🐁

Cas récursif : dans les programmes récursifs, la terminaison se prouve généralement par récurrence.

⚛ Exemples.

$$\mathrm{P}(n) : \forall \mathrm{T}, |\mathrm{T}| \leq n \implies \texttt{Recherche\_Dichotomique}(\mathrm{T}, x) \text{ termine.}$$

Algorithme Recherche_Dichotomique

entrées : T un tableau d'entiers croissants, x un entier
sorties/effets : $-1 \leq i < |\mathrm{T}|$ | si $i = -1, x \notin \mathrm{T}$ sinon $\mathrm{T}[i] = x$

```
1 . si |T| = 0 alors
2 .     renvoyer −1
3 . fin si
4 . m ← |T|/2
5 . si x = T[m] alors
6 .     renvoyer m
7 . sinon si x < T[m] alors
8 .     renvoyer Recherche_Dichotomique(T[0 : m],x)
9 . sinon
10 .    i ← Recherche_Dichotomique(T[m + 1 : |T|],x)
11 .    si i ≠ −1 alors
12 .        i ← i + m + 1
13 .    fin si
14 .    renvoyer i
15 . fin si
```

🐁🐁🐁

1.3.2 Correction

❦ Définition.

> Prouver un algorithme consiste à établir la preuve de la post-condition de l'algorithme en supposant la pré-condition comme étant vrai.

Chaque instruction fait évoluer les données associées aux variables de l'algorithme, la preuve consiste à établir que cette évolution, nous même à un état de la mémoire où la post-condition est vérifiée.

— Une correction est dite *totale* si l'algorithme termine et est correct.
— Une correction est dite *partielle* si l'algorithme est correct *lorsqu'il* termine.

Cas itératif : dans les programmes itératifs, la preuve fait intervenir un *invariant de boucle*. Un invariant de boucle est une propriété logique portant sur l'ensemble des variables de l'algorithme et qui reste vrai avant, après et à chaque itération de la boucle.

⚛ Exemples.

Algorithme Recherche_Dichotomique

entrées : T un tableau d'entiers croissants, x un entier
sorties/effets : $-1 \leq i < |\mathrm{T}| \mid\mid$ si $i = -1, x \notin \mathrm{T}$ sinon $\mathrm{T}[i] = x$

```
1 . d ← 0, f ← |T|
2 . /* Invariant : */
3 . /* 0 ≤ d ≤ f ≤ |T|, ∀0 ≤ i < d, T[i] < x , ∀f ≤ i < |T|, T[i] > x */
4 . tant que d < f faire
5 .     m ← (d+f)/2
6 .     si x = T[m] alors
7 .         renvoyer m
8 .     sinon si x < T[m] alors
9 .         f ← m
10 .    sinon
11 .        d ← m + 1
12 .    fin si
13 . fin tant que
14 . renvoyer −1
```

❧ ❧ ❧

Cas récursif : dans les programmes récursifs, la preuve se fait généralement par récurrence.

⚛ Exemples.

Algorithme Recherche_Dichotomique

entrées : T un tableau d'entiers croissants, x un entier
sorties/effets : $-1 \leq i < |\mathrm{T}|$ | si $i = -1, x \notin \mathrm{T}$ sinon $\mathrm{T}[i] = x$

```
1 . si |T| = 0 alors
2 .     renvoyer −1
3 . fin si
4 . m ← |T|/2
5 . si x = T[m] alors
6 .     renvoyer m
7 . sinon si x < T[m] alors
8 .     renvoyer Recherche_Dichotomique(T[0 : m],x)
9 . sinon
10 .    i ← Recherche_Dichotomique(T[m + 1 : |T|],x)
11 .    si i ≠ −1 alors
12 .        i ← i + m + 1
13 .    fin si
14 .    renvoyer i
15 . fin si
```

$$\begin{aligned} \mathrm{P}(n) : &\forall \mathrm{T}, \forall i\,,\ |\mathrm{T}| \leq n \text{ et } i = \texttt{Recherche\_Dichotomique}(\mathrm{T}, x) \implies \\ &0 \leq i < |\mathrm{T}| \text{ et } \mathrm{T}[i] = x, \text{ ou }, i = -1 \text{ et } x \notin \mathrm{T} \end{aligned}$$

❧ ❧ ❧

⚛ Exemples.

L'algorithme calculant le premier rang auquel la suite de Syracuse atteint 1 est *partiellement* correct.

❧ ❧ ❧

1.3.3 Complexité

Ressources utilisées par un programme #+attr$_{\text{latex}}$:options [label=▷]

- **Temps**
- *Espace mémoire*
- *Énergie*
- *Bande passante*
- *Autonomie de batterie*
- *Coût monétaire*
- …

❦ Définition.

L'étude de la complexité a pour but d'estimer la consommation en ressource d'un algorithme en fonction de la taille de ses entrées.

Pour calculer la *complexité temporelle asymptotique au pire cas* on estime en fonction de la taille de l'entrée :

- le nombre d'instructions (*temporelle*)
- lorsque la taille de l'entrée tends vers l'infini (*asymptotique*)
- pour l'exécution qui produit le plus d'instruction (*au pire cas*)

Autres complexités :

- *Complexité au mieux*
- *Complexité en moyenne*
- *Complexité amortie*

La complexité s'exprime en notation de Landau #+attr$_{\text{latex}}$:options [label=•]

— Asymptotiquement dominée par f : $O(f)$
— Asymptotiquement soumise par f : $\Omega(f)$
— Asymptotiquement dominée et soumise par f : $\Theta(f)$

Voici une liste des complexité que l'on peut classiquement trouver :

— $\Theta(1)$ (*Constante*) : Calcul arithmétique simple, boucle bornée par un entier fixé
— $\Theta(\log(n))$ (*Logarithmique*) : Algorithme par Dichotomie
— $\Theta(n)$ (*Linéaire*) : Parcours de chaque élément d'une structure, séquentiellement.
— $\Theta(n\log(n))$ (*Quasi-Linéaire*) : Algorithme par Dichotomie avec calculs linéaire à chaque division.
— $\Theta(n^2)$ (*Quadratique*) : Boucles imbriquées, Structure matricielle.
— $\Theta(n^\alpha)$, $\alpha \in \mathbb{N}$ (*Polynomiale*) : Plusieurs boucles imbriquées, calcul linéaires sur chaque élément d'une matrice.
— $\Theta(\alpha^n)$, $\alpha > 1$ (*Exponentielle*) : Parcours de tous les sous-ensembles possibles d'une structure séquentielle.
— $\Theta(n!)$, (*Factorielle*) : Parcours de toutes les permutations d'éléments d'une structure séquentielle.

	10	100	1 000	10 000	1 000 000	10^9
$\Theta(\log n)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{n})$	1ns	10ns	100ns	100ns	1mics	1ms
$\Theta(n)$	10ns	100ns	1mics	10mics	1ms	1s
$\Theta(n \log n)$	10ns	100ns	10mics	100mics	10ms	10s
$\Theta(n^2)$	100ns	10mics	1ms	100ms	10 min	10 ans
$\Theta(n^3)$	1mics	1ms	1s	10 min	10 ans	∞
$\Theta(2^n)$	1mics	∞	∞	∞	∞	∞

II Structures de Données Séquentielles

Les complexités précisées lors de la spécification sont celle de la structure de donnée Python

2.1 Type abstrait

Hors Programme

❦ Définition.

Un **type abstrait** donne la spécification abstraite d'un type de données :
- Liste des opérations
- Propriétés des opérations

⚛ Exemples.

Les conteneurs
- Création. *Paramètres* : ∅, crée un conteneur vide.
- Ajout. *Paramètres* : indice i, ajoute un élément dans le conteneur à l'indice i et décale l'indice de tous les éléments suivant de 1.
- Suppression. *Paramètres* : indice i, supprime l'élément d'indice i dans le conteneur et décale l'indice de tous les éléments suivant de -1.
- Accès. *Paramètres* : indice i, renvoie l'élément d'indice i.
- Taille. *Paramètres* : ∅, renvoie la taille d'un conteneur.

❧ ❧ ❧

Un type abstrait peu avoir *plusieurs* implémentations

L'implémentation d'un type abstrait se nomme une *structure de donnée* et s'obtient par ces deux opérations :
- ▷ Définition du type à l'aide des types standards
- ▷ Implémentation des opérations en respectant les spécifications du type abstrait

❦ Définition.

Un objet peut être *mutable* ou *immuable*. Si l'objet est immuable, les opérations ne modifie pas l'objet originel, elles renvoient une copie de l'objet sur laquelle a été effectuée l'opération. Si l'objet est mutable, les opérations modifient l'objet par effet de bord.

⚛ Exemples.

Immuables *Python* :
- *Chaînes de caractères*
- *Entiers*, *Flottants*, *Booléens*
- *Tuples*

Mutables *Python* :
- *Listes*

- *Dictionnaires*

❧ ❧ ❧

Le plus souvent, les opérations qui modifient l'objet lui-même sont implémentées en tant que méthodes de l'objet : `objet.operation(param1,param2, ... )`. *e.g.* `L.sort()`. Les opérations qui ne modifient l'objet sont implémentée en tant que fonctions : `operation(objet,param1,param2, ... )`. *e.g :* `sorted(L)`

2.2 Tableau

❦ Définition.

Structure séquentielle permettant de stocker un nombre arbitraire *fixé* d'éléments

Opérations :

— Création. *Paramètres* : taille, valeur d'initialisation. O(`taille`)
— Accès. *Paramètres* : indice. O(1)
— Modification. *Paramètres* : indice. O(1)

Implémentation *Python* (Hors Programme) :

```
import numpy as np
T = np.full(6,12) # Tableau de taille 6, rempli de 12
T[2] = T[4] ** 2 # Accès à l'indice 4, modification de l'indice 2
```

2.3 Vecteurs, Liste Python

2.3.1 Spécification

❦ Définition.

Structure séquentielle permettant de stocker un nombre *arbitraire* d'éléments, et d'y accéder en temps constant

Opérations :

— Création. *Paramètres* : taille, valeur d'initialisation. O(`taille`)
— Accès. *Paramètres* : indice. O(1)
— Modification. *Paramètres* : indice. O(1)
— Ajout à la fin. *Paramètres* : élément. O(1) (complexité amortie)
— Suppression à la fin. *Paramètres* : indice. O(1)(*complexitamortie*)
— Taille. *Paramètres* : ∅. O(1)

Implémentation *Python* :

```
L = [] # O(1) Vecteur de taille 0
L = [1,2,...,n] # O(n) Vecteur de taille n
L[2] = L[4] ** 2 # O(1) Accès à l'indice 4, modification de l'indice 2
L.append(21) # O(1) Ajout de 21 à la fin
a = L.pop(4) # O(1) Suppression de l'indice 4 et renvoie de sa valeur
b = len(L) # O(1) Longueur de L
```

2.4 File

❦ Définition.

Structure séquentielle permettant de stocker un nombre *arbitraire* d'éléments avec accès `FIFO`

Opérations :

— Création. *Paramètres* : ∅. O(1)
— Accès à la fin. *Paramètres* : ∅. O(1)
— Ajout au début. *Paramètres* : element. O(1)
— Taille. *Paramètres* : ∅. O(1)

Implémentation *Python* :

```
from collections import deque
F = deque() # O(1) Création d'une file vide
F.appendleft(12) # O(1) Ajout de 12 au début
a = F.pop() # O(1) Accès à la fin
b = len(F) # O(1) Longueur de F
```

2.5 Pile

2.5.1 Spécification

❦ Définition.

Structure séquentielle permettant de stocker un nombre *arbitraire* d'éléments avec accès `FIFO`

Opérations :

— Création. *Paramètres* : ∅. O(1)
— Accès à la fin. *Paramètres* : ∅. O(1)
— Ajout à la fin. *Paramètres* : élément. O(1)
— Taille. *Paramètres* : ∅. O(1)

Implémentation *Python* :

```
from collections import deque
P = deque() # O(1) Création d'une pil vide
P.append(12) # O(1) Ajout de 12 à la fin
a = P.pop() # O(1) Accès à la fin
b = len(P) # O(1) Longueur de P
```

2.6 Tableau associatif

❦ Définition.

Structure séquentielle permettant de stocker un nombre *arbitraire* d'associations clé, valeur.

Opérations :

— Création. *Paramètres* : ∅. O(1)

- Ajout. *Paramètres* : clé c et valeur v, une nouvelle clé c est créée et la valeur v est associée à la clé c dans la structure. O(1) (complexité amortie)
- Modification. *Paramètres* : clé c et valeur v, la valeur v' est associée à la valeur de la clé existante c. O(1) (complexité moyenne)
- Suppression. *Paramètres* : clé c, la clé c est supprimé, ainsi que la valeur qui lui est associée. O(1) (complexité moyenne)
- Recherche *Paramètres* : clé c, la valeur associée à la clé c est renvoyée. O(1) (complexité moyenne)
- Taille. *Paramètres* : ∅. O(1)

Implémentation *Python* par *Dictionnaires* :

```
d = {} # O(1) Dictionnaire vide
d['nadine'] = 12 # O(1) Ajout de la clé 'nadine' avec la valeur 12
d[12] = 'nadine' # O(1) Ajout de la clé 12 avec la valeur 'nadine'
n = d['nadine'] # O(1) Recherche de la clé 'nadine'
d[12] = 34 # O(1) Modification de la clé 12 avec la valeur 34
d[1] = 3.0 # O(1) Ajout de la clé 1 avec la valeur 1.0
del d[12] # O(1) Suppression de la clé 12
b = len(d) # O(1) Taille
k in d # O(1) Présence de la clé k
```

Il existe plusieurs implémentations possible des tableaux associatifs. En voici deux possibles :

- Tables de hachage :
  - Création. *Paramètres* : ∅. O(1)
  - Ajout. *Paramètres* : clé et valeur. O(1) (amortie, O(n) au pire)
  - Recherche *Paramètres* : clé. O(1) (en moyenne, O(n) au pire)
- Arbres équilibrés :
  - Création. *Paramètres* : ∅. O(1)
  - Ajout. *Paramètres* : clé et valeur. O(log(n)) au pire
  - Recherche *Paramètres* : clé. O(log(n)) au pire

Python utilise les *tables de hachage*.

❦ Définition.

Une *table de hachage* est un ensemble d'alvéoles indexées par des entiers qui contiennent un ensemble de paires de clé et valeur. L'ensemble d'alvéole est implémenté par un tableau et chaque alvéole par une liste.

Une *fonction de hachage* permet de convertir une clé en un index.

L'accès dans une table de hachage se fait en deux temps :

1. Conversion de la clé en un index à l'aide de la fonction de hachage
2. Recherche de la paire clé, valeur dans l'alvéole.

La complexité de la recherche est donc répartie en trois opérations :

1. Le hachage de la clé. *La complexité dépend du choix de la fonction.*
2. L'accès à l'alvéole. O(1)
3. La recherche dans l'alvéole. O(k) où k est la taille de l'alvéole.

On s'intéresse donc à trois facteurs pour une fonction de hachage :

— *rapidité* : complexité du hachage de la clé
— *espace d'arrivée* : nombre d'alvéoles et donc d'index
— *collisions* : probabilité que deux clés différentes soient hachées vers le même index.

❦ Définition.

Une fonction de hachage est dite *parfaite* si elle n'engendre aucune collision. Elle permet donc un accès en temps constant, mais utilise nécessairement autant de mémoire qu'il y a de clé possible.

Il est souvent plus intéressant de débuter avec une fonction de hachage ayant des collisions et d'augmenter la taille lorsque le nombre collision devient trop grand. On choisit alors une nouvelle fonction de hachage ayant moins de collision et on copie la table en utilisant la nouvelle table de hachage.

❦ Définition.

Le *facteur de compression* est défini comme $\frac{n}{k}$, n étant le nombre de paires clé, valeur et k nombre d'alvéoles. C'est ce facteur de compression qui détermine le moment où la table sera agrandie. Au-delà de 50% le risque de collision devient trop élevé.

⚛ Exemples.

Fonction de hachage des chaînes. Écriture de chaîne de caractère en base 256, en identifiant chaque caractère à son code ASCII. On peut ensuite travailler modulo n.

❧ ❧ ❧