Algorithmique des graphes

0 — Rappels

Marie-Pierre Béal (Cours d'Anthony Labarre)

Université
Gustave Eiffel

Organisation du cours

Déroulement

- 12 séances de CM, 12 séances de TD ;
- Quelques mini-rendus et un projet en Python ;
- . . . et bien sûr un examen.

Supports de cours et bibliographie

- Ces transparents.
- Introduction to Algorithms, 4th Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Cambridge MIT Press, 2022.

Organisation du cours

Les chargés de TD

- Carl-Fredrik Nyberg Brodda
- Claire David
- Cyril Nicaud
- Johan Thapper

Concernant les évaluations

- Une grande partie de l'évaluation de vos programmes se fera à l'aide d'un correcteur automatique ;
- Il est donc **critique** de suivre les consignes données à la lettre :
  - respecter le nom des fonctions ;
  - respecter la signature des fonctions ;
  - ne pas confondre fonction et méthode ;
  - respecter le format d'entrée et de sortie ;
  - ...
- Des jeux de tests vous seront parfois fournis pour vous guider, **mais cela ne vous dispense pas d'écrire les vôtres !**

Avant de rentrer dans le vif du sujet

Commençons par quelques rappels rapides concernant les pré-requis :

1. Complexité algorithmique
2. Programmation orientée objet en Python
3. Pseudocode
4. Techniques de preuves

Motivations

- Un algorithme doit être correct et **efficace**, c'est-à-dire :
  - rapide, et
  - économe en ressources (espace de stockage, mémoire, ...).
- Mesurer le temps de calcul réel nécessite d'implémenter (et tester, et débugger, et optimiser, ...) les algorithmes ;
- Pour éviter cela, on évalue le temps d'exécution théorique à l'aide de la complexité algorithmique, qui donnera une **approximation** du temps de calcul en fonction de la taille des données, représentée par la notation $O(\cdot)$.

La notation $O(\cdot)$

Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

La notation $O(\cdot)$

Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près.

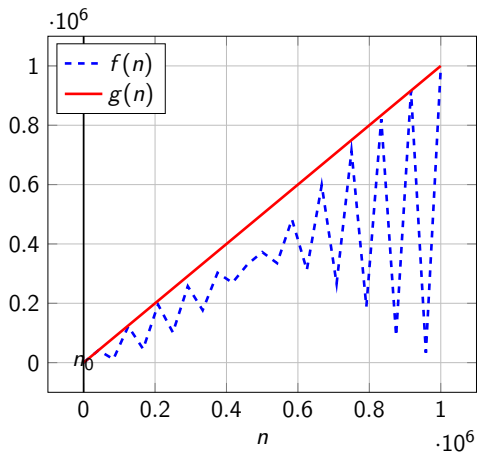
La notation $O(\cdot)$

Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près.

Exemple 1 (quelques cas où $f(n) = O(g(n))$)



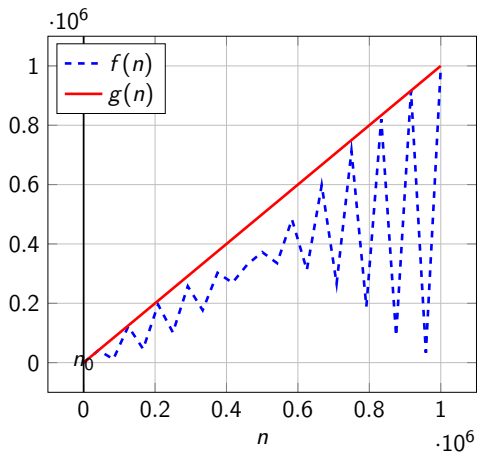
La notation $O(\cdot)$

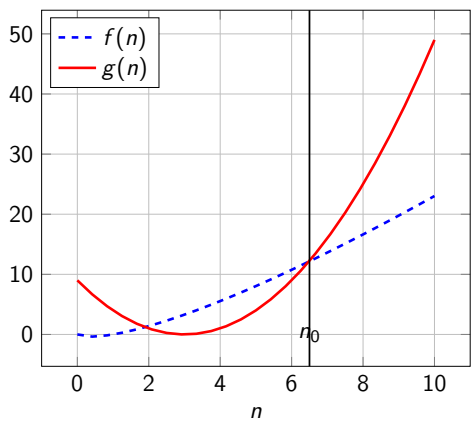
Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près.

Exemple 1 (quelques cas où $f(n) = O(g(n))$)





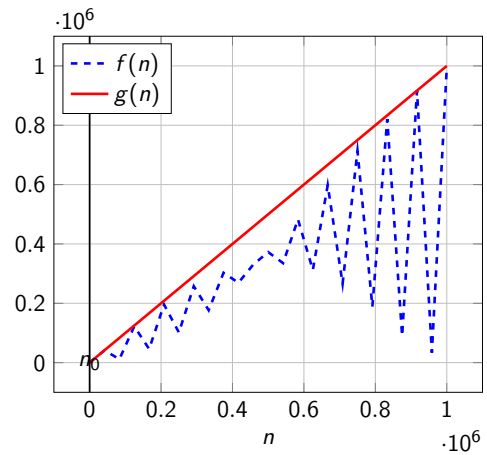
La notation $O(\cdot)$

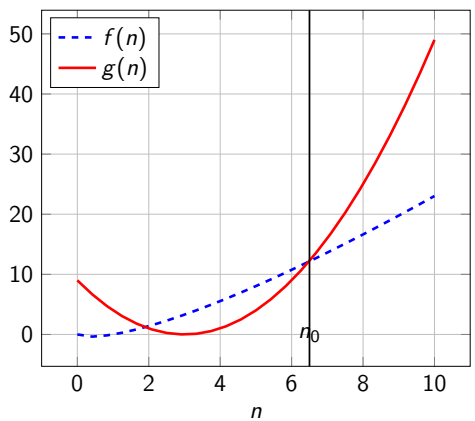
Une fonction $f(n)$ est **en** $O(g(n))$ **("en grand O de** $g(n)$**")** si :

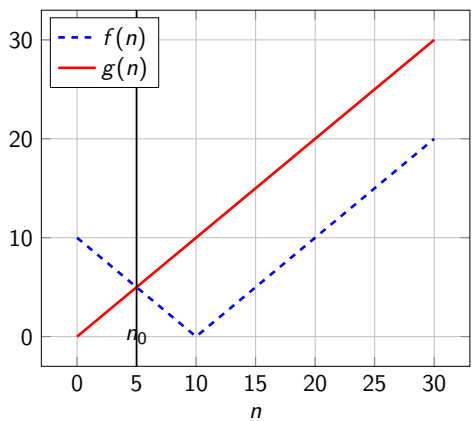
$$\exists\ n_0 \in \mathbb{N}, \exists\ c \in \mathbb{R}, \forall\ n \geq n_0 : \qquad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près.

Exemple 1 (quelques cas où $f(n) = O(g(n))$)







Complexité d'un algorithme

Définition 1

La **complexité** (dans le pire cas) d'un algorithme est la mesure **asymptotique** de son temps d'exécution **dans le pire cas**. Elle s'exprime à l'aide de la notation $O(\cdot)$ en fonction de la taille des données reçues en entrée.

Les deux précisions sur le caractère de cette mesure importent :

Complexité d'un algorithme

Définition 1

La **complexité** (dans le pire cas) d'un algorithme est la mesure **asymptotique** de son temps d'exécution **dans le pire cas**. Elle s'exprime à l'aide de la notation $O(\cdot)$ en fonction de la taille des données reçues en entrée.

Les deux précisions sur le caractère de cette mesure importent :

1. **asymptotique** : on s'intéresse à des données très grandes ;

Complexité d'un algorithme

Définition 1

La **complexité** (dans le pire cas) d'un algorithme est la mesure **asymptotique** de son temps d'exécution **dans le pire cas**. Elle s'exprime à l'aide de la notation $O(\cdot)$ en fonction de la taille des données reçues en entrée.

Les deux précisions sur le caractère de cette mesure importent :

1. **asymptotique** : on s'intéresse à des données très grandes ;
2. **"dans le pire cas"** : on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre.

Calcul de la complexité d'un algorithme

On fait les hypothèses suivantes :

- chaque **instruction basique** (affectation d'une variable de type basique ou comparaison de deux types basiques, $+$, $-$, $*$, $/$, ...) consomme un temps constant, noté $O(1)$;

Calcul de la complexité d'un algorithme

On fait les hypothèses suivantes :

- chaque **instruction basique** (affectation d'une variable de type basique ou comparaison de deux types basiques, $+$, $-$, $*$, $/$, ...) consomme un temps constant, noté $O(1)$;
- chaque **itération** d'une boucle rajoute la complexité de ce qui est effectué dans le corps de cette boucle ;

Calcul de la complexité d'un algorithme

On fait les hypothèses suivantes :

- chaque **instruction basique** (affectation d'une variable de type basique ou comparaison de deux types basiques, $+$, $-$, $*$, $/$, ...) consomme un temps constant, noté $O(1)$;
- chaque **itération** d'une boucle rajoute la complexité de ce qui est effectué dans le corps de cette boucle ;
- chaque **appel de fonction** rajoute la complexité de cette fonction ;

Calcul de la complexité d'un algorithme

On fait les hypothèses suivantes :

- chaque **instruction basique** (affectation d'une variable de type basique ou comparaison de deux types basiques, $+$, $-$, $*$, $/$, ...) consomme un temps constant, noté $O(1)$;
- chaque **itération** d'une boucle rajoute la complexité de ce qui est effectué dans le corps de cette boucle ;
- chaque **appel de fonction** rajoute la complexité de cette fonction ;
- pour obtenir la complexité de l'algorithme, on additionne le tout.

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 2 (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 2 (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 2 (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 2 (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

Simplifications

On aura aussi recours aux simplifications suivantes (justifiées par la définition de $O(\cdot)$) :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants.

Exemple 2 (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

et on a donc $g(n) = O(n^3)$.

Opérations en séquence

On additionne les complexités d'opérations en séquence :

$$O(f_1(\cdot)) + O(f_2(\cdot)) = O(f_1(\cdot) + f_2(\cdot))$$

Opérations en séquence

On additionne les complexités d'opérations en séquence :

$$O(f_1(\cdot)) + O(f_2(\cdot)) = O(f_1(\cdot) + f_2(\cdot))$$

Pareil pour les branchements conditionnels :

si [condition] **alors :** $O(g(\cdot))$
instructions (1) $O(f_1(\cdot))$
sinon
instructions (2) $O(f_2(\cdot))$

$\Big\} = O(g(\cdot) + f_1(\cdot) + f_2(\cdot))$

Opérations en séquence

On additionne les complexités d'opérations en séquence :

$$O(f_1(\cdot)) + O(f_2(\cdot)) = O(f_1(\cdot) + f_2(\cdot))$$

Pareil pour les branchements conditionnels :

```
si [condition] alors :                    O(g(·))
  ## instructions (1)                     O(f1(·))
sinon
  ## instructions (2)                     O(f2(·))
```

$\Big\}$ $= O(g(\cdot) + f_1(\cdot) + f_2(\cdot))$

Les règles de simplification vues plus haut s'appliqueront ensuite. En particulier, une somme constante d'opérations constantes sera constante : $O(1) + O(1) = O(1)$.

Boucles

La complexité d'une boucle se calcule comme suit :

```
## si on a m itérations
tant que [condition] faire
 ## instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

Boucles

La complexité d'une boucle se calcule comme suit :

```
## si on a m itérations
tant que [condition] faire
 ## instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

- Si deux boucles sont en séquence, on additionne leurs complexités ;

Boucles

La complexité d'une boucle se calcule comme suit :

```
## si on a m itérations
tant que [condition] faire
 ## instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

- Si deux boucles sont en séquence, on additionne leurs complexités ;
- Si deux boucles sont imbriquées :

Boucles

La complexité d'une boucle se calcule comme suit :

```
## si on a m itérations
tant que [condition] faire
 ## instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

- Si deux boucles sont en séquence, on additionne leurs complexités ;
- Si deux boucles sont imbriquées :
  - si elles sont **indépendantes**, on multiplie leurs complexités ;

Boucles

La complexité d'une boucle se calcule comme suit :

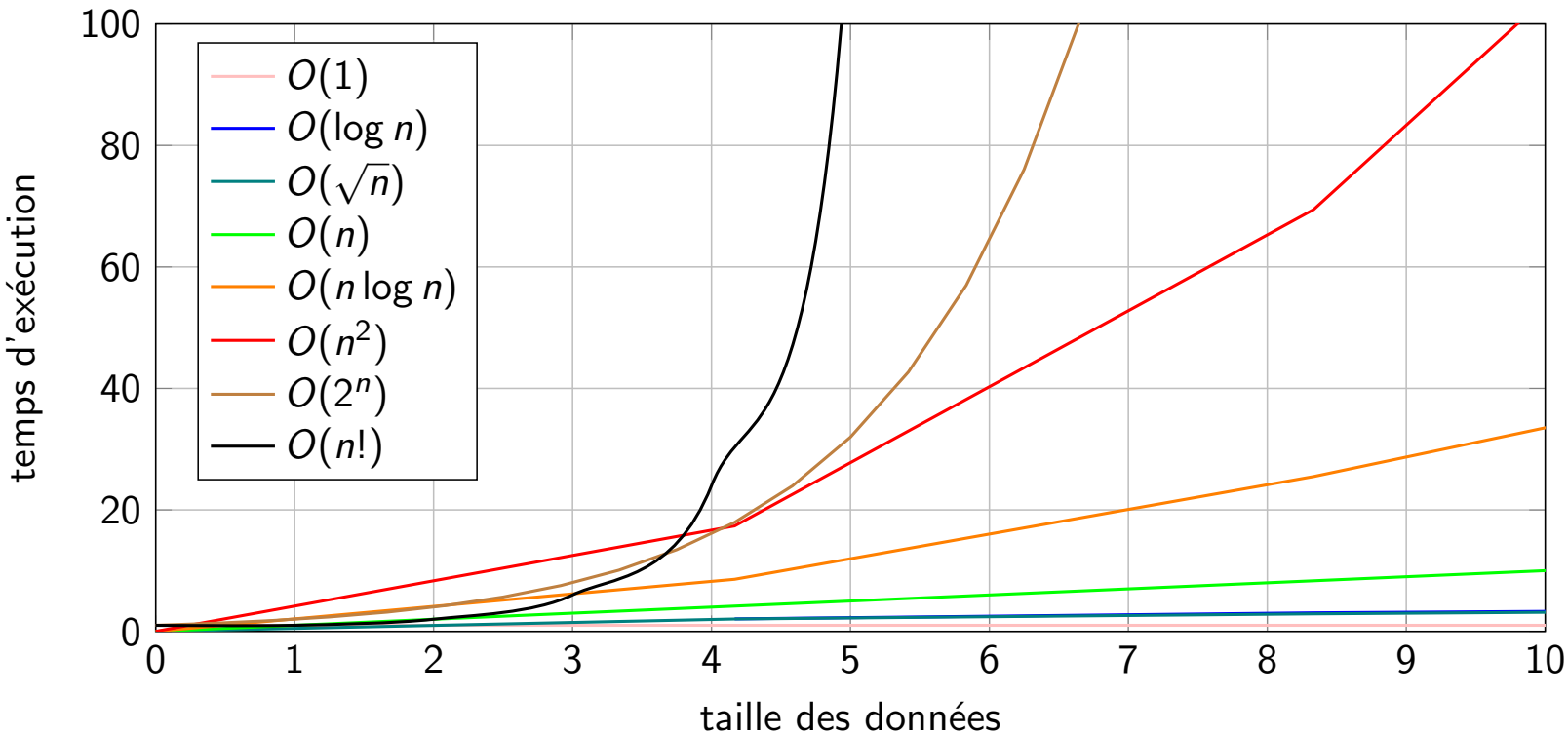
```
## si on a m itérations
tant que [condition] faire
 ## instructions
```

$$\left.\begin{array}{l} O(g(\cdot)) \\ O(f(\cdot)) \end{array}\right\} = O(m * (g(\cdot) + f(\cdot)))$$

- Si deux boucles sont en séquence, on additionne leurs complexités ;
- Si deux boucles sont imbriquées :
  - si elles sont **indépendantes**, on multiplie leurs complexités ;
  - sinon, on devra procéder autrement (voir exemples concrets plus tard).

Complexités fréquentes et comparaisons

On regroupe les fonctions équivalentes ($f = O(g)$ et $g = O(f)$) dans une même **classe**.



En général, on préfère les algorithmes de complexité minimale.

Cours de Python objet

https://python.sdv.univ-paris-diderot.fr/

Classe basique en Python

Exemple 3

```
class Promo(object):
    def __init__(self):  # constructeur
        self.etudiants = dict()

    def ajouter_etudiant(self, nom):
        if nom not in self.etudiants:
            self.etudiants[nom] = dict()

    def moyenne(self, nom):
        return sum(self.etudiants[nom]) / len(self.etudiants[nom])
```

- `self` ≡ `this` en Java ou C++ ;
- `__init__` = constructeur ;

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :
  - dans la classe : `self.methode(param)` ;

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :
  - dans la classe : `self.methode(param)` ;
  - en dehors : `mon_instance.methode(param)`.

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :
  - dans la classe : `self.methode(param)` ;
  - en dehors : `mon_instance.methode(param)`.
- Pour les données :

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :
  - dans la classe : `self.methode(param)` ;
  - en dehors : `mon_instance.methode(param)`.
- Pour les données :
  - `self.data` : données membres, accessibles dans toute la classe ;

Méthodes et données membres

- On définit les méthodes comme les fonctions avec `def`, mais :
  1. on doit les indenter pour les mettre dans la classe ;
  2. le premier paramètre doit toujours être `self` ;
- Appeler une méthode :
  - dans la classe : `self.methode(param)` ;
  - en dehors : `mon_instance.methode(param)`.
- Pour les données :
  - `self.data` : données membres, accessibles dans toute la classe ;
  - `data` sans `self` : variable locale, accessible seulement dans la méthode qui la contient.

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :
  1. paramètres avec valeurs par défaut ;

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :
  1. paramètres avec valeurs par défaut ;
  2. arguments variables (`def methode(self, *args, **kwargs)`) ;

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :
  1. paramètres avec valeurs par défaut ;
  2. arguments variables (`def methode(self, *args, **kwargs)`) ;
- On simule les champs privés à l'aide du préfixe `__` :

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :
  1. paramètres avec valeurs par défaut ;
  2. arguments variables (`def methode(self, *args, **kwargs)`) ;
- On simule les champs privés à l'aide du préfixe `__` :

Exemple 4 (champs "privés")

```
>>> class MaClasse(object):
...     def __init__(self):
...         self.public = 1
...         self.__prive = 0
...
>>> x = MaClasse()
>>> print(x.public)
1
>>> print(x.__prive)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MaClasse' object has no attribute '__prive'
```

Particularités des objets en Python

- Pas de surcharge, y compris pour `__init__` ; alternatives :
  1. paramètres avec valeurs par défaut ;
  2. arguments variables (`def methode(self, *args, **kwargs)`) ;
- On simule les champs privés à l'aide du préfixe `__` :

Exemple 4 (champs "privés")

```
>>> class MaClasse(object):
...     def __init__(self):
...         self.public = 1
...         self.__prive = 0
...
>>> x = MaClasse()
>>> print(x.public)
1
>>> print(x.__prive)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MaClasse' object has no attribute '__prive'
```

- Pas de *getters* ou *setters* ;

Intérêt de la POO dans ce cours

- Nos buts :
  - pouvoir supposer qu'il existe une classe `Graphe` avec une certaine interface ;
  - décrire des algorithmes s'exécutant de la même manière quelle que soit l'implémentation.
- On verra plus loin qu'il existe plusieurs manières d'implémenter les graphes ;
- Elles auront un impact direct sur la complexité de nos algorithmes.

Programmation générique

- Nos fonctions devront parfois renvoyer un sous-graphe du même type que celui d'entrée ;
- On y arrivera avec la syntaxe suivante :

Exemple 5

```
def mon_algo(un_graphe):
    resultat = type(un_graphe)()
    # ...
    return resultat
```

Pseudocode

Les algorithmes seront exprimés en *pseudocode*. Par exemple :

Algorithme 1 : TrouverMinimum(T)

Entrées : un tableau T non vide.
Sortie : le plus petit élément de T.

```
1 minimum ← T[0];
2 pour i allant de 1 à longueur(T) − 1 faire
3 |   si T[i] < minimum alors minimum ← T[i];
4 renvoyer minimum;
```

Pseudocode

Les algorithmes seront exprimés en *pseudocode*. Par exemple :

Algorithme 1 : TrouverMinimum(T)

Entrées : un tableau T non vide.
Sortie : le plus petit élément de T.

```
1 minimum ← T[0];
2 pour i allant de 1 à longueur(T) − 1 faire
3 |   si T[i] < minimum alors minimum ← T[i];
4 renvoyer minimum;
```

Instructions pour le pseudocode :

Pseudocode

Les algorithmes seront exprimés en *pseudocode*. Par exemple :

Algorithme 1 : TrouverMinimum(T)

Entrées : un tableau T non vide.
Sortie : le plus petit élément de T.

```
1 minimum ← T[0];
2 pour i allant de 1 à longueur(T) − 1 faire
3 |   si T[i] < minimum alors minimum ← T[i];
4 renvoyer minimum;
```

Instructions pour le pseudocode :

- éviter au maximum les détails d'implémentation liés au langage (allocation mémoire, pointeurs, . . .),

Pseudocode

Les algorithmes seront exprimés en *pseudocode*. Par exemple :

Algorithme 1 : TrouverMinimum(T)

Entrées : un tableau T non vide.
Sortie : le plus petit élément de T.

```
1 minimum ← T[0];
2 pour i allant de 1 à longueur(T) − 1 faire
3 |   si T[i] < minimum alors minimum ← T[i];
4 renvoyer minimum;
```

Instructions pour le pseudocode :

- éviter au maximum les détails d'implémentation liés au langage (allocation mémoire, pointeurs, ...),
- être suffisamment précis pour que la traduction de l'algorithme vers n'importe quel vrai langage soit facile et sans ambiguïté.

Conventions

Le format que nous adopterons sera le suivant :

1. algo_standard(données) : algorithme "trivial" pas nécessairement vu explicitement ;

Conventions

Le format que nous adopterons sera le suivant :

1. algo_standard(données) : algorithme "trivial" pas nécessairement vu explicitement ;
2. AlgoDuCours(données) : algorithme vu dans ce cours ;

Conventions

Le format que nous adopterons sera le suivant :

1. algo_standard(données) : algorithme "trivial" pas nécessairement vu explicitement ;
2. AlgoDuCours(données) : algorithme vu dans ce cours ;
3. description des problèmes :

Conventions

Le format que nous adopterons sera le suivant :

1. algo_standard(données) : algorithme "trivial" pas nécessairement vu explicitement ;
2. AlgoDuCours(données) : algorithme vu dans ce cours ;
3. description des problèmes :
   - **Entrée(s) :** ce que l'on reçoit, avec les hypothèses nécessaires ;

Conventions

Le format que nous adopterons sera le suivant :

1. algo_standard(données) : algorithme "trivial" pas nécessairement vu explicitement ;
2. AlgoDuCours(données) : algorithme vu dans ce cours ;
3. description des problèmes :
   - **Entrée(s) :** ce que l'on reçoit, avec les hypothèses nécessaires ;
   - **Sortie / But / Question / Résultat :** le résultat, avec les hypothèses nécessaires.

Techniques de preuves

On devra prouver que nos algorithmes sont corrects, ainsi que certaines propriétés. Généralement, on procèdera :

1. **par induction** :

Techniques de preuves

On devra prouver que nos algorithmes sont corrects, ainsi que certaines propriétés. Généralement, on procèdera :

1. **par induction** :
    1. prouver qu'une propriété P est vraie pour un cas de base ;

Techniques de preuves

On devra prouver que nos algorithmes sont corrects, ainsi que certaines propriétés. Généralement, on procèdera :

1. **par induction** :
   1. prouver qu'une propriété P est vraie pour un cas de base ;
   2. puis prouver que si P est vraie pour le cas n, elle l'est aussi pour le cas $n+1$;

Techniques de preuves

On devra prouver que nos algorithmes sont corrects, ainsi que certaines propriétés. Généralement, on procèdera :

1. **par induction** :
   1. prouver qu'une propriété P est vraie pour un cas de base ;
   2. puis prouver que si P est vraie pour le cas n, elle l'est aussi pour le cas $n+1$;
2. **par contraposition** : si prouver que $A \Rightarrow B$ est difficile, on tente plutôt de prouver que $\neg B \Rightarrow \neg A$, ce qui est équivalent (cf. cours de logique).

Techniques de preuves

On devra prouver que nos algorithmes sont corrects, ainsi que certaines propriétés. Généralement, on procèdera :

1. **par induction** :
   1. prouver qu'une propriété P est vraie pour un cas de base ;
   2. puis prouver que si P est vraie pour le cas n, elle l'est aussi pour le cas $n+1$;
2. **par contraposition** : si prouver que $A \Rightarrow B$ est difficile, on tente plutôt de prouver que $\neg B \Rightarrow \neg A$, ce qui est équivalent (cf. cours de logique).
3. **par l'absurde (ou contradiction)** : s'il est difficile de prouver que P est vraie, prouvons plutôt qu'il est impossible que P soit fausse ;