



## Développement Efficace

# 1. introduction : structures de données, complexité

BUT INFO 2A

David Auger

IUT de Vélizy - UVSQ



## 1.2- Structure de données utilisées

Structures de données natives utilisées en python et Java :



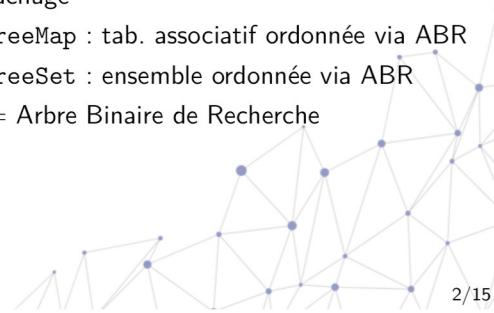
## 1.2- Structure de données utilisées

Structures de données natives utilisées en python et Java :

- ▶ list : tableau statique (amélioré)
- ▶ dict : tableau associatif via table de hachage
- ▶ set : ensemble dynamique via table de hachage

- ▶ ArrayList :: tableau statique (amélioré)
- ▶ HashMap : tableau associatif via table de hachage
- ▶ HashSet : ensemble dynamique via table de hachage
- ▶ TreeMap : tab. associatif ordonnée via ABR
- ▶ TreeSet : ensemble ordonnée via ABR

ABR = Arbre Binaire de Recherche



## 1.3- Structure de données

Une définition

**Structure de données** = façon d'organiser les données en mémoire pour les manipuler efficacement



3/15

## 1.3- Structure de données

Une définition

**Structure de données** = façon d'organiser les données en mémoire pour les manipuler efficacement

Opérations

Au minimum on veut les opérations :

- ▶ **ajouter** un élément dans la structure
- ▶ **extraire** un élément de la structure

Interface et Implémentation

On doit disposer d'une **interface ou API** (application programming interface), par exemple sous forme de méthodes

```
maStructure.ajouter(e)  
maStructure.extraire(val)
```

D'un autre côté, plusieurs **implémentations** de la structure (avec un tableau, avec une liste chaînée, etc.) sont possibles pour cette interface.



## 1.4- Primitives des SdD

### Extraction

L'extraction peut prendre différentes formes suivant comment l'élément est désigné :

- par sa valeur; ex :

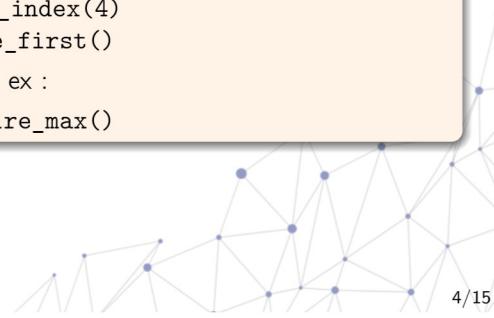
```
maStructure.supprimer(5)
```

- par sa position dans la structure (pointeur, indice); ex :

```
maStructure.remove_index(4)  
maStructure.remove_first()
```

- Par une propriété (valeur maximale, minimale, etc.); ex :

```
maStructure.extraire_max()
```



## 1.5- Primitives des SdD

Autre primitives courantes

- ▶ **taille/longueur** de la structure
- ▶ **rechercher** un élément/une valeur : est-il dans la structure ou non ?
- ▶ **nombre d'occurrences** d'un élément/valeur
- ▶ **valeur max/min** de la structure (sans extraire)
- ▶ **successeur/prédécesseur** d'un élémént

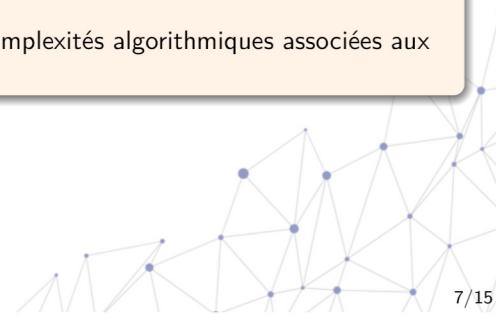




## 1.7- Objectifs du cours

### Objectifs

- ▶ comprendre les mécanismes à l'oeuvre derrière les principales structures de données classiques ;
- ▶ explorer de nouvelles structures de données ;
- ▶ être capable de choisir la structure de données la plus adaptée en fonction des traitements à effectuer ;
- ▶ et pour ceci connaître les ordres de grandeurs des complexités algorithmiques associées aux principales opérations de chaque structure.



## 1.8- Notion de complexité algorithmique

Pour évaluer l'efficacité d'un algorithme, on cherche à **quantifier le nombre total d'opérations élémentaires** effectuées lors de son exécution sur des données. Cette quantité dépend naturellement des données d'entrée, et en particulier de leur taille (souvent mesurée en mémoire ou en nombre d'éléments).

### Opération élémentaire

Une **opération élémentaire** est une action que l'on considère comme prenant un temps constant (noté "1"). Par exemple :

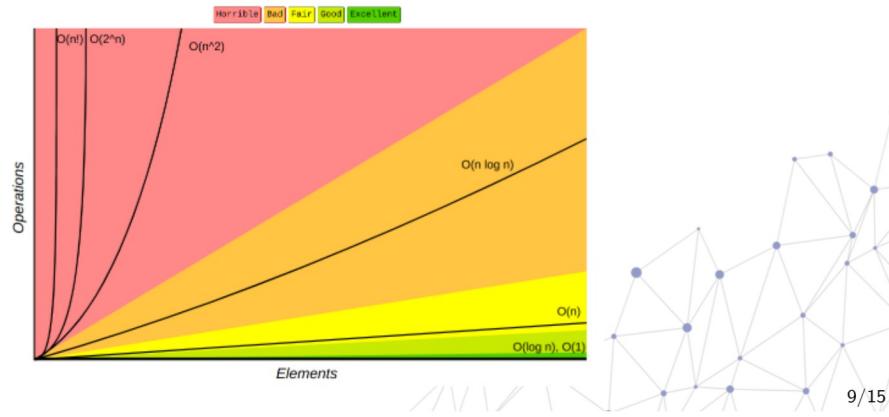
- ▶ changer la valeur d'une variable ;
- ▶ afficher quelque chose à l'écran ;
- ▶ dans certains cas, faire un calcul arithmétique simple comme une addition ou un modulo ;
- ▶ comparer deux valeurs (plus généralement appliquer un opérateur) ;
- ▶ accéder à une case d'un tableau.



### 1.9- La complexité algorithmique est une fonction

La complexité algorithmique est mesurée en fonction de la taille en mémoire prise par l'entrée de l'algorithme. Exemples :

- si l'entrée est une chaîne de caractère, on mesurera le nombre d'opérations effectuées **dans le pire des cas** en fonction de la longueur  $n$  de la chaîne ;
  - si l'entrée est un entier  $k$ , on mesurera le nombre d'opérations effectuées **dans le pire des cas** en fonction du nombre de bits nécessaires pour écrire  $k$  en base 2, soit environ  $n = \log_2 k$ .



## 1.10- Un exemple

## Exemple

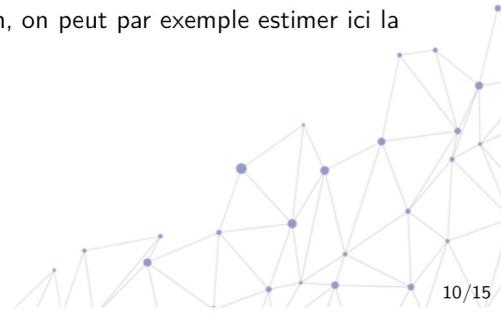
```
def cherche0(tab):
    trouvé = False
    i=0
    while i < len(tab) and tab[i] != 0:
        i+=1
    return i < len(tab)
```

Suivant ce qui est considéré comme une opération ou non, on peut par exemple estimer ici la complexité algorithmique par la fonction

$$c(n) = 4n + 3$$

Pourquoi ? Ou serait-ce

$$c(n) = 5n + 2?$$



## 1.11- Ordre de grandeur



Le fait de trouver  $4n + 3$  ou  $5n + 12$  ou  $n + 23$  dépend exactement des détails de l'implémentation et des opérations qu'on considère. dans ce cas, on retiendra uniquement que l'ordre de grandeur est  $n$  (et non pas  $n^2$ ,  $2^n$  etc).

On va formaliser cette notion d'ordre de grandeur à l'aide des notations "grand O" de domination asymptotique.



## 1.12- Notations de domination asymptotique

Soit  $F$  l'ensemble des fonctions de  $\mathbb{N}$  dans  $\mathbb{R}^+$ .

### Notation Grand O

Si  $f \in F$ , on note  $O(f)$  l'ensemble des fonctions  $g \in F$  telles que

$$\exists C > 0, \exists n_0 \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq C \cdot f(n)$$

C'est l'ensemble des fonctions asymptotiquement dominées par  $f$ .



## 1.12- Notations de domination asymptotique

Soit  $F$  l'ensemble des fonctions de  $\mathbb{N}$  dans  $\mathbb{R}^+$ .

### Notation Grand O

Si  $f \in F$ , on note  $O(f)$  l'ensemble des fonctions  $g \in F$  telles que

$$\exists C > 0, \exists n_0 \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq C \cdot f(n)$$

C'est l'ensemble des fonctions asymptotiquement dominées par  $f$ .

On abrège souvent la notation des fonctions, i.e. on note  $n^2$  au lieu de  $n \mapsto n^2$ .

Exemple :

$$n + 10 \in O(n^2)$$

car avec  $C = 2$  et  $n_0 = 4$

si  $n \geq 4$  alors

$$n + 10 \leq n^2 + n^2 = 2n^2$$



## 1.13-. Critères avec limites

### Théorèmes

Soient  $f, g \in F$ .

► Si

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \ell < +\infty$$

existe et est fini, alors  $g \in O(f)$ ;

► Si  $0 < \ell < +\infty$  alors  $g \in O(f)$  et  $f \in O(g)$  puisque  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{\ell} < +\infty$

► Si

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = +\infty$$

alors  $g \notin O(f)$  mais  $f \in O(g)$  puisque  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 < +\infty$



## 1.14- Comparaison des complexités

- ▶  $O(1)$  : temps constant
- ▶  $O(\log n)$  : temps logarithmique
- ▶  $O(\sqrt{n})$
- ▶  $O(n)$  : temps linéaire
- ▶  $O(n \log n)$  : temps des tris par comparaison optimaux
- ▶  $O(n\sqrt{n})$
- ▶  $O(n^2)$  : temps quadratique
- ▶  $O(n^3)$  : temps cubique
- ▶ ...
- ▶  $O(n^k)$  : temps polynomial
- ▶ ...
- ▶  $O(2^n)$  : temps exponentiel
- ▶  $O(3^n)$  : temps exponentiel
- ▶ ...
- ▶  $O(p^n)$
- ▶ ...
- ▶  $O(2^{n \log n}) = O(n!)$  : temps factoriel



## 1.15- Bilan



### Bilan

Evaluer la complexité d'un algorithme, c'est donner l'ordre de grandeur (avec un  $O()$ ) du nombre d'opérations élémentaires effectuées par l'algorithme dans le pire des cas, pour chaque taille fixée  $n$  des données en entrée.

